Boomslang Final Report

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

April 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.

The GitHub repository for Boosmlang can be found at https://github.com/dsteiner93/boomslang.

2 Boomslang Tutorial

2.1 Compiling programs

To compile a program, use the boomc utility. ./boomc helloworld.boom will compile the contents of helloworld.boom to LLVM IR. Using the flag -r with boomc as in ./boomc -r helloworld.boom will compile the program and automatically execute it in one step.

To compile the compiler, run make inside the src directory. This will produce a boomslang.native program that can be used to generate graphviz representations of the abstract syntax tree or semantically checked abstract syntax tree, as well as LLVM IR.

To run the test suite, run “make test” inside the src directory.

The Boomslang compiler requires OCaml and LLVM to be installed on the compiling machine. The following Docker image can also be used: https://hub.docker.com/r/columbiasedwards/plt

2.2 Writing programs

2.2.1 Basics

Boomslang has the following types built in: int, long, float, char, string, and boolean. Users may define their own types (called classes), as well as make arrays of any type. In addition to the previously mentioned types, functions may return a special “void” type, indicating there is no return value, and objects may be assigned NULL. (No type other than objects may be NULL.)

Here are some examples of how to use the types in Boomslang:

```
int a = 5
long b = 100L
float c = 2.2
char d = 'd'
string str = "foo"
boolean is_boomslang_a_good_language = true
MyObject my_object = NULL
int[] arr = [1, 2, 3, 4, 5]
```

Arrays can also be initialized with a default value, similar to the “new” keyword in Java. Once initialized, you can access their contents using a standard syntax. The length of an array is obtained using the built-in len keyword.

```
int[] arr = default[10]
arr[5] = 5
int array_len = len(arr)
```

Finally, once a variable is properly be initialized, it can be updated to another value, as long as that value has the same type as the original initialization.

```
int a = 5
a = 10  # this is legal
string str = "foo"
str = 10  # this is illegal
```

To inspect output of the program, the println() built-in function can be used. println() is polymorphic and can take any type. If the type is not a string, it will be automatically converted to a string.

```
# All of these work
println("Hello, world")
println(10)
println(10.0)
println(true)
```

Comments use the following syntax:

```
# This is a single-line comment
/# This a multi-line
comment #/
```

Our language supports the following mathematical and boolean operators. Ints, longs, and floats can be automatically coerced to each other in binary operators.

```
+ - * / % == > < >= <= or and not

int a = 5 + 10  # 15
float b = 2.0 + 2  # 4.0
boolean test = 10 > 5  # true

boolean foo = true
boolean not_foo = not foo
boolean another_bool = foo and not_foo or foo
```

Boomslang also supports the following convenience operators for performing updates:

```
+= -= *= /=

int a = 5
a += 5  # a is now 10. this line is equivalent to a = a + 5
```

2.2.2 Control flow

Boomslang contains two types of control flow statements: if/elif/else and loops. Ifs follow the same syntax and semantics as Python, meaning all of the following are supported:

```
if a == 0:
  println(a)

if a == 0:
  println("a == 0")
else:
  println("a != 0")

if a == 0:
  println("a == 0")
```

```
elif a == 1:
	println("a == 1")
else:
	println("a != 0")

if a == 0:
	println("a == 0")
elif a == 1:
	println("a == 1")
elif a == 2:
	println("a == 2")
else:
	println("a != 0")

if a == 0:
	println("a == 0")
elif a == 1:
	println("a == 1")
elif a == 2:
	println("a == 2")
```

Note that for **all** indentation in the language **tabs must be used** with **exactly one tab per indentation level**. Markdown and LaTeX support for tab characters is limited and can vary so copy/pasting from this tutorial may not work as tabs get auto-converted to spaces.

Loops in Boomslang use a novel syntax not seen in any other known programming language. Loops can be defined in two different ways using the “loop” keyword.

```
int i = 0
loop i += 1 while i < 100:
	println(i)

int i = 0
loop while i < 100:
	println(i)
	i += 1
```

2.2.3 Functions

By convention, function names and variable names should be `snake_case` rather than camelCase. By necessity, class names must start with an uppercase letter and only contain letters (no numbers). **Also by necessity, all indentation must be done using tabs. Spaces will not work.**

Functions are defined in a syntax similar to a strongly typed version of Python. A unique aspect of the syntax is that the return type is indicated via the “returns” keyword. Here are some examples of functions in Boomslang:

```
def my_func(int a) returns int:
	return a

def print_hello_world():
	println("hello world")

def gcd(int a, int b) returns int:
	loop while a != b:
		if a > b:
			a = a - b
		else:
    b = b - a
  return a
```

Functions do not need to be defined before use. Functions are allowed to be self-recursive or mutually-recursive with other functions.

2.2.4 Classes

Class basics While Boomslang does not require all functions and variables to be part of a class, it does support classes and object-oriented features.

Fields within a class always appear under the header “static”, “required”, or “optional”. These headers must always appear in that order (static, then required, then optional). Any or all of the three can be omitted for brevity.

A basic class looks like this:

```
class MyClass:
  static:
    int a = 0

  required:
    string b

  optional:
    boolean c = true

  def my_method() returns string:
    return self.b
```

All fields and methods on a class used within a class must always be prefixed with the “self” keyword. Objects are instantiated using the following syntax:

```
MyClass my_class1 = MyClass("new string")
my_class1.b  # "new_string"
my_class1.c  # true

MyClass my_class2 = MyClass("other string", false)
my_class2.b  # "other string"
my_class2.c  # false
```

Objects can also be initialized as NULL. Only the object type in Boomslang may be NULL.

```
MyClass foo = NULL  # this is legal
foo.b  # this will throw a NullPointerException
```

How do the above instantiations work, considering there was no constructor defined by the user? Boomslang automatically generates default constructors for dataclasses. Two constructors will be generated: One that takes just the required fields and initializes the optional ones to their default, and one that takes all of the fields (both optional and required). So for the above class, the following two constructor methods are automatically added to the class.

```
def construct(string b):
  self.b = b
  self.c = true

def construct(string b, boolean c):
  self.b = b
  self.c = c
```

Constructors can be overwritten using the “construct” keyword. Objects can also be printed, which will call the object’s `to_string()` method. This is a special function that takes no arguments and returns a string. By default, Boomslang will add a meaningful `to_string` to each user-defined class so the user doesn’t have to. However, similar to constructors, the user can overwrite it if they wish.

```
println(my_class1)  # this will print the contents of all the class fields

/#
MyClass:
a:0
b:new string
c:true
#/
```

Operator overloading for objects Boomslang supports the standard way to call methods on an object:

```
myobject.mymethod()
```

In addition to this, users can overwrite built-in binary operators that work on primitives to work on objects as well. This is especially useful for overwriting the double equals (==) function.

```
class IntLinkedList:
  required:
    int element

  optional:
    IntLinkedList next = NULL

  def _==(IntLinkedList other) returns boolean:
    # Custom defined equality function
    if self == NULL:
      return other == NULL
    elif other == NULL:
      return false
    else:
      return (self.element == other.element) and (self.next == other.next)

IntLinkedList a = IntLinkedList(1)
IntLinkedList b = IntLinkedList(1)
IntLinkedList c = IntLinkedList(2)
println(a == b)  # true
println(b == c)  # false
```

Generic classes The int linked list class above is nice, but it would be tedious to rewrite the linked list class for every possible type of linked list. To make this more convenient, the user is allowed to define generic classes. Generic classes cannot be instantiated directly as objects, but can be used to declare classes very succinctly. The following is a generic form of the above linked list.

```
class LinkedList[T]:
  required:
    T element

  optional:
    LinkedList next = NULL

  def _==(LinkedList other) returns boolean:
    # Custom defined equality function
    if self == NULL:
      return other == NULL
    elif other == NULL:
      return false
    else:
      return (self.element == other.element) and (self.next == other.next)
```

To use the generic class, it must be used in a real class declaration, using the following syntax:

```
class IntLinkedList = LinkedList(int)
class StringLinkedList = LinkedList(string)

IntLinkedList my_list1 = IntLinkedList(1)
StringLinkedList my_list2 = StringLinkedList("string")
```

3 Language Reference Manual

3.1 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.

3.1.1 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 default`

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`

3.1.2 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.

Primitive type tokens The primitive data type tokens are: `int long float boolean char string void`

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']*`

Array types Array types consist of a type token followed by a [, followed by a closing]. Arrays can be nested, so an array of arrays is legal.

3.1.3 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' '_']*`

3.1.4 Operators

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

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

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.

3.1.5 Literals

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

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']+`

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.

Floating point literals Floating point literals consist of zero or more numbers, followed by a period, followed by one or more numbers. We do not support any syntax for numbers using `e`. The regular expression for float literals is

```
['0'-'9']+('.'['0'-'9']+)? | '.'['0'-'9']+
```

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.

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

```
'\'' [' '-'~'] '\''
```

String literals String literals consist of a double quote, followed by text, followed by a double quote. In order to put double quotes in string characters, they must be escaped with a backslash. The precise regex for string literals is

```
'"' [^'"''\\']* ('\\'_[^'"''\\']* )* '"'
```

3.1.6 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. Comments in Boomslang are expected to be indented correctly with the rest of the code.

3.1.7 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. They are also used for generic classes.
- 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.

3.1.8 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.

3.2 Syntax

3.2.1 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`.

3.2.2 Types

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.

Types in the grammar A type is defined as follows:

$$\begin{aligned} \textit{non-array-type} &\rightarrow \texttt{INT} \\ &\rightarrow \texttt{LONG} \\ &\rightarrow \texttt{FLOAT} \\ &\rightarrow \texttt{CHAR} \\ &\rightarrow \texttt{STRING} \\ &\rightarrow \texttt{BOOLEAN} \\ &\rightarrow \texttt{VOID} \\ &\rightarrow \texttt{CLASS-NAME} \end{aligned}$$

$$\begin{aligned} \textit{type} &\rightarrow \textit{non-array-type} \\ &\rightarrow \textit{type}\,\texttt{[]} \end{aligned}$$

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. Classes in Boomslang may be "real" classes or "generic" classes. Generic classes must be instantiated into real classes before use.

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

$$\begin{aligned} \textit{class-name-list} &\rightarrow \texttt{CLASS-NAME} \\ &\rightarrow \textit{class-name-list}\ \texttt{,}\ \texttt{CLASS-NAME} \end{aligned}$$

classheader → CLASS `CLASS-NAME : NEWLINE`
→ CLASS `CLASS-NAME [`*class-name-list*`] : NEWLINE`

classdecl → *classheader*
`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. For brevity, not every classdecl is listed above. Each of the static, required, and optional lines is optional. For the full classdecl definition, see parser.mly

Objects can be instantiated using the following rule.

object-instantiation → `CLASS-NAME(`*params*`)`
→ `CLASS-NAME()`

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 → *expr*`.IDENTIFIER`
→ `CLASS-NAME.IDENTIFIER`

SELF is a keyword that allows the programmer to access an object's field from within the object itself. When accessing a variable on a class name as opposed to an expression or instance name, the variable must be a static variable.

Arrays Arrays are available as an aggregate data type. Arrays can consist of any type, e.g. primitives, objects, or other arrays. Boomslang allows for nested arrays, so something like `int[][] x` is 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*.

Default arrays can also be initialized using default array constructors.

array-default → *non-array-type*[*expr*]
→ *array-default*[*expr*]

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

array-access → *expr*[*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. Similarly, the expression being accessed like an array must be of type array.

3.2.3 Functions

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`.

$$
\begin{aligned}
\mathit{fdecl} &\rightarrow \texttt{DEF IDENTIFIER}(\mathit{type\text{-}params})\ \texttt{RETURNS}\ \mathit{type}:\ \texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{DEF IDENTIFIER}(\mathit{type\text{-}params}):\ \texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{DEF IDENTIFIER}()\ \texttt{RETURNS}\ \mathit{type}:\ \texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{DEF IDENTIFIER}():\texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{DEF OBJ-OPERATOR-METHOD-NAME}(\texttt{TYPE IDENTIFIER})\ \texttt{RETURNS}\ \mathit{type}:\texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{DEF OBJ-OPERATOR-METHOD-NAME}(\texttt{TYPE IDENTIFIER}):\texttt{NEWLINE INDENT}\ \mathit{stmts}\ \texttt{DEDENT}
\end{aligned}
$$

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

$$
\begin{aligned}
\mathit{fdecls} &\rightarrow \mathit{fdecl}\\
&\rightarrow \mathit{fdecls}\ \mathit{fdecl}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{optional\text{-}fdecls} &\rightarrow \mathit{fdecls}\\
&\rightarrow \epsilon
\end{aligned}
$$

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

$$
\begin{aligned}
\mathit{func\text{-}call} &\rightarrow \mathit{expr}.\texttt{IDENTIFIER}(\mathit{params})\\
&\rightarrow \texttt{IDENTIFIER}(\mathit{params})\\
&\rightarrow \mathit{expr}.\texttt{IDENTIFIER}()\\
&\rightarrow \texttt{IDENTIFIER}()\\
&\rightarrow \mathit{expr}\ \texttt{OBJ-OPERATOR}\ \mathit{expr}
\end{aligned}
$$

Note that the parentheses are *always* mandatory, even if the function takes no arguments. If a function is called as in *expr*.`IDENTIFIER`, it is expected that the *expr* is a class type.

`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.

3.2.4 Expressions

Literals A literal by itself can be considered an expression. `SELF` and `NULL` are also considered expressions.

Thus all of the following productions are valid expressions:

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

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:

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

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

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

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.

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

Mathematical operators

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

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 $-$.

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

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.

$$\mathit{assign} \rightarrow \mathit{type}\ \texttt{IDENTIFIER = }\ \mathit{expr}$$

Boomslang also includes the update and assign operators `+= -= *= /=`. 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.

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

Thus the following are valid expressions:

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

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

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

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.

3.2.5 High-level program structure

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:

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

3.2.6 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:

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

A group of statements can appear one after the other:

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

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:

$$\begin{aligned}
\textit{if-stmt} &\rightarrow \texttt{IF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{IF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT ELSE :}\ \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{IF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\ \textit{elif}\ \texttt{ELSE :}\ \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{IF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\ \textit{elif}
\end{aligned}$$

Where *elif* is defined as:

$$\begin{aligned}
\textit{elif} &\rightarrow \texttt{ELIF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \textit{elif}\ \texttt{ELIF}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}
\end{aligned}$$

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

$$\begin{aligned}
\textit{loop} &\rightarrow \texttt{LOOP}\ \textit{expr}\ \texttt{WHILE}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}\\
&\rightarrow \texttt{LOOP WHILE}\ \textit{expr} : \texttt{NEWLINE INDENT}\ \textit{stmts}\ \texttt{DEDENT}
\end{aligned}$$

3.2.7 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:

$$\begin{aligned}
\textit{type-params} &\rightarrow \textit{type}\ \texttt{IDENTIFIER}\\
&\rightarrow \textit{type-params}\ ,\ \textit{type}\ \texttt{IDENTIFIER}
\end{aligned}$$

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

$$\begin{aligned} \textit{params} &\rightarrow \textit{expr} \\ &\rightarrow \textit{params}\ \ ,\ \ \textit{expr} \end{aligned}$$

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

$$\textit{vdecl} \rightarrow \textit{type}\ \ \texttt{IDENTIFIER}$$

$$\begin{aligned} \textit{vdecls} &\rightarrow \textit{vdecl} \\ &\rightarrow \textit{vdecls}\ \texttt{NEWLINE}\ \textit{vdecl} \end{aligned}$$

3.2.8 Associativity and precedence table

The table is increasing order of precedence.

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

3.3 Semantics

3.3.1 Declaration rules

Variables must be defined before they are used, but functions and classes need not be. 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.

3.3.2 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.

3.3.3 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)

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

3.3.4 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
```

3.3.5 Mutability

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

3.3.6 Exception handling

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

3.4 Conversions

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

3.4.1 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.

3.4.2 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.

3.4.3 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'
```

4 Project Plan

Our team met once a week with our TA to ask questions and check in on progress. In addition to the weekly TA meetings, we would set up ad hoc meetings between ourselves to pair program and work on the language about twice a week. During these sections, we would also divide up work that people could complete independently. To help make sure there was no overlap, we used GitHub issues assigned to people to disambiguate who should work on what task. Additionally, we would tag each issue to a GitHub milestone (e.g. language reference manual, hello world, and final report) to make sure we were making incremental progress along the way.

The class structure, with the various milestones (project proposal, reference manual, hello world, and final report) made it easy to make incremental progress in a structured way. First we designed the language. Then we implemented the scanner and parser with an unambiguous grammar in order to write the LRM. Then for hello world, we had a nearly complete AST/SAST/semantic checker. Finally for the final report, we focused most of the effort on codegen.

Stack Our project stack consisted of the following tools:
- **Source code and project management:** GitHub
- **Documentation and Presentation:** Overleaf for LaTeX editing, Google Slides for presentation slides
- **Pair programming:** Visual Studio Code Live Share
- **Video conferencing:** Google Hangouts and Zoom
- **Chat:** Signal App

Style Each of us used a different development environment, so to remain in sync we used the provided Docker image, which was extremely helpful. For coding style, we did not impose many hard limits. When coding in OCaml, we settled on 2 spaces for indents, and using `snake_case` everywhere.

Roles Our roles were roughly as follows:
- Nathan Cuevas - Architect/Language Guru
- Robert Kim - Tester/Architect
- Nikhil Kovelamudi - Language Guru/Tester
- David Steiner - Project manager/Architect

All of us contributed in multiple ways. For instance, **everyone** wrote tests, although not everyone is listed as tester above.

5 Language Evolution

We are proud to say the core functionality of the language changed very little from start to finish. We were able to execute on our original vision of a language that combines some of the best features of Python and Java, meaning our final product did feature syntactically significant whitespace, object operator overloading, and auto-generated constructors. We even exceeded our original proposal in some ways, for instance by adding generic class macros.

The biggest change in our language's evolution was actually the name. Our team was originally named Python++, because we were building a Python-like language. However, this was confusing because we intentionally changed the loop and class syntax, as well as several other **intentional** deviations from Python. To make the name less confusing, we changed it from Python++ to Boomslang. "Boomslang" means "tree snake", so there is still a connection to Python, but the user is no longer misled into thinking the language is meant to be a carbon copy of Python.

6 Translator Architecture

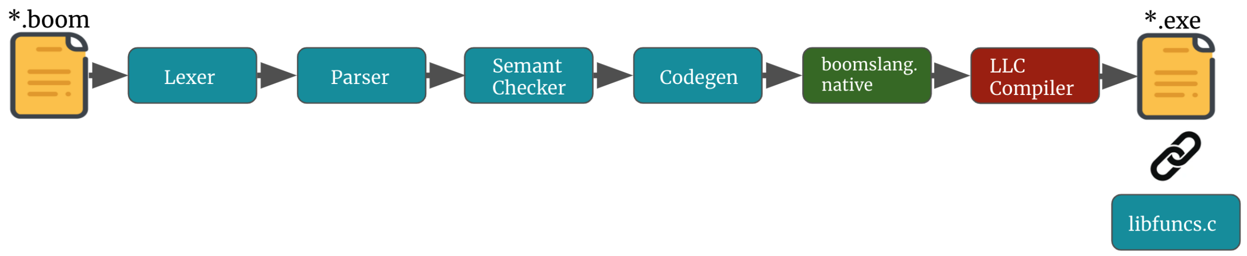
6.1 Translator overview

The translator for Boomslang can be broken down into four main components. The lexer, parser, semantic checker, and codegen. Nearly the entire translator is written in OCaml with the exception of a single .c file that is used to implement the library functions. The translator creates LLVM IR code, which is then compiled using the LLC compiler to generate an executable. Below is a visualisation of the how the various components of the translator interact.

6.2 The lexer

The lexer is implemented in an ocamllex file called scanner.mll. The lexer goes through the input .boom file and converts regular expressions in the source file into tokens for the parser. In addition to creating the tokens, the lexer removes comments and converts the syntactically significant white space into the appropriate INDENT and DEDENT tokens.

Figure 1: The Boomslang Translator Visualized



6.3 The parser

The parser is implemented in an ocamlyacc file called parser.mly. The parser takes the tokens generated from the lexer and converts them to an abstract syntax (AST) defined by Boomslang's grammar. The nodes of the AST are defined in ast.ml. In Boomslang, the code for generating the the graphiz visualization of the AST is implemented in ast.ml as well.

6.4 The semantic checker

The semantic checker is implemented in semant.ml and takes in the AST generated from the parser and performs a semantic check. Among many things, the semantic checker in Boomslang verifies the types of the nodes of the AST, does a search and replace in the AST for all the necessary generics, does error checking for everything that can be checked at compile time, handles the pseudo-polymorphism of the println and string concatenation and defines the function signatures for the built-in functions. The semantic checker outputs a syntactically checked abstract syntax tree or SAST, with nodes defined in sast.ml. Similar to the parser, the utility code for generating graphiz files for the SAST is implemented in this file.

6.5 Codegen

Codegen is implemented in codegen.ml and takes the SAST generated by the semantic checker and converts it into LLVM IR 3-address code using the LLVM module in OCaml. Codegen is responsible for walking through the SAST nodes and building the appropriate LLVM IR while also keeping track of namespaces, global variables and built-in functions. Codegen also handles checks that can't be verified at compile time such as divide by zero exceptions and null exceptions. Both arrays and classes in Boomslang are implemented using LLVM structs that are allocated on the heap; therefore, the code for generating and managing the low level components of these features are completely in codegen.

6.6 End-to-end program example

Consider the following program written in Boomslang:

```
int a = 5
if a * 2 >= 10:
        a += 2
```

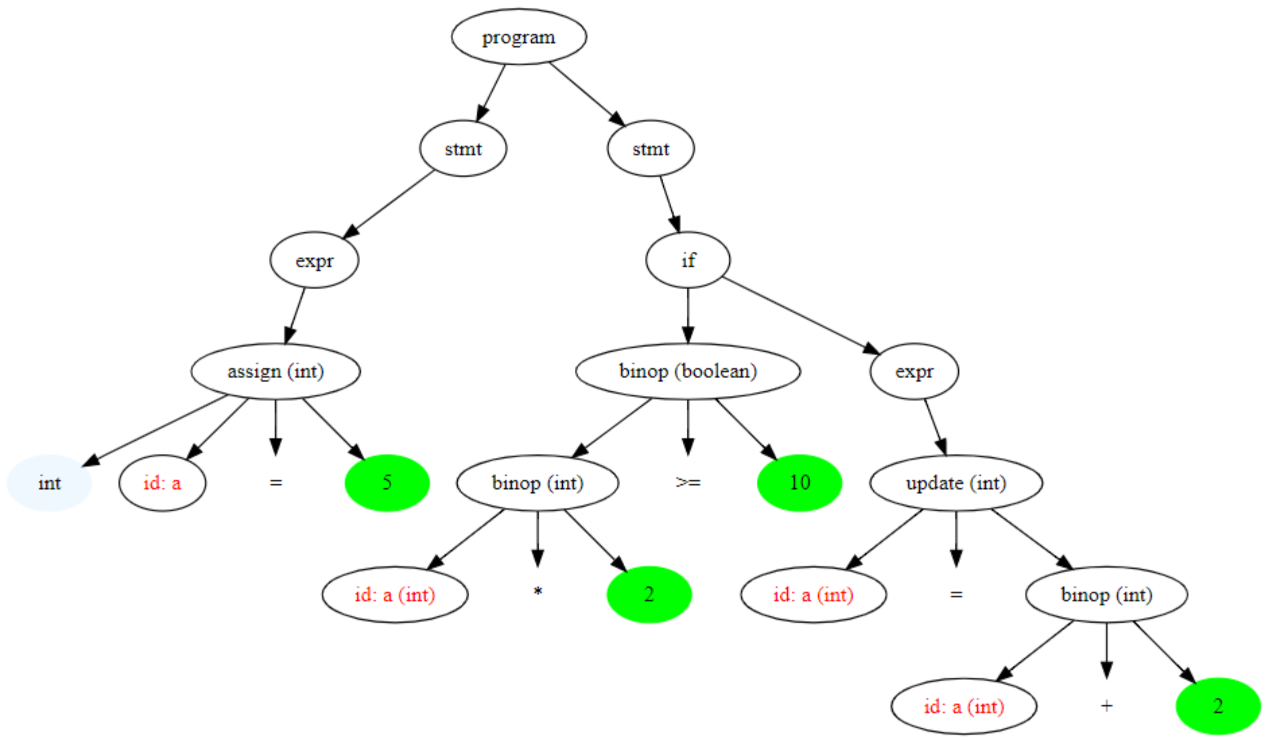
This will get tokenized as `INT IDENTIFIER(a) EQ INT-LITERAL(5) NEWLINE IF IDENTIFIER(a) TIMES INT-LITERAL(2) GTE INT-LITERAL(10) COLON NEWLINE INDENT IDENTIFIER(a) PLUS-EQ INT-LITERAL(2)`.

The tokens will be parsed by the parser and run through the semantic checker, yielding the semantically checked AST shown in 4.

After the SAST is built, it is converted by `codegen.ml` into the following LLVM code:

```
; ModuleID = 'Boomslang'
source_filename = "Boomslang"
```

Figure 2: The semantically checked AST for the program



```
@a = global i32 0

; NOTE: Omitting other built-ins here for brevity
declare void @println(i8*, ...)

define i32 @main(...) {
entry:
  store i32 5, i32* @a
  %a = load i32, i32* @a
  %tmp = mul i32 %a, 2
  %tmp1 = icmp sge i32 %tmp, 10
  br i1 %tmp1, label %then, label %merge

merge:                                            ; preds = %entry, %then
  ret i32 0

then:                                             ; preds = %entry
  %a2 = load i32, i32* @a
  %tmp3 = add i32 %a2, 2
  store i32 %tmp3, i32* @a
  br label %merge
}
```

7 Test Plan and Scripts

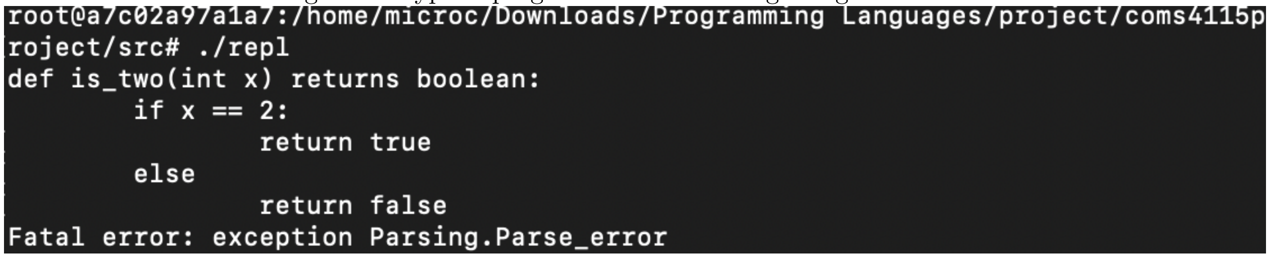
7.1 Overview

Throughout the entire project we followed a test-driven methodology where every feature that was implemented was accompanied by its associated tests covering both the happy and sad paths. We started out by creating a `run_tests.py` script that included unit tests for the lexer and the parser of our language. It helped us to ensure that as we built out our language, none of the previously developed features were broken. This script was using the REPL in the background to test the programs. Additionally, we utilized the REPL by itself to be able to test our code in an interactive environment, which was very useful throughout the entire project to be able to troubleshoot parsing issues of Boomslang programs, especially the long ones. Later on we took the MicroC test suite as a baseline, converted all the tests to our language syntax, added the tests from `run_tests.py` script, and added more tests that covered the rest of the Boomslang features.

7.2 Interactive environment for testing

The REPL has proved to be crucial in being able to test the lexer and the parser in an interactive environment. Since Boomslang uses syntactically significant whitespace, it was easy to make mistakes in the beginning by using spaces, or adding extra tabs on newlines. Such issues were easily caught using the REPL by copy and pasting the code into it and finding out which line was causing the issues. Figure 3 shows an example of a typical program we would run in the REPL to troubleshoot a parsing issue.

Figure 3: Typical program troubleshooting using the REPL



```
root@a7c02a97a1a7:/home/microc/Downloads/Programming Languages/project/coms4115p
roject/src# ./repl
def is_two(int x) returns boolean:
        if x == 2:
                return true
        else
                return false
Fatal error: exception Parsing.Parse_error
```

After running it in the REPL we can see that the program is missing a colon after the else statement.

7.3 Unit tests for lexer and parser

In the beginning of the project, before we even had the semantic checking and code generation, we wanted to be able to test the code we developed in the lexer and the parser to ensure that it not only worked as expected, but did not break as we added more features. In order to achieve this we created a `run_tests.py` script. The script was implemented by using the REPL in the backend and reading the output from the terminal.

Here is an example of a simple happy path test.

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

Here is an example of a simple sad path test.

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

For sad path tests we expect to see error messages that contain one of the phrases below

```
lexing: empty token
Stdlib.Parsing.Parse_error
Illegal
Fatal error
```

By the end of the lexer and parser implementation we had 1017 lines of test code.

7.3.1 run_tests.py

Since the `run_tests.py` is not part of the final repository anymore, I added the entire source file here to showcase the test coverage for all the features in the lexer and the parser.

```python
"""Unit tests for lexer and parser.
Usage: python3 -m unittest run_tests
"""
import os
import subprocess
import unittest

_PASSED = b"Passed\n"

class TestBoomslang(unittest.TestCase):

  def setUp(self):
    self.makeClean()

  def tearDown(self):
    self.makeClean()

  def makeClean(self):
    process = subprocess.Popen(["make", "clean"],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    _, _ = process.communicate()
    process.terminate()

  def make(self):
    process = subprocess.Popen(["make", "repl"],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    _, _ = process.communicate()
    process.terminate()

  def assertPassed(self, stdout, stderr):
    self.assertIn(_PASSED, stdout)
    self.assertNotIn(b"Stdlib.Parsing.Parse_error", stderr)

  def assertFailed(self, stdout, stderr):
    self.assertEqual(b"", stdout)
    self.assertTrue(b"lexing: empty token" in stderr or
                    b"Stdlib.Parsing.Parse_error" in stderr or
                    b"Illegal" in stderr or
                    b"Fatal error" in stderr)
  def assertProgram(self, program, passes=True):
    self.make()
    process = subprocess.Popen(["./repl"],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    stdout, stderr = process.communicate(input=program)
    process.terminate()
    if passes:
      self.assertPassed(stdout, stderr)
    else:
      self.assertFailed(stdout, stderr)

  def assertProgramPasses(self, program):
    self.assertProgram(program, passes=True)

  def assertProgramFails(self, program):
    self.assertProgram(program, passes=False)

  def test_grammar_is_not_ambiguous(self):
    process = subprocess.Popen(["ocamlyacc", "-v", "parser.mly"],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE)
    stdout, stderr = process.communicate()
    process.terminate()
    self.assertNotIn(b"shift/reduce conflicts", stdout)
    self.assertNotIn(b"shift/reduce conflicts", stderr)

  def test_objoperator_decl_1(self):
    program = b"""
class Horse:
        def _+#(Horse other) returns Horse:
                return Horse()
Horse yak = Horse()
Horse saddle = Horse()
Horse winne = yak+#saddle
"""
    self.assertProgramPasses(program)

  def test_objoperator_decl_2(self):
    program = b"""
class Horse:
        def _^&%$(Horse other) returns Horse:
                return Horse()
        def _$%(Horse other) returns Horse:
                return Horse()
Horse yak = Horse()
Horse saddle = Horse()
Horse poop = Horse()
Horse winne = yak ^&%$ saddle $% poop
"""
    self.assertProgramPasses(program)
  def test_objoperator_ambiguity_1(self):
    program = b"""
class Horse:
        def _+#(Horse other) returns int:
                return 5
Horse yak = Horse()
Horse saddle = Horse()
int winne = 8 + yak+#saddle + 6
"""
    self.assertProgramPasses(program)

  def test_empty_program_passes(self):
    program = b""
    self.assertProgramPasses(program)

  def test_simple_assignment_passes_1(self):
    program = b"int x = 5 \n"
    self.assertProgramPasses(program)

  def test_simple_assignment_passes_2(self):
    program = b"float yEs = .5 \n"
    self.assertProgramPasses(program)

  def test_simple_assignment_passes_3(self):
    program = b'''
def myfunc(int x) returns string:
        return "hey"
string foo = myfunc(1+1+2+3+5)
'''
    self.assertProgramPasses(program)

  def test_simple_assignment_passes_4(self):
    program = b"float yEs = -.5 \n"
    self.assertProgramPasses(program)

  def test_simple_assignment_passes_5(self):
    program = b"float yEs = -2.5 \n"
    self.assertProgramPasses(program)

  def test_assignment_without_type_passes(self):
    program = b"""
int x = 0
x = 5
"""
    self.assertProgramPasses(program)

  def test_object_assignment_passes(self):
    program = b"""
class MyObject:
        required:
                int x
                string y
def myfunc(int x) returns string:
        return "hey"
```

```
MyObject foo = MyObject(2, myfunc(2+2))
"""
    self.assertProgramPasses(program)

  def test_arithmetic_passes(self):
    program = b"2 + 3 * 5 / 4\n"
    self.assertProgramPasses(program)

  def test_array_declaration_passes_1(self):
    program = b"int[6] array = [1, 2, 3, 4, 5, 6]\n"
    self.assertProgramPasses(program)

  def test_array_declaration_passes_2(self):
    program = b"[]\n"
    self.assertProgramPasses(program)

  def test_array_declaration_passes_3(self):
    program = b"[[1, 2, 3]]\n"
    self.assertProgramPasses(program)

  def test_array_access_passes(self):
    program = b"""
int[3] array = [0,1,2]
int x = array[2]
"""
    self.assertProgramPasses(program)

  def test_null_assignment(self):
    program = b'''
class MyObject:
        def myfunc():
                println("hey")
MyObject x = NULL
'''
    self.assertProgramPasses(program)

  def test_single_comments_1(self):
    program = b"#comments\n"
    self.assertProgramPasses(program)

  def test_single_comments_2(self):
    program = b"int x = 2 \n int foo = 2+2 # comment\n\n\n int y = 2\n"
    self.assertProgramPasses(program)

  def test_single_comments_3(self):
    program = b"int x = 2 \n int foo = 2+2 # comment\n int y = 2\n"
    self.assertProgramPasses(program)

  def test_multi_comments(self):
    program = b"/#comments\ncomments#/\n"

  def test_double_eq(self):
    program = b"int x = 2\n3 == x\n"
    self.assertProgramPasses(program)
  def test_newlines_1(self):
    program = b"\n"
    self.assertProgramPasses(program)

  def test_newlines_2(self):
    program = b"\n\n\n\n\n"
    self.assertProgramPasses(program)

  def test_newlines_3(self):
    program = b"\n\n\n\n int x = 5 \n\n int y = 6 \n\n"
    self.assertProgramPasses(program)

  def test_newlines_4(self):
    program = b"\n\n\n\n int x = 5\n"
    self.assertProgramPasses(program)

  def test_newlines_5(self):
    program = b"int x = 5 \n\n\n\n"
    self.assertProgramPasses(program)

  def test_self_access_with_field(self):
    program = b"""
class Foo:
        optional:
                string x = "str"
        def myfunc():
                self.x
"""
    self.assertProgramPasses(program)

  def test_self_assignment_with_field(self):
    program = b"""
class MyClass:
        required:
                int x
        def myfunc():
                self.x = 5
"""
    self.assertProgramPasses(program)

  def test_self_with_expression(self):
    program = b'''
class MyClass:
        required:
                int foo
        def myfunc():
                int x = self.foo * 5
'''
    self.assertProgramPasses(program)

  def test_self_access_with_function(self):
    program = b"""
class MyClass:
        def myfunction(int x):
                println("hey")
        def myfunction2():
                self.myfunction(2 % 3)
"""
    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_object_variable_assignment(self):
    program = b"""
class MyObject:
        required:
                boolean f0Oo12345
MyObject myobject = MyObject(false)
myobject.f0Oo12345 = true
"""
    self.assertProgramPasses(program)

  def test_object_variable_with_expression(self):
    program = b'''
class MyObject:
        static:
                boolean is_this_true = true
MyObject myobject = MyObject()
boolean x = true and myobject.is_this_true
'''
    self.assertProgramPasses(program)

  def test_object_function_call(self):
    program = b'''
class MyObject:
        def myfunction(int a, string b, boolean c):
                println("hey")
int a = 5
string b = "foo"
boolean c = false
MyObject myobject = MyObject()
myobject.myfunction(a, b, c)
'''
    self.assertProgramPasses(program)

  def test_long_initialization_passes(self):
    program = b"long lo = 500000000000L\n"
    self.assertProgramPasses(program)
  def test_valid_minus_eq(self):
    program = b"long fOo = 10L\nfOo -= 500L\n"
    self.assertProgramPasses(program)

  def test_valid_divide_eq_with_object(self):
    program = b'''
class MyObject:
        required:
                int heY
MyObject myobject = MyObject(5)
myobject.heY /= 20
'''
    self.assertProgramPasses(program)

  def test_valid_char_literal_1(self):
    program = b"char c = 'a'\n"
    self.assertProgramPasses(program)

  def test_valid_char_literal_2(self):
    program = b"char c = '''\n"
    self.assertProgramPasses(program)

  def test_valid_string_literal(self):
    program = b'string foo = "foooooOOOoo !@$%^&*()_-+={}|[]:;/<,.>"\n'
    self.assertProgramPasses(program)

  def test_complicated_program_1(self):
    program = b"""
int x = 5
def myfunc(int x, MyObject foo) returns string:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                elif x > 20:
                        println("hey")
                elif x > 30:
                        println("hey")
                        println("hey")
                else:
                        println("hey")
        return "hey"
int x = 50
"""
    self.assertProgramPasses(program)

  def test_complicated_program_2(self):
    program = b"""
int x = 5
def myfunc(int x, MyObject foo) returns string:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                        return "hey"
                elif x > 20:
                        println("hey")
                        return "hey"
                elif x > 30:
                        println("hey")
                        println("hey")
                        return "hey"
                else:
                        println("hey")
                        return "hey"
        else:
                return "hey"
def myfunc2(int x, MyObject foo) returns string:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                elif x > 20:
                        println("hey")
                elif x > 30:
                        println("hey")
                        println("hey")
                else:
                        println("hey")
        return "hey"
int x = 50
"""
    self.assertProgramPasses(program)

  def test_complicated_program_3(self):
    program = b"""
int x = 5
def myfunc(int x, MyObject foo) returns void:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                elif x > 20:
                        println("hey")
                elif x > 30:
                        println("hey")
                        println("hey")
                else:
                        println("hey")
"""
    self.assertProgramPasses(program)

  def test_class_declaration_1(self):
    program = b"""
```

```
class MyObject:
        static:
                int x = 5
                string foo = "bar"
        required:
                int z
                float f0000
        optional:
                boolean boo = true
"""
    self.assertProgramPasses(program)

  def test_class_declaration_2(self):
    program = b"""
class MyObject:
        static:
                int x = 5
                string foo = "bar"
        required:
                int z
                float f0000
        optional:
                boolean boo = true
"""
    self.assertProgramPasses(program)

  def test_class_declaration_3(self):
    program = b"""
class MyObject:
        static:
                int x = 5
                string foo = "bar"
        required:
                int z
                float f0000
        optional:
                boolean boo = true
        def mymethod():
                self.x = 5
        def _++%%(MyObject other) returns MyObject:
                return MyObject(1, 5.0)
"""
    self.assertProgramPasses(program)

  def test_class_declaration_4(self):
    program = b"""
class MyObject:
        static:
                int x = 5
                string foo = "bar"
        required:
                int z
                float f0000
        optional:
                boolean boo = true
        def mymethod():
                self.x = 10
        def _++%%(MyObject other) returns MyObject:
                return MyObject(1, 5.0, false)
        def mymethod2(int x) returns int:
                return 5
"""
    self.assertProgramPasses(program)

  def test_class_declaration_5(self):
    program = b"""
class MyObject:
        def _++%%(MyObject other) returns MyObject:
                return MyObject()
        def mymethod2(int x) returns int:
                return 5
"""
    self.assertProgramPasses(program)

  def test_class_declaration_6(self):
    program = b"""
class MyObject:
        static:
                int x = 5
"""
    self.assertProgramPasses(program)

  def test_class_declaration_7(self):
    program = b"""
class MyObject:
        required:
                int x
"""
    self.assertProgramPasses(program)

  def test_class_declaration_8(self):
    program = b"""
class MyObject:
        optional:
                int x = 5
"""
    self.assertProgramPasses(program)

  def test_class_declaration_9(self):
    program = b"""
class MyObject:
        static:
                int x = 5
        required:
                int y
"""
    self.assertProgramPasses(program)
  def test_class_declaration_10(self):
    program = b"""
class MyObject:
        static:
                int x = 5
        optional:
                int y = 5
"""
    self.assertProgramPasses(program)

  def test_class_declaration_11(self):
    program = b"""
class MyObjectTwo:
        required:
                string x
        optional:
                string z = "foo"
"""
    self.assertProgramPasses(program)

  def test_loop_1(self):
    program = b"""
int x = 0
loop x+=1 while x<100:
        println("hey")
"""
    self.assertProgramPasses(program)

  def test_loop_2(self):
    program = b"""
int x = 0
loop while x<100:
        println("hey")
"""
    self.assertProgramPasses(program)

  def test_valid_statement_without_newline_succeeds_1(self):
    program = b"2 + 3 * 5 / 4"
    self.assertProgramPasses(program)

  def test_valid_statement_without_newline_succeeds_2(self):
    program = b'println("hello world")'
    self.assertProgramPasses(program)

  def test_valid_returns_succeeds(self):
    program = b'''
def my_func(int x) returns int:
        if x > 5:
                println("hey")
        else:
                return 5
        return 1
'''
    self.assertProgramPasses(program)
  def test_valid_construct_succeeds(self):
    program = b'''
class MyClass:
        def print_5():
                println("5")
MyClass my_object = MyClass()
MyClass x = my_object
x.print_5()
'''
    self.assertProgramPasses(program)

  def test_valid_array_succeeds(self):
    program = b'''
def my_func(int x) returns int:
        return x
int[5] my_array = [1, (2+2), my_func(1), my_func(1) + 1, 2 * (3 + 1)]
'''
    self.assertProgramPasses(program)

  def test_valid_not_succeeds(self):
    program = b'''
boolean x = true
not x
'''
    self.assertProgramPasses(program)

  def test_valid_neg_succeeds(self):
    program = b'''
int x = -1
x = -x
'''
    self.assertProgramPasses(program)

  def test_valid_self_return_succeeds(self):
    program = b'''
class MyClass:
        def foo() returns MyClass:
                return self
'''
    self.assertProgramPasses(program)

  def test_constructor_overwriting_is_allowed(self):
    program = b'''
class MyClass:
        required:
                int x
        def construct(int x):
                println("i am overwriting the default constructor")
'''
    self.assertProgramPasses(program)

  def test_assigning_to_array_index_succeeds(self):
    program = b'''
```

```
string[3] arr = ["one", "two", "three"]
arr[1] = "newvalue"
arr[1]
'''
    self.assertProgramPasses(program)

  def test_array_update_succeeds(self):
    program = b'''
int[2] arr = [1, 2]
arr[0] += 5
'''
    self.assertProgramPasses(program)

  def test_nonsense_fails(self):
    program = b"%-$_? !?\n"
    self.assertProgramFails(program)

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

  def test_assert_invalid_function_call_fails(self):
    program = b"myfunc(1,2,) \n"
    self.assertProgramFails(program)

  def test_invalid_array_literal_fails_1(self):
    program = b"[1, 2, 3,]\n"
    self.assertProgramFails(program)

  def test_invalid_array_literal_fails_2(self):
    program = b"[[1, 2, 3]\n"
    self.assertProgramFails(program)

  def test_invalid_arithmetic_fails(self):
    program = b"5 *\n"
    self.assertProgramFails(program)

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

  def test_invalid_single_comment(self):
    program = b"/ comment\n"
    self.assertProgramFails(program)

  def test_invalid_multi_comment(self):
    program = b"comment #/\n"
    self.assertProgramFails(program)

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

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

  def test_invalid_self_usage(self):
    program = b"self.\n"
    self.assertProgramFails(program)

  def test_invalid_object_usage(self):
    program = b"myobject.\n"
    self.assertProgramFails(program)

  def test_invalid_primitive_type_fails(self):
    program = b"short x = 500\n"
    self.assertProgramFails(program)

  def test_invalid_long_assignment_fails(self):
    program = b"long f0oo = 5000LL\n"
    self.assertProgramFails(program)

  def test_invalid_floating_point_fails(self):
    program = b"float v = 1.\n"
    self.assertProgramFails(program)

  def test_invalid_plus_eq(self):
    program = b"int x += 5\n"
    self.assertProgramFails(program)

  def test_invalid_times_eq(self):
    program = b"int x *= 6\n"
    self.assertProgramFails(program)

  def test_simple_objop(self):
    program = b"Wfoueb $^@#&@ Wefoudvn\n"
    self.assertProgramFails(program)

  def test_multi_objop(self):
    program = b"Wfoueb $^@#&@ Wefoudvn %&@@$^ HdfEFow\n"
    self.assertProgramFails(program)

  def test_objoperator_nofirsteq_1(self):
    program = b"woof = woofwoof =^&$# harry\n"
    self.assertProgramFails(program)

  def test_objoperator_nofirsteq_2(self):
    program = b"woof = woofwoof ?= harry\n"

  def test_invalid_char_literal_fails(self):
    program = u"char c = ''".encode('utf-16')
    self.assertProgramFails(program)

  def test_invalid_string_literal_fails(self):
    program = u'string x = "f0000  foo"\n'.encode('utf-16')
    self.assertProgramFails(program)
  def test_bad_indentation_fails_1(self):
    program = b"""
int x = 5
def myfunc(int x, MyObject foo) returns string:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                elif x > 20:
                        println("hey")
                elif x > 30:
                        println("hey")
                        println("hey")
                else:
                                println("hey")
int x = 50
"""
    self.assertProgramFails(program)

  def test_bad_indentation_fails_2(self):
    program = b"""
int x = 5
def myfunc(int x, MyObject foo) returns string:
        int y = 5
        int z = 7
        if x > 5:
                if x > 10:
                        int z = 20
                elif x > 20:
                        println("hey")
                elif x > 30:
                        println("hey")
                        println("hey")
                else:
                println("hey")
int x = 50
"""
    self.assertProgramFails(program)

  def test_bad_loop_fails(self):
    program = b"""
loop x+2 while:
        2+2
"""
    self.assertProgramFails(program)

  def test_bad_returns_1(self):
    program = b'''
def my_func() returns int:
        if x > 5:
                println("hey")
        else:
                return 5
```

```
'''
    self.assertProgramFails(program)

  def test_bad_returns_2(self):
    program = b'''
def my_func() returns int:
        println("hey")
'''
    self.assertProgramFails(program)

  def test_bad_constructor_1(self):
    program = b'''
class MyClass:
        static:
                my_object.x = 5
'''
    self.assertProgramFails(program)

  def test_bad_constructor_2(self):
    program = b'''
class MyClass:
        static:
                void x = NULL
'''
    self.assertProgramFails(program)

  def test_void_assignment_fails(self):
    program = b"""void x = 5"""
    self.assertProgramFails(program)

  def test_null_assignment_fails(self):
    program = b"""NULL x = NULL"""
    self.assertProgramFails(program)

  def test_duplicate_function_args_fails(self):
    program = b"""
def my_func(int x, float x) -> returns int:
        return 5
"""
    self.assertProgramFails(program)

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

  def test_bad_negation_fails(self):
    program = b"""
boolean x = true
-x
"""
    self.assertProgramFails(program)
  def test_bad_not_fails(self):
    program = b"""
int x = 5
not x
"""
    self.assertProgramFails(program)

  def test_bad_array_init_fails(self):
    program = b"""
def my_func(int x) returns string:
        return "hey"
int[5] my_array = [1, (2+2), my_func(1), my_func(1) + 1, 2 * (3 + 1)]
"""
    self.assertProgramFails(program)

  def test_invalid_type_return_fails_1(self):
    program = b"""
def my_func() returns int:
        if x > 5:
                return
        else:
                return 5
"""
    self.assertProgramFails(program)

  def test_invalid_type_return_fails_2(self):
    program = b"""
def my_func() returns int:
        if x > 5:
                return "string"
        else:
                return 5
"""
    self.assertProgramFails(program)

  def test_nothing_comes_after_return(self):
    program = b"""
def myfunc():
  return
  println("hey")
"""
    self.assertProgramFails(program)

  def test_no_matching_signature_fails(self):
    program = b"""
def myfunc(string x) returns int:
  return 5
myfunc(5)
"""
    self.assertProgramFails(program)

  def test_invalid_self_return_fails(self):
    program = b"""
```

```
class MyClass:
        def foo() returns int:
                return self
"""

  def test_self_on_its_own_fails(self):
    program = b"""self\n"""
    self.assertProgramFails(program)

  def test_self_as_identifier_fails(self):
    program = b"""
int self = 5
self
"""
    self.assertProgramFails(program)

  def test_constructors_must_not_have_returns(self):
    program = b"""
class MyClass:
        def construct() returns int:
                return 5
"""
    self.assertProgramFails(program)

  def test_bad_array_assign_fails(self):
    program = b"""
string[3] arr = ["one", "two", "three"]
arr[1] = 50.0
"""
    self.assertProgramFails(program)


if __name__ == '__main__':
    unittest.main()
```

7.4 Test Suite

The final test suite was implemented on the foundation of the MicroC's test suite with the addition of the tests from `run_tests.py` script, runtime compile error tests, and additional tests specifically related to our language. Our language has a lot of features such as single and multidimensional arrays, classes, recursive functions/classes, nested/multi-statement if statements and loops, which means that we had to significantly increase the number of happy and sad path tests to achieve close to 100% test coverage for our language. This resulted in 2218 lines of code across 251 tests. Each test was accompanied with a .out or a .err file for happy path and sad path tests respectively. These files were used to compare the output from each test file. All the tests were run using the testall.sh script which we borrowed from MicroC and tweaked to work with our language.

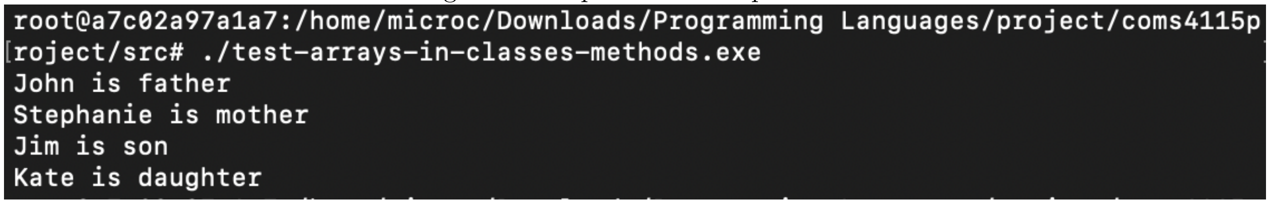
The tests that we wrote ranged from very simple integer assignment to multidimensional arrays in classes and operator overloading with classes. The test suite section has the complete listing of every Boomslang test.

Here is sneak peak into one of the tests from the test suite.

```
class Family:
        optional:
                string[][] relation = default string[4][2]
        def updateFathersName(string newName):
                self.relation[0][0] = newName

Family doeFamily = Family([["Typo","Stephanie","Jim","Kate"], ["father","mother","son","daughter"]])
doeFamily.updateFathersName("John")
int i = 0
loop i += 1 while i < 4:
        println(doeFamily.relation[0][i] + " is " + doeFamily.relation[1][i])
```

Figure 4: Output of the sample test



```
root@a7c02a97a1a7:/home/microc/Downloads/Programming Languages/project/coms4115p
roject/src# ./test-arrays-in-classes-methods.exe
John is father
Stephanie is mother
Jim is son
Kate is daughter
```

8 Conclusions

8.1 Who did what

8.1.1 Nathan Cuevas

I touched most parts of the translator's code over the course of the project, mostly by setting up boilerplate, scaleable code. My work started with pair programming with the rest of the team on the parser. I then wrote a novel implementation of syntactically significant whitespace code in the scanner, which was later revised. My largest contribution was codegen, where I led the effort in getting the hello world deliverable to work, then later set up the boilerplate code for codegen for the rest of the team to build on. I started by supporting built-in functions in codegen, then later I wrote all of the code for the function builder. I then wrote all of the code for the array implementation in codegen. During this time I wrote about a dozen tests and made modifications to the semantic checker, scanner, and parser as needed. I also wrote the architecture portion of the final report and contributed to the final presentation slides.

8.1.2 Robert Kim

In the beginning of the project, I implemented a few little features in the lexer and the parser such as NULL, double equals binary operator, multi-line comments and tests associated with those features. In addition, we had two three hour pair programming sessions where we worked on the foundation for the scanner, parser and the AST code, which I actively contributed to. Later on, I worked closely with Nathan to develop the foundation for the semant and codegen code that could print the helloworld program. We spent about 12 hours brainstorming and implementing it. At the end of the project I was in charge of developing the test suite for our language. I took the test suite from MicroC as the baseline. I tweaked the makefile, testall.sh script and changed the syntax for all the tests from MicroC (154 files) and made sure that they all passed, which involved troubleshooting parsing issues and fixing the syntax. Once the baseline was done, I continued by contributing to the codegen implementation. I attempted to write the if loops and the divide by zero exception features in codegen, but unfortunately even with the help of the TA and countless hours of researching, I wasn't able to get it fully working. David helped me a lot to understand the proper way of implementing both the if loops and the divide by zero exception, so now I feel like I have a greater understanding of recursive functions and how things work in codegen overall. Moreover, I worked closely with Nathan and helped him by implementing the tests for the arrays in classes. Lastly, I worked closely with Nikhil to create the presentation. We added the skeleton with all the appropriate titles, related code and formatting. For the LRM, I helped create the test section outlining various test tools we used and developed throughout the semester and the significance of proper testing for our language implementation.

8.1.3 Nikhil Kovelamudi

At the very start of the project, I designed the overall syntax ideas of the language after consultation with everyone else in our group and wrote most of the initial project proposal. The loop-while loop, object binary operators, and several other ideas were added by me into the LRM. I was tasked with specifying the behavior of our language in the LRM while writing the document together as a group. When it came time to create the lexer, parser, and AST, we had 2 long pair-programming session that I attended to help write out the initial submission. I worked primarily on the parser and specifying missing language rules at this beginning stage. Later I was tasked with working on codegen to implement numeric and boolean arithmetic to work as expected as specified in our language. I also worked on implementing escape characters in our parser. Towards the end of the project, I worked alongside Robert to write tests. I wrote several shell scripts including one that moved all tests from our old `run_tests.py` testing suite to our output file testing system, albeit with a lot of cleanup required. These tests encompassed loops, nested class structures, recursion, and object operator behavior. I wrote the binary tree library function as a demonstration of the working recursive features. I created the presentation document alongside Robert with appropriate information from our LRM and test/stdlib examples. I wrote scripts to output all our source files and test files to an easily copy-able and formatted document that could be pasted at the bottom of this report.

8.1.4 David Steiner

For coding, I contributed over 7,000 lines of code and test. I wrote the majority of the scanner, AST, SAST, and semantic checker. (The parser was written as a joint effort in a pair programming session.) I also contributed several key features in codegen, such as assisting in getting all primitives working, binary operators, unary operators, assisted with if statements, wrote loops, and wrote all the class code. I contributed most of our built-in functions in C, and added runtime checks for null pointer exceptions and divide by zero errors. I also wrote pretty printers for the AST and SAST (our language can print graphviz files for both), wrote standard library modules, and wrote dozens of tests. In total, I wrote at least 60% of all the code and tests in Boomslang.

For documentation, I wrote the majority of the pages in the project proposal LaTeX and the majority of the pages in the language reference manual. For the final report, I wrote most of the tutorial, reference manual, project plan, and language evolution.

8.2 Lessons learned

8.2.1 Nathan Cuevas

Out all the university courses that I have taken, Programming Languages and Translators with Professor Edwards is one of my favorites. The content of this class is the perfect mix of balance and practicality. Lex and Yacc is something that I have worked with a little bit as a software developer but didn't get fluent with until taking this course. LLVM is also a gold standard in compiler software and being exposed to LLVM IR was a unique experience. Functional programming is a paradigm that is getting more popular and using OCaml definitely opened my eyes to other styles of programming and showing me how smart a compiler can really be. There is a also a good amount of theory with the discussions of parsing algorithms and lambda calculus. Lambda calculus especially blew my mind because because it is a very mathematical definition of what is computable, more so than the Turing machine.

I also learned a lot in the project. One of the highlights was using functional programming and made me appreciate how important it is to get bugs when trying to compile the program rather than to get bugs when the program is already compiled. While coding in OCaml takes longer for me than a standard impreretive language like Python, I notice that I spend less time on weird dependency bugs and race conditions are almost out of the question! Writing the Boomslang compiler made me appreciate the intricacies of compiler design and how important the construct of recursion really is. I learned the hard way that it is impossible to do certain things at compile time (like handling dynamically resizing arrays and checking divide by zero). Learning to read/write LLVM IR also made me much better at working with 3-address code which I had next to no experience to coming into this class. I also learned how to use git/github to its fullest potential by working in a team environment.

8.2.2 Robert Kim

I would be lying if I said that I wasn't scared to take this course in the beginning of the semester. As my first semester being a CS major (previously a EE), I thought that there was no way I could design my own programming language. I am happy to say that after many hours of hard work, rewatching the lectures, and reading the ocaml API for LLVM, I know how programming languages with static typing similar to C, object oriented style similar to Java and significant whitespace similar to Python, are designed and implemented.

I now have a good understanding of how a string of text is converted into a program that can be executed using the lexer, parser, semantic checking and code generation, which I am quite proud of. I learned how important it is to develop good tests that cover both happy and sad paths, to ensure that existent features don't break throughout the development cycle. I also learned the importance of writing tests that combine multiple features to catch bugs that are hard to find when using features independently.

Lastly, being able to work with my teammates and pair program allowed me to learn more than I could have without them. I had a pretty good understanding about the way the parser and the lexer worked, but codegen was one thing that I struggled to understand at first. Once I pair programmed with my teammates and spend some time working on the if loops I had a better understanding of how to take the semantically checked abstract syntax tree and generate the LLVM IR code. I understood the power of recursion in codegen for nested and multistatement if loops and the purpose of basic blocks and how the pointer moves from one basic block to another and eventually ends with the merge basic block for each scope.

8.2.3 Nikhil Kovelamudi

Programming Languages and Translators was a very interesting course for me to take since the inner workings of compilers was a complete mystery to me prior to taking the course. Compiler architectures that can convert a readable language into three-address code assembly seemed like an overwhelmingly complicated task to accomplish. Now having taken this course and re-watching certain lectures countless times, I now understand and can better appreciate the intricacies of implementing what I previously thought of as minor changes in object-oriented programming languages. I also have a newfound appreciation for ocaml and how small programs of high complexity can get with the language. The differences between how languages handle whitespace, class inheritance, methods/functions, function scope, type inferencing and conversion are far more apparent and I can visualize now after reading online how python's interpreter and the C++ compiler work to implement difficult features.

The steps a compiler takes to interpret a language, from lexing/parsing to generating LLVM IR code from the abstract syntax tree is more intuitive to me now. However, working together with my teammates on this project over the past semester made me realize how critical in-person meetings/coding sessions are when it comes to understanding how the compiler needs to be laid out physically in code as opposed to simply understanding the concepts laid out in class. I started this class behind on ocaml programming skill and needed to catch up fast to contribute to the project. I really appreciated this class's focus on looking at example code after understanding concepts to prevent any strange misunderstandings about ocaml implementations that may rise up. Writing many, many test cases for our language also showed me how easy it is for languages to have weird edge-cases. Having to interact with a large variety of files has strangely given me a lot of scripting experience from moving things around and manipulating file contents in our git repository. Additionally, ocaml did make me struggle quite a bit in this class, but in the end I can say that I really like how satisfying it is to write neat looking code that does very complicated things. I completely understand why it was chosen as our primary compiler language.

8.2.4 David Steiner

I learned a tremendous amount about what makes a good programming language through the development of Boomslang. The epiphany I had is there seems to be a surprisingly harmonious relationship between what is elegant for the compiler writer and what is elegant for the programmer. This leads to "purity" in languages that require everything to be an expression, or everything to be a list, for instance. Things that are unwieldy for the programmer, like `NULL`, are also a nightmare for the compiler writer. `NULL` seems simple at first but then you have to address not just null pointer exceptions but what happens if a user calls a function with `NULL` as an argument, what is `NULL` is used in an array literal, etc. Optional/Maybe types are much better.

In compilers, the abstract syntax tree really is critical. When you get it right, features seem to flow naturally in unexpectedly powerful ways. When you get it wrong, adding a feature is like pulling teeth. The class also gave me a great amount of appreciation for modern production language compilers. Debugging our code with a compiler that doesn't even tell you which line the error occurred on was a nightmare! The debuggers and optimizers available for languages like OCaml, Java, etc. are really impressive and non-trivial, I now realize.

As advice to future takers of the course, I would say follow the advice of the professor and TA's. Focus on a language that does a few good things well and is composable rather than chasing every last tiny feature (e.g. adding "short" when you already have "int"). Work hard all throughout the semester as trying to finish the project in one week won't be possible. Compilers are very difficult and the feature you think is "simple" may end up taking many more hours than you expected.

9 Full Source Listing

9.1 Makefile

```
# "make all" builds the executable

.PHONY : all
all : boomslang.native

# "make boomslang.native" compiles the compiler

boomslang.native :
	opam config exec -- \
	ocamlbuild -use-ocamlfind -pkgs str boomslang.native

repl : parser.cmo scanner.cmo ast.cmo sast.cmo semant.cmo repl.cmo
	ocamlc str.cma -w +a-4 -o repl $^

%.cmo : %.ml
	ocamlc -w +a-4 -c $<

%.cmi : %.mli
	ocamlc -w +a-4 -c $<

%.cmi : %.ml
	ocamlc -w +a-4 -c $<

scanner.ml : scanner.mll
	ocamllex $^

parser.ml parser.mli : parser.mly
	ocamlyacc $^

# Depedencies from ocamldep
repl.cmo : scanner.cmo parser.cmi ast.cmi sast.cmi semant.cmi
repl.cmx : scanner.cmx parser.cmx ast.cmi sast.cmi semant.cmi
```

```
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmi parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx

# "make clean" removes all generated files

.PHONY : clean
clean :
        ocamlbuild -clean
        rm -rf ocamlllvm *.diff
        rm -rf \
        *.cmi *.cmo parser.ml parser.mli parser.output scanner.ml \
        repl.out repl *.out __pycache__ _build boomslang.native \
        *.ll *.exe *.s *.o testall.log

TARFILES = README.md Makefile \
        scanner.mll parser.mly ast.ml sast.ml semant.ml \
        codegen.ml boomslang.ml repl.ml repl.tb run_tests.py \
        _tags boom

zip :
        zip boomslang $(TARFILES)

test :
        ../test/testall.sh
```

9.2 ast.ml

```
type primitive = Int | Long | Float | Char | String | Bool | Void

type binop = Plus | Subtract | Times | Divide | Modulo | DoubleEq | BoGT | BoLT | BoGTE | BoLTE | BoOr

type unaryop = Not | Neg

type updateop = Eq

type typ =
  Primitive of primitive
| Class of string
| Array of typ
| NullType

type bind = typ * string

type expr =
  IntLiteral of int
| LongLiteral of int64
| FloatLiteral of string
| CharLiteral of char
| StringLiteral of string
| BoolLiteral of bool
| Id of string
| Self
| NullExpr
| Call of call
| ObjectInstantiation of string * expr list
| ObjectVariableAccess of object_variable_access
| ArrayAccess of array_access
| ArrayLiteral of expr list
(* e.g. int[2+2][5][6] would be parsed as
   (Array(int), [2+2]) then
   (Array(Array(int)), [5, 2+2]) then
   (Array(Array(Array(int))), [6, 5, 2+2])
   Telling us this is a default triple array that is an array of length 6
   containing arrays of length 5 containing arrays of length 4. *)
| DefaultArray of typ * expr list
| Binop of expr * binop * expr
| Unop of unaryop * expr
| Assign of assign
| Update of update
and array_access = expr * expr (* First expr must be an array, second must be int *)
and call =
  FuncCall of string * expr list (* my_func(1, 2, 3) *)
| MethodCall of expr * string * expr list (* expr.identifier(params) *)
and assign =
  RegularAssign of typ * string * expr
and update =
  RegularUpdate of string * updateop * expr
| ObjectVariableUpdate of object_variable_access * updateop * expr
| ArrayAccessUpdate of array_access * updateop * expr
and object_variable_access = {
  ova_expr: expr;
  ova_class_name: string;
  ova_var_name: string;
  ova_is_static: bool;
}

type stmt =
  Expr of expr
| Return of expr
| ReturnVoid
| If of expr * stmt list * elif list * stmt list
| Loop of expr * expr * stmt list
and elif = expr * stmt list

type fdecl = {
  rtype: typ;
  fname: string;
  formals: bind list;
  body: stmt list;
}

type classdecl = {
  (* The name of the class being defined. *)
  cname: string;
  (* This is only used for the case of instantiating a generic class.
     e.g. in class StringToIntMap = HashMap(string, int), this will
     be set to HashMap, as this is a real valued instantiation of
     the generic HashMap. *)
  source_class_name: string;
  (* If this is the generic class definition, these are the names
     of the generic types. e.g. in class HashMap[K, V]: the generics
     are [K, V]. If this is the real class instantiation of the
     generic class, these are the types that fill in the generics.
     e.g. class StringToIntMap = HashMap(string, int), generics
     becomes [string, int]. If this is a regular class def, e.g.
     class MyClass:, then this list is empty. *)
  generics: typ list;
  static_vars: assign list;
  required_vars: bind list;
  optional_vars: assign list;
  methods: fdecl list;
}

type p_unit =
  Stmt of stmt
| Fdecl of fdecl
| Classdecl of classdecl

type program = p_unit list

(* Begin visualization functions *)
let rec mapiplus i f = function
    [] -> []
  | a::l -> let r = f i a in r :: mapiplus (i + 1) f l

let mapiplus plus f l = mapiplus plus f l

let get_label_without_suffix suffixed_name unsuffixed_name = suffixed_name ^ " [label=\"" ^ unsuffixed_

let new_suffix existing_suffix new_index = existing_suffix ^ (string_of_int new_index)

let get_combine_function start_node subtuple = (start_node ^ " -> " ^ fst subtuple) :: snd subtuple

let get_single_node_generator node_name suffix subconverter subelement =
  let start_node = node_name ^ suffix in
  let subgraphs = (subconverter suffix 0 subelement) in
  (start_node, (get_label_without_suffix start_node node_name)::(get_combine_function start_node subgrap

let get_multi_node_generator node_name suffix subconverter input_list =
  let start_node = node_name ^ suffix in
  let subgraphs = (List.mapi (subconverter suffix) input_list) in (* List<Tuple<StartNodeString, List<S
  (start_node, (get_label_without_suffix start_node node_name)::List.concat (List.map (get_combine_func

let combine_list node_name suffix input_list =
  let start_node = node_name ^ suffix in
  (start_node, (get_label_without_suffix start_node node_name)::List.concat (List.map (get_combine_func

let get_op_node_label name suffix symbol = name ^ suffix ^ " [label=\"" ^ symbol ^ "\", color=transpare
let get_prim_node_label name suffix = name ^ suffix ^ " [label=\"" ^ name ^ "\", color=aliceblue, fillc
let get_class_node_label name suffix classname = name ^ suffix ^ " [label=\"" ^ classname ^ "\", color=
let get_literal_node name suffix value = name ^ suffix ^ " [label=\"" ^ value ^ "\", color=green, fillc

let string_of_id existing_suffix new_index id_string =
  let suffix = new_suffix existing_suffix new_index in
  ("id" ^ suffix, ["id" ^ suffix ^ " [label=\"id: " ^ id_string ^ "\" fontcolor=red]"])

let rec string_of_typ existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  Primitive(primitive) -> string_of_primitive suffix 0 primitive
| Class(class_string) -> ("class" ^ suffix, [get_class_node_label "class" suffix class_string])
| Array(typ) -> combine_list "array_type" suffix ([string_of_typ suffix 0 typ])
| NullType -> ("NULL" ^ suffix, [get_class_node_label "NULL" suffix "NULL"])
and string_of_primitive existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
| Int -> ("int" ^ suffix, [get_prim_node_label "int" suffix])
| Long -> ("long" ^ suffix, [get_prim_node_label "long" suffix])
| Float -> ("float" ^ suffix, [get_prim_node_label "float" suffix])
| Char -> ("char" ^ suffix, [get_prim_node_label "char" suffix])
| String -> ("string" ^ suffix, [get_prim_node_label "string" suffix])
| Bool -> ("boolean" ^ suffix, [get_prim_node_label "boolean" suffix])
| Void -> ("void" ^ suffix, [get_prim_node_label "void" suffix])
and string_of_expr existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  IntLiteral(integer) -> ("intlit" ^ suffix, [get_literal_node "intlit" suffix (string_of_int integer)]
| LongLiteral(long) -> ("longlit" ^ suffix, [get_literal_node "longlit" suffix (Int64.to_string long)])
| FloatLiteral(f) -> ("floatlit" ^ suffix, [get_literal_node "floatlit" suffix f])
| CharLiteral(c) -> ("charlit" ^ suffix, [get_literal_node "charlit" suffix (String.make 1 c)])
| StringLiteral(s) -> ("stringlit" ^ suffix, [get_literal_node "stringlit" suffix s])
| BoolLiteral(b) -> ("boollit" ^ suffix, [get_literal_node "boollit" suffix (string_of_bool b)])
| Id(id_string) -> string_of_id suffix 0 id_string
| NullExpr -> ("nullexpr" ^ suffix, [get_literal_node "nullexpr" suffix "NULL"])
| Self -> string_of_id suffix 0 "self"
| Call(call) -> string_of_call suffix 0 call
| ObjectInstantiation(id_string, exprs) -> combine_list "object_instantiation" suffix ([string_of_id su
| ObjectVariableAccess(object_variable_access) -> string_of_object_variable_access suffix 0 object_vari
| ArrayAccess(array_access) -> string_of_array_access suffix 0 array_access
| ArrayLiteral(exprs) -> get_multi_node_generator "array_literal" suffix string_of_expr exprs
| DefaultArray(typ, exprs) -> combine_list "default_array" suffix ([string_of_typ suffix 0 typ] @ (mapi
| Binop(expr1, binop, expr2) -> combine_list "binop" suffix ([string_of_expr suffix 0 expr1] @ [string_
| Unop(unaryop, expr) -> combine_list "unaryop" suffix ([string_of_unaryop suffix 0 unaryop] @ [string_
| Assign(assign) -> string_of_assign suffix 0 assign
| Update(update) -> string_of_update suffix 0 update
and string_of_array_access existing_suffix new_index array_access =
  let suffix = new_suffix existing_suffix new_index in
  let expr1 = (fst array_access) in
  let expr2 = (snd array_access) in
  combine_list "array_access" suffix ([string_of_expr suffix 0 expr1] @ [string_of_expr suffix 1 expr2]
and string_of_call existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  FuncCall(id_string, exprs) -> combine_list "func_call" suffix ([string_of_id suffix 0 id_string] @ (ma
| MethodCall(expr1, id2, exprs) -> combine_list "method_call" suffix ([string_of_expr suffix 0 expr1] @
and string_of_assign existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  RegularAssign(typ, id_string, expr) -> combine_list "assign" suffix ([string_of_typ suffix 0 typ] @ [
and string_of_update existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  RegularUpdate(id_string, updateop, expr) -> combine_list "update" suffix ([string_of_id suffix 0 id_s
| ObjectVariableUpdate(object_variable_access, updateop, expr) -> combine_list "update" suffix ([string
| ArrayAccessUpdate(array_access, updateop, expr) -> combine_list "update" suffix ([string_of_array_acc
and string_of_binoperator existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  Plus -> ("plus" ^ suffix, [get_op_node_label "plus" suffix "+"])
| Subtract -> ("subtract" ^ suffix, [get_op_node_label "subtract" suffix "-"])
| Times -> ("times" ^ suffix, [get_op_node_label "times" suffix "*"])
| Divide -> ("divide" ^ suffix, [get_op_node_label "divide" suffix "÷"])
| Modulo -> ("modulo" ^ suffix, [get_op_node_label "modulo" suffix "%"])
| DoubleEq -> ("doubleeq" ^ suffix, [get_op_node_label "doubleeq" suffix "=="])
| BoGT -> ("gt" ^ suffix, [get_op_node_label "gt" suffix ">"])
| BoLT -> ("lt" ^ suffix, [get_op_node_label "lt" suffix "<"])
| BoGTE -> ("gte" ^ suffix, [get_op_node_label "gte" suffix ">="])
| BoLTE -> ("lte" ^ suffix, [get_op_node_label "lte" suffix "<="])
| BoOr -> ("or" ^ suffix, [get_op_node_label "or" suffix "or"])
| BoAnd -> ("and" ^ suffix, [get_op_node_label "and" suffix "and"])
and string_of_updateop existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
| Eq -> ("eq" ^ suffix, [get_op_node_label "eq" suffix "="])
and string_of_unaryop existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  Not -> ("not" ^ suffix, [get_op_node_label "not" suffix "not"])
| Neg -> ("neg" ^ suffix, [get_op_node_label "neg" suffix "-"])
and string_of_object_variable_access existing_suffix new_index = function
  { ova_class_name = class_name; ova_var_name = var_name; ova_is_static = true; _ } ->
    let suffix = new_suffix existing_suffix new_index in
    ("static_var_access" ^ suffix, ["static_var_access" ^ suffix ^ " [label=\"" ^ (class_name) ^ "." ^
| { ova_expr = expr; ova_var_name = var_name; ova_is_static = false; _ } ->
    let suffix = new_suffix existing_suffix new_index in
    combine_list "obj_var_access" suffix ([string_of_expr suffix 0 expr] @ [string_of_id suffix 1 var_na

let rec string_of_stmt existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  Expr(expr) -> get_single_node_generator "expr" suffix string_of_expr expr
| Return(expr) -> get_single_node_generator "return" suffix string_of_expr expr
| ReturnVoid -> combine_list "return" suffix [("returnvoid" ^ suffix, [get_literal_node "returnvoid" su
| If(expr, sl1, elifs, sl2) -> combine_list "if" suffix ([string_of_expr suffix 0 expr] @ (mapiplus 1 (
| Loop(expr1, expr2, sl1) -> combine_list "loop" suffix ([string_of_expr suffix 0 expr1] @ [string_of_e
and
string_of_elif existing_suffix new_index elif_tuple =
 let suffix = new_suffix existing_suffix new_index in
 combine_list "elif" suffix ([string_of_expr suffix 0 (fst elif_tuple)] @ (mapiplus 1 (string_of_stmt su

let string_of_bind existing_suffix new_index bind =
  let suffix = new_suffix existing_suffix new_index in
  combine_list "bind" suffix ([string_of_typ suffix 0 (fst bind)] @ [string_of_id suffix 1 (snd bind)])

let string_of_fdecl existing_suffix new_index fdecl =
  let suffix = new_suffix existing_suffix new_index in
  combine_list "fdecl" suffix ([string_of_id suffix 0 fdecl.fname] @ (mapiplus 1 (string_of_bind suffix

let string_of_classdecl existing_suffix new_index classdecl =
  let suffix = new_suffix existing_suffix new_index in
  combine_list "classdecl" suffix ([string_of_id suffix 0 classdecl.cname] @ (mapiplus 1 (string_of_ass

let string_of_p_unit existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function (* Takes a program unit and returns a Tuple<StartNodeString, List<String>> *)
  Stmt(stmt) -> get_single_node_generator "stmt" suffix string_of_stmt stmt
| Fdecl(fdecl) -> string_of_fdecl suffix 0 fdecl
| Classdecl(classdecl) -> string_of_classdecl suffix 0 classdecl

let string_of_program program = (* Takes a program object and returns a Tuple<StartNodeString, List<Str
  let suffix = "0" in
  get_multi_node_generator "program" suffix string_of_p_unit program

let graphviz_string_of_program program =
  "digraph G { \n" ^ (String.concat "\n" (snd (string_of_program program))) ^ "\n}"
```

9.3 boomc

```
#!/bin/sh

# Usage:
# Running "./boomc helloworld.boom" will compile the source to an exe.
# Running "./boomc -r helloworld.boom" will compile and automatically run the exe.
input_file=$1
should_execute=false
if [ "$1" = "-r" ];
then
  input_file=$2
  should_execute=true
fi

without_extension=$(echo "$input_file" | cut -f 1 -d '.')
ll=".ll"
s=".s"
exe=".exe"
ll_file="$without_extension$ll"
s_file="$without_extension$s"
exe_file="$without_extension$exe"

make clean
make
./boomslang.native $input_file > $ll_file
```

```
llc -relocation-model=pic $ll_file > $s_file
gcc -c libfuncs.c
cc -o $exe_file $s_file libfuncs.o

if [ "$should_execute" = true ];
then
./$exe_file
fi
```

9.4 boomslang.ml

```
(* Top-level of the Boomslang compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./boomslang.native [-a|-s|-l|-c] [file.boom]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.read_next_token lexbuf in
  match !action with
    Ast -> print_string (Ast.graphviz_string_of_program ast)
  | _ -> let sast = Semant.check ast in
         match !action with
           Ast -> ()
         | Sast -> print_string (Sast.graphviz_string_of_sprogram sast)
         | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
         | Compile -> let m = Codegen.translate sast in
                      Llvm_analysis.assert_valid_module m;
                      print_string (Llvm.string_of_llmodule m)
```

9.5 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:
```

```
http://llvm.moe/
http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
module S = Semant
open Sast

module StringMap = Map.Make(String)
module SignatureMap = S.SignatureMap
module StringHash = Hashtbl.Make(struct
  type t = string (* type of keys *)
  let equal x y = x = y (* use structural comparison *)
  let hash = Hashtbl.hash (* generic hash function *)
end)
(* rather then defining all the array types, create a hashtable and only define the ones that
 * are used in this program. This is because there are infinite types of arrays
 * eg. int[] , int[][], int[][][], ... *)
module ArrayTypHash = Hashtbl.Make(struct
  type t = A.typ (* type of keys *)
  let equal x y = x = y (* use structural comparison *)
  let hash = Hashtbl.hash (* generic hash function *)
end)

type symbol_table_entry = {
  llvalue: L.llvalue;
  typ: A.typ;
}

let rec lookup v_symbol_tables s =
  match v_symbol_tables with
  [] -> raise (Failure ("undeclared identifier " ^ s))
  | hd::tl -> try (StringHash.find hd s).llvalue
              with Not_found -> (lookup tl s)

let rec lookup_class_name v_symbol_tables s =
  match v_symbol_tables with
  [] -> raise (Failure ("could not find a class name for id " ^ s))
  | hd::tl -> try (match (StringHash.find hd s).typ with
                     A.Class(class_name) -> class_name
                   | _ -> (lookup_class_name tl s))
              with Not_found -> (lookup_class_name tl s)

let get_static_var_name class_name var_name = class_name ^ "." ^ var_name

let get_class_name kind v_symbol_tables = function
  (_, SSelf) -> lookup_class_name v_symbol_tables "self"
| (A.Class(class_name), _) -> class_name
| _ -> raise (Failure("The LHS expression of a " ^ kind ^ " is expected to be a class type."))

let check_not_zero_fname = function
```

```
  A.Primitive(A.Int) -> "check_int_not_zero"
| A.Primitive(A.Long) -> "check_long_not_zero"
| A.Primitive(A.Float) -> "check_float_not_zero"
| _ -> raise (Failure("check_not_zero is not supported for the given type."))

(* translate : Sast.program -> Llvm.module *)
let translate sp_units =

  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "Boomslang" in

  (* Get types from the context *)
  let i1_t       = L.i1_type            context
  and i8_t       = L.i8_type            context
  and i32_t      = L.i32_type           context
  and i64_t      = L.i64_type           context
  and float_t    = L.double_type        context
  and str_t      = L.pointer_type       (L.i8_type context)
  and void_t     = L.void_type          context
  in

  (* arrays are implemented as a struct in Boomslang, this is so the length and the array itself
     can be put in one container. this hashtable stores all of the array types in the input program.
     The reason that it is a hashtable and not a predefined list is because there are infinitely many
     array types *)
  let arrtyp_table = ArrayTypHash.create 10 in

  let len_func_table = ArrayTypHash.create 10 in


  (* get array from array struct pointer *)
  let arrp_from_arrstruct s builder =
    L.build_struct_gep s 0 "arrp_from_arrstruct" builder in

  (* get size of array from array struct pointer *)
  let size_from_arrstruct s builder =
    L.build_struct_gep s 1 "" builder in

  (* Need to build a struct type for every class *)
  let class_name_to_named_struct =
    let helper m e = match e with
      SClassdecl(scd) -> (StringMap.add scd.scname (L.named_struct_type context scd.scname) m)
    | _ -> m in
  List.fold_left helper StringMap.empty sp_units
  in

  (* remove option from type, WARNING: use this wisely! *)
  let remove_option stro = match stro with
    None -> raise (Failure ("attemted to evaluate None option"))
  | Some x -> x in
```

```
(* create llvm len() function for the given array type *)
let create_len_func (typ : A.typ) =
  if (not (ArrayTypHash.mem len_func_table typ)) then (
    let t = if ArrayTypHash.mem arrtyp_table typ then
              ArrayTypHash.find arrtyp_table typ
            else raise (Failure ("arrtyp is not in table")) in
    let func = L.define_function "len" (L.var_arg_function_type i32_t [| L.pointer_type t |]) the_modu
    let _ = ArrayTypHash.add len_func_table typ func in
    let func_builder = L.builder_at_end context (L.entry_block func) in
    let alloc = L.build_alloca (L.pointer_type t) "" func_builder in          (* make space on the st
    let formal = List.hd (Array.to_list (L.params func)) in  (* get the llvalue of the only formal ar
    let _ = L.build_store formal alloc func_builder in
    let loaded_formal = L.build_load alloc "" func_builder in
    let sizep = size_from_arrstruct loaded_formal func_builder in
    let size = L.build_load sizep "size" func_builder in
    let _ = L.build_ret size func_builder in ()
    )
  else  ()  in

(* Return the LLVM type for a Boomslang type *)
let rec ltype_of_typ = function
  A.Primitive(A.Int)    -> i32_t
| A.Primitive(A.Long)   -> i64_t
| A.Primitive(A.Float)  -> float_t
| A.Primitive(A.Char)   -> i8_t
| A.Primitive(A.String) -> str_t
| A.Primitive(A.Bool)   -> i1_t
| A.Primitive(A.Void)   -> void_t
(* Classes, arrays, and null type *)
(* Classes always get passed around as pointers to the memory where the full struct is stored *)
| A.Class(class_name)   -> L.pointer_type (StringMap.find class_name class_name_to_named_struct)
| A.Array(typ)          ->
  let rec helper typ suffix = (* returns the typ of the array *)
    if ArrayTypHash.mem arrtyp_table typ then
      ArrayTypHash.find arrtyp_table typ
    else (
      match typ with
        A.Array(A.Array(ityp))  -> (* is an array of arrays *)
        let lityp = helper (A.Array(ityp)) "[]" in
        let arr_t = L.named_struct_type context ((remove_option (L.struct_name lityp)) ^ suffix) in
        L.struct_set_body arr_t [| (L.pointer_type (L.pointer_type lityp)) ; i32_t |] false;
        ArrayTypHash.add arrtyp_table typ arr_t;
        create_len_func typ;
        arr_t
      | A.Array(ityp) -> (* is an array of primitives *)
        let arr_t = L.named_struct_type context ((str_of_typ ityp) ^ suffix) in
        L.struct_set_body arr_t [| (L.pointer_type (ltype_of_typ ityp)) ; i32_t |] false;
        ArrayTypHash.add arrtyp_table typ arr_t;
        create_len_func typ;
        arr_t
     | _ -> raise (Failure ("ltyp_of_typ failure"))
    )
  in
  if ArrayTypHash.mem arrtyp_table (A.Array(typ)) then   (* check if array struct is already in hasht
    L.pointer_type (ArrayTypHash.find arrtyp_table (A.Array(typ)))
  else
    L.pointer_type (helper (A.Array(typ)) "[]")
| _                     -> void_t (* TODO remove this and fill in other types *)
in
let get_bind_from_assign = function
  SRegularAssign(typ, name, _) -> (typ, name)
| SStaticAssign(_, typ, name, _) -> (typ, name)
in
let helper = function
  SClassdecl(scd) ->
    (* In order to have it work recursively, first you have to use a named struct type.
       Then you have to fill in the body. *)
    let elts = (Array.of_list (List.map ltype_of_typ (List.map (fst) (scd.srequired_vars @ (List.map g
    L.struct_set_body (StringMap.find scd.sname class_name_to_named_struct) elts false
  | _ -> () in
let _ = List.iter helper sp_units in
let get_lvalue_of_bool = function
  true -> (L.const_int (ltype_of_typ (A.Primitive(A.Bool))) 1)
| false -> (L.const_int (ltype_of_typ (A.Primitive(A.Bool))) 0)
in

(* define the default values for all the types *)
let default_val_of_typ typ builder = match typ with
  A.Primitive(A.Int)    -> L.const_int i32_t 0
| A.Primitive(A.Long)   -> L.const_int i64_t 0
| A.Primitive(A.Float)  -> L.const_float float_t 0.0
| A.Primitive(A.Char)   -> L.const_int i8_t 0
| A.Primitive(A.String) -> L.build_global_stringptr "" "" builder
| A.Primitive(A.Bool)   -> L.const_int i1_t 0
| A.Class(name)         -> L.const_pointer_null (ltype_of_typ (A.Class(name)))
(* TODO: make the default type for arrays be an array of size 0 *)
| _                     -> L.const_null i32_t (* TODO remove this and fill in other types *)
in

(* create a map of all of the built in functions *)
let built_in_map =
 let built_in_funcs : (string * A.typ * (A.typ list)) list =
   let convert (fs : (function_signature * A.typ)) =
     (((fst fs).fs_name), (snd fs), ((fst fs).formal_types)) in
   List.map convert Semant.built_in_funcs in
 let helper m e = match e with fun_name, ret_t, arg_ts ->
  let arg_t_arr = Array.of_list
                   (List.fold_left (fun s e -> s @ (if (e = (A.Primitive(A.Void))) then [(L.pointer_ty
  let func = L.declare_function fun_name
             (L.var_arg_function_type (ltype_of_typ ret_t) arg_t_arr) the_module in
  let signature = { fs_name = fun_name; formal_types = arg_ts } in
  SignatureMap.add signature func m in
 List.fold_left helper SignatureMap.empty built_in_funcs in

(* create a map of all the user defined functions *)
let user_func_map =
  let helper m e = match e with
    SFdecl(sf) ->
```

```
      let func_t = L.function_type (ltype_of_typ sf.srtype) (Array.of_list
        (List.fold_left (fun s e -> match e with typ, _ -> s @ [ltype_of_typ typ])
        [] sf.sformals)) in
      let func = L.define_function (sf.sfname ^ "_usr") func_t the_module in
      let signature = { fs_name = sf.sfname; formal_types =
        List.fold_left (fun s (typ, _) -> s @ [typ]) [] sf.sformals } in
      SignatureMap.add signature func m
  | _ -> m in
  List.fold_left helper SignatureMap.empty sp_units in

(* builds a map from string to map of function signature to LLVM function definition *)
let class_signature_map =
  let helper1 m1 e1 = match e1 with
    SClassdecl(scd) ->
      let helper2 m2 mdecl =
        let func_t = L.function_type (ltype_of_typ mdecl.srtype) (Array.of_list
          (ltype_of_typ (A.Class(scd.scname))::
          (List.fold_left (fun s e -> match e with typ, _ -> s @ [ltype_of_typ typ]) [] mdecl.sformal
        ) in
        let func = L.define_function (mdecl.sfname ^ "_classmethod") func_t the_module in
        let signature = { fs_name = mdecl.sfname; formal_types =
          List.fold_left (fun s (typ, _) -> s @ [typ]) [] mdecl.sformals } in
        SignatureMap.add signature func m2 in
      StringMap.add scd.scname (List.fold_left helper2 SignatureMap.empty scd.smethods) m1
  | _ -> m1 in
  List.fold_left helper1 StringMap.empty sp_units in

(* to get an element from the struct, we have to use its index in the struct, rather
   than the name. this is annoying, but there doesn't seem to be a way to avoid it. *)
let class_name_to_decl =
  let helper1 m1 e1 = match e1 with
    SClassdecl(scd) -> StringMap.add scd.scname scd m1
  | _ -> m1 in
List.fold_left helper1 StringMap.empty sp_units in
let rec find x lst = (* stolen from https://stackoverflow.com/questions/31279920/finding-an-item-in-a
  match lst with
  | [] -> raise (Failure("Index for v_name " ^ x ^ " not found"))
  | h :: t -> if x = h then 0 else 1 + find x t
in
let get_index_in_class class_name v_name =
  let scdecl = StringMap.find class_name class_name_to_decl in
  (find v_name (List.map (snd) (scdecl.srequired_vars @ (List.map get_bind_from_assign scdecl.soptiona
in
let get_name_of_assign = function
  SRegularAssign(_, lhs_name, _) -> lhs_name
| SStaticAssign(_, _, lhs_name, _) -> lhs_name
in
let is_static_variable class_name v_name =
  let scdecl = StringMap.find class_name class_name_to_decl in
  List.mem v_name (List.map get_name_of_assign scdecl.sstatic_vars)
in

(* expression builder *)
let rec build_expr builder v_symbol_tables (exp : sexpr) = match exp with
```

```
  _, SIntLiteral(i)      -> L.const_int i32_t i
| _, SLongLiteral(l)     -> L.const_of_int64 i64_t l true
| _, SFloatLiteral(f)    -> L.const_float_of_string float_t f
| _, SCharLiteral(c)     -> L.const_int i8_t (Char.code c)
| _, SStringLiteral(str) -> L.build_global_stringptr str "STRINGLITERAL" builder
| _, SBoolLiteral(true)  -> L.const_int i1_t 1
| _, SBoolLiteral(false) -> L.const_int i1_t 0
| _, SId(id)             -> L.build_load (lookup v_symbol_tables id) id builder
| _, SSelf               -> L.build_load (lookup v_symbol_tables "self") "self" builder
| typ, SNullExpr         -> L.const_pointer_null (ltype_of_typ typ)
(* Function calls *)
| typ, SCall(sc) -> (match sc with
    (* Nulls must be handled with care *)
      SFuncCall("null_to_string", [(_, SNullExpr)]) -> build_expr builder v_symbol_tables (A.Primitive
    | SFuncCall("len", [(arrtyp, expr)])  -> L.build_call (ArrayTypHash.find len_func_table arrtyp)
                                               [| build_expr builder v_symbol_tables (arrtyp, expr) |]
    | SFuncCall(func_name, expr_list) ->
        let expr_typs = List.fold_left (fun s (t, _) -> s @ [t]) [] expr_list in
        let signature_with_possible_nulls = { fs_name = func_name; formal_types = expr_typs } in
        if SignatureMap.mem signature_with_possible_nulls built_in_map then (* is a built in func *)
          L.build_call (SignatureMap.find signature_with_possible_nulls built_in_map)
          (Array.of_list (List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tables e])
          [] expr_list))
          (if typ = A.Primitive(A.Void) then "" else (func_name ^ "_res"))
          builder
        else (* is a user defined func *)
          let matching_signature = S.find_matching_signature signature_with_possible_nulls user_func_
          L.build_call (SignatureMap.find matching_signature user_func_map)
          (Array.of_list (List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tables e])
          [] expr_list))
          (if typ = A.Primitive(A.Void) then "" else (func_name ^ "_res"))
          builder
    | SMethodCall(expr, method_name, expr_list) ->
        let expr_typs = List.fold_left (fun s (t, _) -> s @ [t]) [] expr_list in
        let signature_with_possible_nulls = { fs_name = method_name; formal_types = expr_typs } in
        let class_name = get_class_name "method call" v_symbol_tables expr in
        let class_signatures = StringMap.find class_name class_signature_map in
        let signature = S.find_matching_signature signature_with_possible_nulls class_signatures in
        if SignatureMap.mem signature class_signatures then (* is a user defined method *)
          let expr' = build_expr builder v_symbol_tables expr in
          L.build_call (SignatureMap.find signature class_signatures)
          (Array.of_list (expr'::(List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tables
          [] (expr_list))))
          (if typ = A.Primitive(A.Void) then "" else (class_name ^ "_" ^ method_name ^ "_res"))
          builder
        else raise (Failure ("method " ^ method_name ^ " not found on class " ^ class_name))
    )
| _, SObjectInstantiation(class_name, expr_list) ->
  let expr_typs = List.fold_left (fun s (t, _) -> s @ [t]) [] expr_list in
  let signature_with_possible_nulls = { fs_name = "construct"; formal_types = expr_typs } in
  let class_signatures = StringMap.find class_name class_signature_map in
  let signature = S.find_matching_signature signature_with_possible_nulls class_signatures in
  if SignatureMap.mem signature class_signatures then (* found a valid constructor *)
    (* first, create an empty struct of the right type. *)
```

```
        (* this is the one place where we DON'T use the pointer version of the struct type *)
        let struct_malloc = L.build_malloc (* (ltype_of_typ (A.Class(class_name))) *) (StringMap.find cla
        (* then call the constructor to initialize it properly *)
        ignore (L.build_call (SignatureMap.find signature class_signatures)
          (Array.of_list (struct_malloc::(List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tabl
           [] expr_list)))
          "" builder); struct_malloc  (* We call the constructor function, but return the llvalue for the
      else raise (Failure ("constructor function for " ^ class_name ^ " not found"))
    | _, SObjectVariableAccess(sova) ->
       let expr = sova.sova_sexpr in
       let class_name = sova.sova_class_name in
       let var_name = sova.sova_var_name in
       let is_static = sova.sova_is_static in
       if is_static then
         (* This is the case where we access static var x like MyClass.x instead of myinstance.x *)
         (let static_var_name = get_static_var_name class_name var_name in
          L.build_load (lookup v_symbol_tables static_var_name) static_var_name builder)
       else
         let class_name = get_class_name "object variable access" v_symbol_tables expr in
         if (is_static_variable class_name var_name) then
           (let static_var_name = get_static_var_name class_name var_name in
            L.build_load (lookup v_symbol_tables static_var_name) static_var_name builder)
         else
           (let index_in_class = (get_index_in_class class_name var_name) in
            (* Check that the object whose variable we are trying to access isn't null *)
            let expr' = build_expr builder v_symbol_tables expr in
            let bitcast = L.build_bitcast expr' (L.pointer_type i8_t) "bcast" builder in
            let _ = L.build_call (SignatureMap.find ({ fs_name = "check_not_null"; formal_types = [A.Prim
            let gep = L.build_struct_gep expr' index_in_class var_name builder in
            L.build_load gep "" builder)
    | _, SArrayAccess(sexpr1, sexpr2) ->
        let n = build_expr builder v_symbol_tables sexpr2 in (* the integer (as an llvalue) we are indexi
        let structp = build_expr builder v_symbol_tables sexpr1 in
        let arrpp = arrp_from_arrstruct structp builder in
        let arrp = L.build_load arrpp "arr" builder in
        let elemp = L.build_gep arrp [| n |] "gep_of_arr" builder in
        L.build_load elemp "arr_elem" builder
    | A.Array(typ) , SArrayLiteral(sexpr_list) ->
        (* create list of llvalue from the evaluated sexpr list *)
        let llvalue_arr = List.fold_left (fun s sexpr -> s @ [build_expr builder v_symbol_tables sexpr])
                           [] sexpr_list in

        (* always put the array literal in the heap, maybe find a way to free this memory later *)
        let arrt = L.build_malloc (L.array_type (ltype_of_typ typ) (List.length sexpr_list)) "" builder i
        (* 'cast' this llvm array type into of a pointer type *)
        let arrp = L.build_gep arrt [| L.const_int i64_t 0; L.const_int i64_t 0 |] "arrp" builder in
        (* for each element of the array, gep and store value *)
        let _ = List.fold_left
                (fun i e ->  ignore (L.build_store e (L.build_gep arrp [| L.const_int i64_t i |]
                "" builder) builder); i + 1) 0 llvalue_arr in
        (* malloc the array struct and put the necessary elements inside *)
        let struct_tp = ltype_of_typ (A.Array(typ)) in              (* get the arr struct pointer *)
        let struct_t = L.element_type struct_tp in
        let structp = L.build_malloc struct_t "arr_structp" builder in
        let _ =
          ignore (L.build_store arrp (arrp_from_arrstruct structp builder) builder);
          L.build_store (L.const_int i32_t (List.length sexpr_list)) (size_from_arrstruct structp builder

        structp
    | A.Array(typ1), SDefaultArray(_, sexprs) ->
        let base_typ =
          let rec helper typ = match typ with
            A.Array(ityp) -> helper ityp
          | _             -> typ in
          helper typ1 in
       let default_val = default_val_of_typ base_typ builder in (* get the default value *)
       (* evaluate exprs into int list *)
       let ints =
         let helper s e = match e with
           _, SIntLiteral(v) -> v::s
         | _, _              -> raise (Failure ("Default array failure, size not int literl")) in
         List.fold_left helper [] sexprs in
       let rec build_arr typ expr_list = match typ, expr_list with
         A.Array(ityp), fst::[]  ->
           let arrt = L.build_malloc (L.array_type (ltype_of_typ ityp) fst) "arrt" builder in
           let arrp = L.build_gep arrt [| L.const_int i64_t 0; L.const_int i64_t 0 |] "arrp" builder in
           let rec helper i = match i with
             _ when i < 0 -> ()
           | _ -> ignore (L.build_store default_val (L.build_gep arrp [| L.const_int i64_t i |] "" builde
                   helper (i - 1) in
           let _ = helper (fst - 1) in
           (* malloc the actual array *)
           let struct_tp = ltype_of_typ typ in
           let struct_t = L.element_type struct_tp in
           let structp = L.build_malloc struct_t "arr_structp" builder in
           let _ =
             ignore (L.build_store arrp (arrp_from_arrstruct structp builder) builder);
             L.build_store (L.const_int i32_t fst) (size_from_arrstruct structp builder) builder in
           structp
       | A.Array(ityp), fst::snd ->
           (* get the array type *)
           let arrt = L.build_malloc (L.array_type (ltype_of_typ ityp) fst) "arrt" builder in
           let arrp = L.build_gep arrt [| L.const_int i64_t 0; L.const_int i64_t 0 |] "arrp" builder in
           let rec helper i = match i with
             _ when i < 0 -> ()
           | _ ->
               ignore (L.build_store (build_arr ityp snd) (L.build_gep arrp [| L.const_int i64_t i |] ""
                              builder); helper (i - 1) in
           let _ = helper (fst - 1) in
           (* malloc the actual array *)
           let struct_tp = ltype_of_typ typ in
           let struct_t = L.element_type struct_tp in
           let structp = L.build_malloc struct_t "arr_structp" builder in
           let _ =
             ignore (L.build_store arrp (arrp_from_arrstruct structp builder) builder);
             L.build_store (L.const_int i32_t fst) (size_from_arrstruct structp builder) builder in
           structp
       | _ -> raise (Failure ("default array generation failed: unrecognize pattern in build_arr"))
       in
       build_arr (A.Array(typ1)) ints

    (* == is the only binop that can apply to any two types. *)
    | _, SBinop(sexpr1, A.DoubleEq, sexpr2) ->
        let sexpr1' = build_expr builder v_symbol_tables sexpr1
        and sexpr2' = build_expr builder v_symbol_tables sexpr2 in
        let sexpr1typ = (fst sexpr1)
        and sexpr2typ = (fst sexpr2)
        and sexpr1sx = (snd sexpr1)
        and sexpr2sx = (snd sexpr2)
        in
        if ((sexpr1typ = A.NullType || sexpr1sx = SNullExpr) && (sexpr2typ = A.NullType || sexpr2sx = SNul
          (get_lvalue_of_bool true)
        else if (sexpr1typ = A.NullType || sexpr1sx = SNullExpr) then
          (* Only an object (not a primitive nor array) may be Null. *)
          (match sexpr2typ with
               A.Class(_) -> L.build_is_null sexpr2' "" builder
             | _ -> L.const_int (ltype_of_typ (A.Primitive(A.Bool))) 0)
        else if (sexpr2typ = A.NullType || sexpr2sx = SNullExpr) then
          (match sexpr1typ with
               A.Class(_) -> L.build_is_null sexpr1' "" builder
             | _ -> L.const_int (ltype_of_typ (A.Primitive(A.Bool))) 0)
        else
          (match sexpr1typ with
            (* Even though these look different, they are all integers internally *)
              A.Primitive(A.Int)
            | A.Primitive(A.Long)
            | A.Primitive(A.Char)
            | A.Primitive(A.Bool) -> L.build_icmp L.Icmp.Eq sexpr1' sexpr2' "tmp" builder
            (* Floats are similar *)
            | A.Primitive(A.Float) -> L.build_fcmp L.Fcmp.Oeq sexpr1' sexpr2' "tmp" builder
            (* Strings are not comparable using an LLVM native function,
               so we call our own C function here. *)
            | A.Primitive(A.String) ->
                let signature = { fs_name = "compare_strings"; formal_types = [(fst sexpr1); (fst sexpr2)
                L.build_call (SignatureMap.find signature built_in_map)
                  (Array.of_list (List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tables e])
                  (signature.fs_name ^ "_res") builder
            | A.Class(_) -> L.build_icmp L.Icmp.Eq (L.const_int i64_t 0) (L.build_ptrdiff sexpr1' sexpr2'
            | A.Array(_) -> L.build_icmp L.Icmp.Eq (L.const_int i64_t 0) (L.build_ptrdiff sexpr1' sexpr2'
            | _          -> raise (Failure ("SBinop matching error"))
          )
    (* Integer and long binops *)
    | _, SBinop(((A.Primitive(A.Int), _) as sexpr1), binop, ((A.Primitive(A.Int), _) as sexpr2))
    | _, SBinop(((A.Primitive(A.Long), _) as sexpr1), binop, ((A.Primitive(A.Long), _) as sexpr2)) ->
        let sexpr1' = build_expr builder v_symbol_tables sexpr1
        and sexpr2' = build_expr builder v_symbol_tables sexpr2 in
        (match binop with
            A.Plus         -> L.build_add
          | A.Subtract     -> L.build_sub
          | A.Times        -> L.build_mul
          | A.Divide       ->
```

```
            (* Check no divide by zero *)
            let typ = (fst sexpr1) in
            let error_message = build_expr builder v_symbol_tables ((A.Primitive(A.String)), SStringLitera
            let _ = L.build_call (SignatureMap.find ({ fs_name = (check_not_zero_fname typ); formal_types
            L.build_sdiv (* signed division*)
          | A.Modulo       ->
            (* Check no mod by zero *)
            let typ = (fst sexpr1) in
            let error_message = build_expr builder v_symbol_tables ((A.Primitive(A.String)), SStringLitera
            let _ = L.build_call (SignatureMap.find ({ fs_name = (check_not_zero_fname typ); formal_types
            L.build_srem (* signed remainder *)
          | A.DoubleEq     -> L.build_icmp L.Icmp.Eq (* ordered and equal to *)
          | A.BoGT         -> L.build_icmp L.Icmp.Sgt (* ordered and greater than *)
          | A.BoLT         -> L.build_icmp L.Icmp.Slt (* ordered and less than *)
          | A.BoGTE        -> L.build_icmp L.Icmp.Sge (* etc. *)
          | A.BoLTE        -> L.build_icmp L.Icmp.Sle
          | _ -> raise (Failure("Found ineligible binop for int/long operands"))
      ) sexpr1' sexpr2' "tmp" builder
  (* Float binops *)
  | _, SBinop(((A.Primitive(A.Float), _) as sexpr1), binop, ((A.Primitive(A.Float), _) as sexpr2)) ->
      let sexpr1' = build_expr builder v_symbol_tables sexpr1
      and sexpr2' = build_expr builder v_symbol_tables sexpr2 in
      (match binop with
          A.Plus         -> L.build_fadd
        | A.Subtract     -> L.build_fsub
        | A.Times        -> L.build_fmul
        | A.Divide       ->
          (* Check no divide by zero *)
          let typ = (fst sexpr1) in
          let error_message = build_expr builder v_symbol_tables ((A.Primitive(A.String)), SStringLitera
          let _ = L.build_call (SignatureMap.find ({ fs_name = (check_not_zero_fname typ); formal_types
          L.build_fdiv (* signed division*)
        | A.Modulo       ->
          (* Check no mod by zero *)
          let typ = (fst sexpr1) in
          let error_message = build_expr builder v_symbol_tables ((A.Primitive(A.String)), SStringLitera
          let _ = L.build_call (SignatureMap.find ({ fs_name = (check_not_zero_fname typ); formal_types
          L.build_frem (* signed remainder *)
        | A.DoubleEq     -> L.build_fcmp L.Fcmp.Oeq (* ordered and equal to *)
        | A.BoGT         -> L.build_fcmp L.Fcmp.Ogt (* ordered and greater than *)
        | A.BoLT         -> L.build_fcmp L.Fcmp.Olt (* ordered and less than *)
        | A.BoGTE        -> L.build_fcmp L.Fcmp.Oge (* etc. *)
        | A.BoLTE        -> L.build_fcmp L.Fcmp.Ole
        | _ -> raise (Failure("Found ineligible binop for float operands"))
      ) sexpr1' sexpr2' "tmp" builder
  (* String binops (the only one supported is + for concatenate) *)
  | _, SBinop(((A.Primitive(A.String), _) as sexpr1), binop, ((A.Primitive(A.String), _) as sexpr2)) ->
      (match binop with
          A.Plus         ->
            let signature = { fs_name = "concat_strings"; formal_types = [(fst sexpr1); (fst sexpr2)] }
            L.build_call (SignatureMap.find signature built_in_map)
              (Array.of_list (List.fold_left (fun s e -> s @ [build_expr builder v_symbol_tables e]) []
              (signature.fs_name ^ "_res") builder
        | _ -> raise (Failure("Found ineligible binop for string operands"))
```

```
    )
(* Boolean binops *)
| _, SBinop(((A.Primitive(A.Bool), _) as sexpr1), binop, ((A.Primitive(A.Bool), _) as sexpr2)) ->
    let sexpr1' = build_expr builder v_symbol_tables sexpr1
    and sexpr2' = build_expr builder v_symbol_tables sexpr2 in
    (match binop with
        A.BoOr         -> L.build_or
      | A.BoAnd        -> L.build_and
      | _ -> raise (Failure("Found ineligible binop for boolean operands"))
    ) sexpr1' sexpr2' "tmp" builder
(* Unary operators *)
| _, SUnop(A.Not, sexpr1) ->
    let sexpr1' = build_expr builder v_symbol_tables sexpr1 in
    L.build_not sexpr1' "tmp" builder
| _, SUnop(A.Neg, ((A.Primitive(A.Int), _) as sexpr1))
| _, SUnop(A.Neg, ((A.Primitive(A.Long), _) as sexpr1)) ->
    let sexpr1' = build_expr builder v_symbol_tables sexpr1 in
    L.build_neg sexpr1' "tmp" builder
| _, SUnop(A.Neg, ((A.Primitive(A.Float), _) as sexpr1)) ->
    let sexpr1' = build_expr builder v_symbol_tables sexpr1 in
    L.build_fneg sexpr1' "tmp" builder
| A.Array(arrtyp), SAssign(SRegularAssign(_, name, sexpr)) ->
    let arrp = build_expr builder v_symbol_tables sexpr in (* get pointer for arr literal *)
    let arrp_typ = L.type_of arrp in
    if List.length v_symbol_tables = 1 then
      (* Build a global *)
      let global_symbol_table = List.hd v_symbol_tables in
      let declared_global = (L.declare_global arrp_typ name the_module) in
      let _ = L.set_initializer (L.const_null arrp_typ) declared_global in
      ((StringHash.add global_symbol_table name { llvalue = declared_global; typ = A.Array(arrtyp) })
      ignore(L.build_store arrp (lookup v_symbol_tables name) builder));
      arrp
    else
      (* Build a local. This means allocating space on the stack, and then
         storing the value of the expr there. *)
      let this_scopes_symbol_table = List.hd v_symbol_tables in
      let new_symbol_table_entry = { llvalue = (L.build_alloca arrp_typ name builder); typ = A.Array(a
      ((StringHash.add this_scopes_symbol_table name new_symbol_table_entry);
      ignore(L.build_store arrp (lookup v_symbol_tables name) builder));
      arrp
| _, SAssign(SRegularAssign(typ, name, sexpr)) ->
    (* Variables outside of classes and functions should be globals,
       those inside functions and classes should be locals.
       In Boomslang, we say that only entries going into the first
       (i.e. highest scope) symbol table are globals.
       Consider the following program:
       int x = 5
       if x > 2:
              int y = 1
       else:
              int y = 2
       In the above program, x is treated like a global variable, and
       y is treated like a local variable inside of main().
     *)
```

```
      let e' = build_expr builder v_symbol_tables sexpr in
      if List.length v_symbol_tables = 1 then
        (* Build a global *)
        let global_symbol_table = List.hd v_symbol_tables in
        (* This looks a little weird. Basically, we declare a global of the given type,
           then initialize it to the null version of the that type. Then inside main,
           we build a store to put the contents of the RHS into the global variable.
           For some reason this works but L.define_global with the RHS does not. *)
        let declared_global = (L.declare_global (ltype_of_typ typ) name the_module) in
        let _ = L.set_initializer (L.const_null (ltype_of_typ typ)) declared_global in
        ((StringHash.add global_symbol_table name { llvalue = declared_global; typ = typ });
        ignore(L.build_store e' (lookup v_symbol_tables name) builder));
        e'
      else
        (* Build a local. This means allocating space on the stack, and then
           storing the value of the expr there. *)
        let this_scopes_symbol_table = List.hd v_symbol_tables in
        let new_symbol_table_entry = { llvalue = (L.build_alloca (ltype_of_typ typ) name builder); typ =
        ((StringHash.add this_scopes_symbol_table name new_symbol_table_entry);
        ignore(L.build_store e' (lookup v_symbol_tables name) builder));
        e'
  | typ1, SAssign(SStaticAssign(class_name, typ2, var_name, sexpr)) ->
      (* . can never be part of an identifier in our language, but it can in LLVM.
         Thus we use the LLVM global name to handle static variables, as the class name
         basically gives each static var a unique prefix and unique id that won't
         conflict with other global variables or other class static variables. *)
      build_expr builder v_symbol_tables (typ1, (SAssign(SRegularAssign(typ2, (get_static_var_name class
  | _, SUpdate(SRegularUpdate(name, A.Eq, sexpr)) ->
      let e' = build_expr builder v_symbol_tables sexpr in
      ignore(L.build_store e' (lookup v_symbol_tables name) builder); e'
  | typ, SUpdate(SObjectVariableUpdate(sova, A.Eq, rhs_sexpr)) ->
      let lhs_sexpr = sova.sova_sexpr in
      let class_name = sova.sova_class_name in
      let var_name = sova.sova_var_name in
      let is_static = sova.sova_is_static in
      let rhs_expr' = build_expr builder v_symbol_tables rhs_sexpr in
      let lhs_expr' = build_expr builder v_symbol_tables lhs_sexpr in
      if is_static then
        let static_var_name = get_static_var_name class_name var_name in
        build_expr builder v_symbol_tables (typ, SUpdate(SRegularUpdate(static_var_name, A.Eq, rhs_sexp
      else
        let class_name = get_class_name "object variable update" v_symbol_tables lhs_sexpr in
        if (is_static_variable class_name var_name) then
          let static_var_name = get_static_var_name class_name var_name in
          build_expr builder v_symbol_tables (typ, SUpdate(SRegularUpdate(static_var_name, A.Eq, rhs_se
        else
          (* First check the LHS is not null before assigning to it. *)
          (let bitcast = L.build_bitcast lhs_expr' (L.pointer_type i8_t) "bcast" builder in
           let _ = L.build_call (SignatureMap.find ({ fs_name = "check_not_null"; formal_types = [A.Pri
           let gep = L.build_struct_gep lhs_expr' (get_index_in_class class_name var_name) var_name bui
           ignore(L.build_store rhs_expr' gep builder); rhs_expr')
  | _, SUpdate(SArrayAccessUpdate((sexpr_arr, sexpr_index), A.Eq, sexpr)) ->
      let newvalue = build_expr builder v_symbol_tables sexpr in
      let n = build_expr builder v_symbol_tables sexpr_index in (* the integer (as an llvalue) we are i
      let structp = build_expr builder v_symbol_tables sexpr_arr in (* load in structp *)
      let arrp = L.build_load (arrp_from_arrstruct structp builder) "arr" builder in
      let elemp = L.build_gep arrp [| n |] "gep_of_arr" builder in
      let _ = L.build_store newvalue elemp builder in
      newvalue
  | _ -> raise (Failure("unimplemented expr in codegen"))
  in

  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (instr builder) in
  (* statement builder *)
  let rec build_stmt the_function v_symbol_tables builder (ss : sstmt) = match ss with
    SExpr(se)   -> ignore (build_expr builder v_symbol_tables se); builder
  | SReturn(sexpr) -> ignore (L.build_ret (build_expr builder v_symbol_tables sexpr) builder);
                      builder
  | SReturnVoid -> ignore (L.build_ret_void builder); builder
  | SIf (predicate, then_stmts, elif_stmts, else_stmts) ->
    let bool_val = build_expr builder v_symbol_tables predicate in
    if ((List.length elif_stmts) = 0 && (List.length else_stmts) = 0) then
      let merge_bb = L.append_block context "merge" the_function in
      let b_br_merge = L.build_br merge_bb in
      let then_bb = L.append_block context "then" the_function in
      let _ = add_terminal (build_stmt_list the_function ((StringHash.create 10)::v_symbol_tables) (L.bu
      (ignore(L.build_cond_br bool_val then_bb merge_bb builder));
      L.builder_at_end context merge_bb
    else
      let then_bb = L.append_block context "then" the_function in
      let then_stmts_builder = build_stmt_list the_function ((StringHash.create 10)::v_symbol_tables) (
      let else_bb = L.append_block context "else" the_function in
      let else_stmts_builder =
        if (List.length elif_stmts = 0) then
          (build_stmt_list the_function ((StringHash.create 10)::v_symbol_tables) (L.builder_at_end con
        else
          let first_elif = List.hd elif_stmts in
          let first_elif_predicate = fst first_elif in
          let first_elif_stmts = snd first_elif in
          (build_stmt the_function ((StringHash.create 10)::v_symbol_tables) (L.builder_at_end context
      in
      (match (L.block_terminator (L.insertion_block then_stmts_builder)) with
         Some(_) -> (
                     (match (L.block_terminator (L.insertion_block else_stmts_builder)) with
                        Some(_) -> (ignore(L.build_cond_br bool_val then_bb else_bb builder);
                                    builder
                                   )
                      | None -> (let merge_bb = L.append_block context "merge" the_function in
                                 let b_br_merge = L.build_br merge_bb in
                                 let _ = add_terminal else_stmts_builder b_br_merge in
                                 ignore(L.build_cond_br bool_val then_bb else_bb builder);
                                 L.builder_at_end context merge_bb
                                )
                     )
                    )
      | None -> (let merge_bb = L.append_block context "merge" the_function in
                 let b_br_merge = L.build_br merge_bb in
                 let _ = add_terminal then_stmts_builder b_br_merge in
                 let _ = add_terminal else_stmts_builder b_br_merge in
                 ignore(L.build_cond_br bool_val then_bb else_bb builder);
                 L.builder_at_end context merge_bb
                )
    )
| SLoop(update, predicate, body_stmt_list) ->
  let pred_bb = L.append_block context "while" the_function in
  ignore(L.build_br pred_bb builder);

  let body_bb = L.append_block context "while_body" the_function in

  let while_stmts_builder = (build_stmt_list the_function ((StringHash.create 10)::v_symbol_tables) (
  let _ = (match (L.block_terminator (L.insertion_block while_stmts_builder)) with
     Some(_) -> () (* The body already returns, so no need to add the update expression to the end, n
   (* The body did not already return. So we need to do 2 things: Add the update expression and a ter
   | None -> add_terminal (build_stmt the_function v_symbol_tables while_stmts_builder (SExpr(update))
  ) in

  let pred_builder = L.builder_at_end context pred_bb in
  let bool_val = build_expr pred_builder v_symbol_tables predicate in

  let merge_bb = L.append_block context "merge" the_function in
  ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
  L.builder_at_end context merge_bb
and
build_stmt_list the_function v_symbol_tables builder stmt_list =
  List.fold_left (build_stmt the_function v_symbol_tables) builder stmt_list
in

(* function declaration builder *)
let build_func v_symbol_tables class_name builder (sf : sfdecl) =
  let signature = { fs_name = sf.sfname; formal_types =
    List.fold_left (fun s (typ, _) -> s @ [typ]) [] sf.sformals } in
  let func =
    if class_name = "" then
      SignatureMap.find signature user_func_map
    else
      SignatureMap.find signature (StringMap.find class_name class_signature_map)
  in
  let func_builder = L.builder_at_end context (L.entry_block func) in
  (* allocs formals in the stack *)
  let alloca_formal s (typ, name) =
    s @ [{ llvalue = (L.build_alloca (ltype_of_typ typ) name func_builder) ; typ = typ }] in
  let stack_vars =
    if class_name = "" then
      List.fold_left alloca_formal [] sf.sformals
    else
      let class_typ = (A.Class(class_name)) in
      let fst_stmt = { llvalue = (L.build_alloca ((ltype_of_typ class_typ)) "self" func_builder); typ
      fst_stmt::(List.fold_left alloca_formal [] sf.sformals)
  in
  (* stores pointers to the stack location of the formal args *)
  let rec store_formals param stack_p = match param, stack_p with
    [], [] -> []
  | hd1::[], hd2::[] -> [L.build_store hd1 hd2.llvalue func_builder]
  | hd1::tl1, hd2::tl2 -> let fst_stmt = (L.build_store hd1 hd2.llvalue func_builder) in
                          fst_stmt::(store_formals tl1 tl2)
  | _ -> raise (Failure "store_formals array mismatch!") in
  (* add a new elem in this function's v_symbol_tables and add formals *)
  let this_scopes_symbol_table = StringHash.create 10 in
  let v_symbol_tables = this_scopes_symbol_table::v_symbol_tables in
  let _ =
    ignore (store_formals (Array.to_list (L.params func)) stack_vars);
    List.iter (fun elem -> StringHash.add this_scopes_symbol_table
                              (L.value_name elem.llvalue) elem) stack_vars in
  let last_builder = List.fold_left (fun builder stmt ->
         build_stmt func v_symbol_tables builder stmt) func_builder sf.sbody in
 (* if user didn't specify return on void function, then add it ourselves *)
  let _ = if (sf.srtype = A.Primitive(A.Void)) &&
         (not (List.mem SReturnVoid sf.sbody)) then (* TODO does this need to be updated to be more r
         ignore (L.build_ret_void last_builder) in
 builder in

(* class declaration builder *)
let sassign_to_sexpr = function
  SRegularAssign(lhs_typ, _, _) as sra -> (lhs_typ, (SAssign(sra)))
| SStaticAssign(_, lhs_typ, _, _) as ssa -> (lhs_typ, (SAssign(ssa)))
in
let build_class v_symbol_tables builder (sc : sclassdecl) =
  (* First build the struct type in LLVM, this will be important *)
  (* Classes can have other classes as their fields - these are just pointers *)
  (* loop over all the static vars *)
  let _ = List.map (build_expr builder v_symbol_tables) (List.map sassign_to_sexpr sc.sstatic_vars) i
  (* Then loop through all the fdecls, including constructors *)
  let _ = (List.fold_left (build_func v_symbol_tables sc.scname) builder sc.smethods) in
  builder
in

(* LLVM requires a 'main' function as an entry point *)
let main_t : L.lltype =
    L.var_arg_function_type i32_t [| |] in
let main_func : L.llvalue =
  L.define_function "main" main_t the_module in
let main_builder = L.builder_at_end context (L.entry_block main_func) in

(*
let a = ltype_of_typ (A.Array(A.Array(A.Array(A.Primitive(A.Int))))) in
let b = L.build_malloc a "test" main_builder in
*)

(* program builder *)
let build_program v_symbol_tables builder (spunit : sp_unit) = match spunit with
  SStmt(ss)       -> build_stmt main_func v_symbol_tables builder ss
| SFdecl(sf)      -> build_func v_symbol_tables "" builder sf
| SClassdecl(sc)  -> build_class v_symbol_tables builder sc in
```

```
  let final_builder = List.fold_left (build_program [StringHash.create 10]) main_builder sp_units in
  ignore (L.build_ret (L.const_int i32_t 0) final_builder); (* build return for main *)
  the_module
```

9.6 libfuncs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int println(char *s)
{
    printf("%s\n", s);
}

char *int_to_string(int i)
{
    char *str = malloc(11 * sizeof(char)); /* max len for int is 10 */
    snprintf(str, 11 * sizeof(char), "%d", i);
    return str;
}

char *long_to_string(long l)
{
    char *str = malloc(24 * sizeof(char));
    snprintf(str, 24 * sizeof(char), "%li", l);
    return str;
}

char *float_to_string(double f)
{
    char *str = malloc(24 * sizeof(char));
    snprintf(str, 24 * sizeof(char), "%.4f", f);
    return str;
}

char *char_to_string(char c)
{
    char *str = malloc(2 * sizeof(char));
    snprintf(str, 2 * sizeof(char), "%c", c);
    return str;
}

char *bool_to_string(int b)
{
    if (b) return "true"; else return "false";
}

long int_to_long(int i)
{
    return (long) i;
}
```

```
double int_to_float(int i)
{
    return (double) i;
}

// The following function was inspired by the stack ovrflow post
// https://stackoverflow.com/questions/8465006/how-do-i-concatenate-two-strings-in-c
char* concat_strings(const char *s1, const char *s2)
{
    char *result = malloc(strlen(s1) + strlen(s2) + 1);
    strcpy(result, s1);
    strcat(result, s2);
    return result;
}

int compare_strings(const char *s1, const char *s2)
{
    return (strcmp(s1, s2) == 0);
}

void check_int_not_zero(int i, char *message)
{
    if (i == 0)
    {
        fprintf(stderr, "%s\n", message);
        exit(1);
    }
}

void check_long_not_zero(long l, char *message)
{
    if (l == 0)
    {
        fprintf(stderr, "%s\n", message);
        exit(1);
    }
}

void check_float_not_zero(double f, char *message)
{
    if (f == 0.0)
    {
        fprintf(stderr, "%s\n", message);
        exit(1);
    }
}

void check_not_null(void *p)
{
    // Inspired by https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/null-pointer-guards/
    if (p == NULL)
    {
        fprintf(stderr, "NullPointerException\n");
        exit(1);
    }
}
```

9.7 parser.mly

```
%{
open Ast
%}

/* Primitive types */
%token INT LONG FLOAT BOOLEAN CHAR STRING VOID
/* Boolean operators */
%token NOT OR AND
/* Loops and conditionals */
%token LOOP WHILE IF ELIF ELSE
/* Named literals */
%token NULL
/* Words related to functions and classes */
%token DEF CLASS SELF RETURN RETURNS STATIC REQUIRED OPTIONAL
/* Mathematical operators */
%token PLUS MINUS TIMES DIVIDE MODULO
/* Assignment operators */
%token EQ PLUS_EQ MINUS_EQ TIMES_EQ DIVIDE_EQ
/* Comparison operators */
%token DOUBLE_EQ NOT_EQ GT LT GTE LTE
/* Misc. punctuation */
%token LPAREN RPAREN LBRACKET RBRACKET COLON PERIOD COMMA UNDERSCORE
/* Syntactically significant whitespace */
%token NEWLINE INDENT DEDENT EOF
/* Misc. Keywords */
%token DEFAULT
/* Parameterized tokens */
%token <int> INT_LITERAL
%token <int64> LONG_LITERAL
%token <string> FLOAT_LITERAL
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <bool> BOOLEAN_LITERAL
%token <string> CLASS_NAME
%token <string> IDENTIFIER
%token <string> OBJ_OPERATOR
%token <string> OBJ_OPERATOR_METHOD_NAME

/* Set precedence and associativity rules */
/* https://docs.python.org/3/reference/expressions.html#operator-precedence */
%nonassoc DEFAULT
%right EQ PLUS_EQ MINUS_EQ TIMES_EQ DIVIDE_EQ
%left OR
%left AND
%left NOT
%left DOUBLE_EQ NOT_EQ GT LT GTE LTE
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%left OBJ_OPERATOR
```

```
%left PERIOD
%nonassoc UNARY_MINUS
%nonassoc FIELD
%right LBRACKET
%left RBRACKET

%start program /* the entry point */
%type <Ast.program> program

%%

program:
  program_without_eof EOF { List.rev $1 }

program_without_eof:
  program_without_eof stmt { (Stmt $2)::$1 }
| program_without_eof fdecl { (Fdecl $2)::$1 }
| program_without_eof classdecl { (Classdecl $2)::$1 }
| program_without_eof NEWLINE { $1 }
| /* nothing */ { [] }

stmts:
  { [] }
| stmts stmt { $2 :: $1 }

stmt:
  expr NEWLINE { Expr $1 }
| RETURN expr NEWLINE { Return $2 }
| RETURN NEWLINE { ReturnVoid }
| RETURN VOID NEWLINE { ReturnVoid }
| if_stmt  { $1 }
| loop { $1 }

if_stmt:
  IF expr COLON NEWLINE INDENT stmts DEDENT { If ($2, List.rev $6, [], []) }
| IF expr COLON NEWLINE INDENT stmts DEDENT ELSE COLON NEWLINE INDENT stmts DEDENT { If ($2, List.rev $6
| IF expr COLON NEWLINE INDENT stmts DEDENT elif ELSE COLON NEWLINE INDENT stmts DEDENT { If ($2, List.r
| IF expr COLON NEWLINE INDENT stmts DEDENT elif { If ($2, List.rev $6, List.rev $8, []) }

fdecl:
  DEF IDENTIFIER LPAREN type_params RPAREN RETURNS typ COLON NEWLINE INDENT stmts DEDENT { {rtype = $7;
| DEF IDENTIFIER LPAREN type_params RPAREN COLON NEWLINE INDENT stmts DEDENT { {rtype = Primitive Void;
| DEF IDENTIFIER LPAREN RPAREN RETURNS typ COLON NEWLINE INDENT stmts DEDENT { {rtype = $6; fname = $2;
| DEF IDENTIFIER LPAREN RPAREN COLON NEWLINE INDENT stmts DEDENT { {rtype = Primitive Void; fname = $2;
| DEF OBJ_OPERATOR_METHOD_NAME LPAREN typ IDENTIFIER RPAREN RETURNS typ COLON NEWLINE INDENT stmts DEDE
| DEF OBJ_OPERATOR_METHOD_NAME LPAREN typ IDENTIFIER RPAREN COLON NEWLINE INDENT stmts DEDENT { {rtype =

elif:
  ELIF expr COLON NEWLINE INDENT stmts DEDENT { [($2, List.rev $6)] }
| elif ELIF expr COLON NEWLINE INDENT stmts DEDENT { ($3, List.rev $7) :: $1 }

loop:
  LOOP expr WHILE expr COLON NEWLINE INDENT stmts DEDENT { Loop ($2, $4, List.rev $8) }
| LOOP WHILE expr COLON NEWLINE INDENT stmts DEDENT { Loop (NullExpr, $3, List.rev $7) }
```

```
type_params:  /* these are the method signature type */
  typ IDENTIFIER { [($1, $2)] }
| type_params COMMA typ IDENTIFIER { ($3, $4) :: $1 }

params: /* these are the params used to invoke a function */
  expr { [$1] }
| params COMMA expr { $3 :: $1 }

classheader:
  CLASS CLASS_NAME COLON NEWLINE { ($2, []) }
| CLASS CLASS_NAME LBRACKET class_name_list RBRACKET COLON NEWLINE { ($2, (List.rev $4)) }

classdecl:
  classheader
    INDENT STATIC COLON NEWLINE INDENT assigns NEWLINE
    DEDENT REQUIRED COLON NEWLINE INDENT vdecls NEWLINE
    DEDENT OPTIONAL COLON NEWLINE INDENT assigns NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT STATIC COLON NEWLINE INDENT assigns NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT REQUIRED COLON NEWLINE INDENT vdecls NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT OPTIONAL COLON NEWLINE INDENT assigns NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT STATIC COLON NEWLINE INDENT assigns NEWLINE
    DEDENT REQUIRED COLON NEWLINE INDENT vdecls NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT STATIC COLON NEWLINE INDENT assigns NEWLINE
    DEDENT OPTIONAL COLON NEWLINE INDENT assigns NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| classheader
    INDENT REQUIRED COLON NEWLINE INDENT vdecls NEWLINE
    DEDENT OPTIONAL COLON NEWLINE INDENT assigns NEWLINE
    DEDENT optional_fdecls DEDENT { {cname = (fst $1); source_class_name = ""; generics = (snd $1); sta
| CLASS CLASS_NAME EQ CLASS_NAME LPAREN typ_list RPAREN { {cname = $2; source_class_name = $4; generics

optional_fdecls:
  fdecls { $1 }
| /* nothing */ { [] }

fdecls:
  fdecl { [$1] }
| fdecls fdecl { $2::$1 }

vdecls:
  vdecl { [$1] }
```

```
| vdecls NEWLINE vdecl { $3::$1 }

vdecl:
  typ IDENTIFIER { ($1, $2) }

assigns:
  assign { [$1] }
| assigns NEWLINE assign { $3::$1 }

assign:
  typ IDENTIFIER EQ expr { RegularAssign ($1, $2, $4) }

assign_update:
  IDENTIFIER EQ expr { RegularUpdate ($1, Eq, $3) }
| IDENTIFIER PLUS_EQ expr { RegularUpdate ($1, Eq, Binop(Id($1), Plus, $3)) }
| IDENTIFIER MINUS_EQ expr { RegularUpdate ($1, Eq, Binop(Id($1), Subtract, $3)) }
| IDENTIFIER TIMES_EQ expr { RegularUpdate ($1, Eq, Binop(Id($1), Times, $3)) }
| IDENTIFIER DIVIDE_EQ expr { RegularUpdate ($1, Eq, Binop(Id($1), Divide, $3)) }
| object_variable_access EQ expr { ObjectVariableUpdate ($1, Eq, $3) }
| object_variable_access PLUS_EQ expr { ObjectVariableUpdate ($1, Eq, Binop(ObjectVariableAccess($1), Pl
| object_variable_access MINUS_EQ expr { ObjectVariableUpdate ($1, Eq, Binop(ObjectVariableAccess($1), S
| object_variable_access TIMES_EQ expr { ObjectVariableUpdate ($1, Eq, Binop(ObjectVariableAccess($1), T
| object_variable_access DIVIDE_EQ expr { ObjectVariableUpdate ($1, Eq, Binop(ObjectVariableAccess($1),
| array_access EQ expr { ArrayAccessUpdate ($1, Eq, $3) }
| array_access PLUS_EQ expr { ArrayAccessUpdate ($1, Eq, Binop(ArrayAccess($1), Plus, $3)) }
| array_access MINUS_EQ expr { ArrayAccessUpdate ($1, Eq, Binop(ArrayAccess($1), Subtract, $3)) }
| array_access TIMES_EQ expr { ArrayAccessUpdate ($1, Eq, Binop(ArrayAccess($1), Times, $3)) }
| array_access DIVIDE_EQ expr { ArrayAccessUpdate ($1, Eq, Binop(ArrayAccess($1), Divide, $3)) }

func_call:
  expr PERIOD IDENTIFIER LPAREN params RPAREN { MethodCall ($1, $3, List.rev $5) }
| IDENTIFIER LPAREN params RPAREN { FuncCall ($1, List.rev $3) }
| expr PERIOD IDENTIFIER LPAREN RPAREN { MethodCall ($1, $3, []) }
| IDENTIFIER LPAREN RPAREN { FuncCall ($1, []) }
| expr OBJ_OPERATOR expr { MethodCall ($1, "_" ^ $2, [$3]) }

object_instantiation:
  CLASS_NAME LPAREN params RPAREN { ObjectInstantiation ($1, List.rev $3) }
| CLASS_NAME LPAREN RPAREN { ObjectInstantiation ($1, []) }

object_variable_access:
  expr PERIOD IDENTIFIER { { ova_expr = $1; ova_class_name = ""; ova_var_name = $3; ova_is_static = fal
| CLASS_NAME PERIOD IDENTIFIER { { ova_expr = NullExpr; ova_class_name = $1; ova_var_name =  $3; ova_is

array_access:
  expr LBRACKET expr RBRACKET { ($1, $3) }

array_literal:
  LBRACKET params RBRACKET { ArrayLiteral (List.rev $2) }
| LBRACKET RBRACKET { ArrayLiteral ([]) }

non_array_typ:
  INT { Primitive Int }
| LONG { Primitive Long }
| FLOAT { Primitive Float }
| CHAR { Primitive Char }
| STRING { Primitive String }
| BOOLEAN { Primitive Bool }
| VOID { Primitive Void }
| CLASS_NAME { Class $1 }

typ:
  non_array_typ { $1 }
| typ LBRACKET RBRACKET { Array ($1) }

array_default:
  non_array_typ LBRACKET expr RBRACKET { (Array ($1), [$3]) }
| array_default LBRACKET expr RBRACKET { (Array (fst $1), $3::(snd $1)) }

typ_list:
  typ { [$1] }
| typ_list COMMA typ { $3::$1 }

class_name_list:
  CLASS_NAME { [Class $1] }
| class_name_list COMMA CLASS_NAME { (Class $3)::$1 }

expr:
  INT_LITERAL { IntLiteral $1 }
| LONG_LITERAL { LongLiteral $1 }
| FLOAT_LITERAL { FloatLiteral $1 }
| CHAR_LITERAL { CharLiteral $1 }
| STRING_LITERAL { StringLiteral $1 }
| BOOLEAN_LITERAL { BoolLiteral $1 }
| IDENTIFIER { Id $1 }
| SELF { Self }
| NULL { NullExpr }
| func_call { Call $1 }
| object_instantiation { $1 }
| object_variable_access { ObjectVariableAccess $1 }
| array_access { ArrayAccess $1 }
| array_literal { $1 }
| DEFAULT array_default { DefaultArray ((fst $2), (snd $2)) }
| LPAREN expr RPAREN { $2 }
| expr PLUS expr { Binop ($1, Plus, $3) }
| expr MINUS expr { Binop ($1, Subtract, $3) }
| expr TIMES expr { Binop ($1, Times, $3) }
| expr DIVIDE expr { Binop ($1, Divide, $3) }
| expr MODULO expr { Binop ($1, Modulo, $3) }
| MINUS expr %prec UNARY_MINUS { Unop (Neg, $2) }
| assign { Assign $1 }
| assign_update { Update $1 }
| expr DOUBLE_EQ expr { Binop ($1, DoubleEq, $3) }
| expr NOT_EQ expr { Unop (Not, Binop ($1, DoubleEq, $3)) }
| expr GT expr { Binop ($1, BoGT, $3) }
| expr LT expr { Binop ($1, BoLT, $3) }
| expr GTE expr { Binop ($1, BoGTE, $3) }
| expr LTE expr { Binop ($1, BoLTE, $3) }
| NOT expr { Unop (Not, $2) }
| expr OR expr { Binop ($1, BoOr, $3) }
| expr AND expr { Binop ($1, BoAnd, $3) }
```

9.8 repl.ml

```
(* How to use this primitive REPL to test:

   Set export OCAMLRUNPARAM='p' in your shell.
   This will show interesting diagnostic info from the shift/reduce
   tables generated by the parser. (See chapter 4 of the dragon
   book for info on how the parser generated by YACC works.)

   Then, run ./repl and type different programs to see how
   they got tokenized. If you enter a valid expression, the
   program should end with "Passed" when you hit ctrl-D. If you
   enter an invalid program, it will give you a parse error.
*)
open Sast

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.read_next_token lexbuf in
  let sast = Semant.check ast in
  print_endline ("Passed\n" ^ (graphviz_string_of_sprogram sast))
```

9.9 sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr = typ * sx
and sx =
  SIntLiteral of int
| SLongLiteral of int64
| SFloatLiteral of string
| SCharLiteral of char
| SStringLiteral of string
| SBoolLiteral of bool
| SId of string
| SSelf
| SNullExpr
| SCall of scall
| SObjectInstantiation of string * sexpr list
| SObjectVariableAccess of sobject_variable_access
| SArrayAccess of sarray_access
| SArrayLiteral of sexpr list
| SDefaultArray of typ * sexpr list
| SBinop of sexpr * binop * sexpr
| SUnop of unaryop * sexpr
| SAssign of sassign
| SUpdate of supdate
and sarray_access = sexpr * sexpr
and scall =
  SFuncCall of string * sexpr list
| SMethodCall of sexpr * string * sexpr list
and sassign =
  SRegularAssign of typ * string * sexpr
(* These look similar, but will be treated differently in codegen.
   The first string is the class name, the second is the var name.
   This is only used when setting the initial value inside a class.
   Things of the form MyClass.x = foo are SUpdates. *)
| SStaticAssign of string * typ * string * sexpr
and supdate =
  SRegularUpdate of string * updateop * sexpr
| SObjectVariableUpdate of sobject_variable_access * updateop * sexpr
| SArrayAccessUpdate of sarray_access * updateop * sexpr
and sobject_variable_access = {
  sova_sexpr: sexpr;
  sova_class_name: string;
  sova_var_name: string;
  sova_is_static: bool;
}

type sstmt =
  SExpr of sexpr
| SReturn of sexpr
| SReturnVoid
| SIf of sexpr * sstmt list * selif list * sstmt list
| SLoop of sexpr * sexpr * sstmt list
and selif = sexpr * sstmt list

type sfdecl = {
  srtype: typ;
  sfname: string;
  sformals: bind list;
  sbody: sstmt list;
}

type sclassdecl = {
  scname: string;
  sstatic_vars: sassign list;
  srequired_vars: bind list;
  soptional_vars: sassign list;
  smethods: sfdecl list;
}

type sp_unit =
  SStmt of sstmt
| SFdecl of sfdecl
| SClassdecl of sclassdecl

type sprogram = sp_unit list

type function_signature = {
  fs_name: string;
  formal_types: typ list;
```

```
}

(* Begin visualization functions *)
let rec str_of_typ = function
  Primitive(Int) -> "int"
| Primitive(Long) -> "long"
| Primitive(Float) -> "float"
| Primitive(Char) -> "char"
| Primitive(String) -> "string"
| Primitive(Bool) -> "boolean"
| Primitive(Void) -> "void"
| Class(str) -> "Class_" ^ str ^ "_"
| Array(typ) -> "Array_" ^ (str_of_typ typ) ^ "_"
| NullType -> "NULL"

let get_label_with_type suffixed_name unsuffixed_name typ = suffixed_name ^ " [label=\"" ^ unsuffixed_na

let get_multi_node_generator_typ node_name suffix subconverter input_list typ =
  let start_node = node_name ^ suffix in
  let subgraphs = (List.mapi (subconverter suffix) input_list) in (* List<Tuple<StartNodeString, List<S
  (start_node, (get_label_with_type start_node node_name typ)::List.concat (List.map (get_combine_functi

let combine_list_typ node_name suffix input_list typ =
  let start_node = node_name ^ suffix in
  (start_node, (get_label_with_type start_node node_name typ)::List.concat (List.map (get_combine_functi

let string_of_id_typ existing_suffix new_index id_string typ =
  let suffix = new_suffix existing_suffix new_index in
  ("id" ^ suffix, ["id" ^ suffix ^ " [label=\"id: " ^ id_string ^ " (" ^ (str_of_typ typ) ^ ")\" fontco

let rec string_of_sexpr existing_suffix new_index sexpr =
  let suffix = new_suffix existing_suffix new_index in
  let typ = (fst sexpr) in
  let sx = (snd sexpr) in
  match sx with
  SIntLiteral(integer) -> ("intlit" ^ suffix, [get_literal_node "intlit" suffix (string_of_int integer)
| SLongLiteral(long) -> ("longlit" ^ suffix, [get_literal_node "longlit" suffix (Int64.to_string long)]
| SFloatLiteral(f) -> ("floatlit" ^ suffix, [get_literal_node "floatlit" suffix f])
| SCharLiteral(c) -> ("charlit" ^ suffix, [get_literal_node "charlit" suffix (String.make 1 c)])
| SStringLiteral(s) -> ("stringlit" ^ suffix, [get_literal_node "stringlit" suffix s])
| SBoolLiteral(b) -> ("boollit" ^ suffix, [get_literal_node "boollit" suffix (string_of_bool b)])
| SId(id_string) -> string_of_id_typ suffix 0 id_string typ
| SNullExpr -> ("nullexpr" ^ suffix, [get_literal_node "nullexpr" suffix "NULL"])
| SSelf -> string_of_id_typ suffix 0 "self" typ
| SCall(scall) -> string_of_scall typ suffix 0 scall
| SObjectInstantiation(id_string, sexprs) -> combine_list_typ "object_instantiation" suffix ([string_of_
| SObjectVariableAccess(sobject_variable_access) -> string_of_sobject_variable_access suffix 0 sobject_v
| SArrayAccess(sarray_access) -> string_of_sarray_access suffix 0 sarray_access typ
| SArrayLiteral(sexprs) -> get_multi_node_generator_typ "array_literal" suffix string_of_sexpr sexprs ty
| SDefaultArray(typ, sexprs) -> combine_list_typ "default_array" suffix ([string_of_typ suffix 0 typ] @
| SBinop(sexpr1, binop, sexpr2) -> combine_list_typ "binop" suffix ([string_of_sexpr suffix 0 sexpr1] @
| SUnop(unaryop, sexpr) -> combine_list_typ "unaryop" suffix ([string_of_unaryop suffix 0 unaryop] @ [s
| SAssign(sassign) -> string_of_sassign suffix 0 sassign
| SUpdate(supdate) -> string_of_supdate typ suffix 0 supdate
```

```
and string_of_sarray_access existing_suffix new_index sarray_access typ =
  let suffix = new_suffix existing_suffix new_index in
  let sexpr1 = (fst sarray_access) in
  let sexpr2 = (snd sarray_access) in
  combine_list_typ "array_access" suffix ([string_of_sexpr suffix 0 sexpr1] @ [string_of_sexpr suffix 1
and string_of_scall typ existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  SFuncCall(id_string, sexprs) -> combine_list_typ "func_call" suffix ([string_of_id suffix 0 id_string
| SMethodCall(sexpr1, id2, sexprs) -> combine_list_typ "method_call" suffix ([string_of_sexpr suffix 0 s
and string_of_sassign existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  SRegularAssign(typ, id_string, sexpr) -> combine_list_typ "assign" suffix ([string_of_typ suffix 0 typ
| SStaticAssign(_, typ, id_string, sexpr) -> combine_list_typ "assign" suffix ([string_of_typ suffix 0 t
and string_of_supdate typ existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  SRegularUpdate(id_string, updateop, sexpr) -> combine_list_typ "update" suffix ([string_of_id_typ suf
| SObjectVariableUpdate(sobject_variable_access, updateop, sexpr) -> combine_list_typ "update" suffix (
| SArrayAccessUpdate(sarray_access, updateop, sexpr) -> combine_list_typ "update" suffix ([string_of_sa
and string_of_sobject_variable_access existing_suffix new_index = function
  { sova_class_name = class_name; sova_var_name = var_name; sova_is_static = true; _ } ->
    let suffix = new_suffix existing_suffix new_index in
    ("static_var_access" ^ suffix, ["static_var_access" ^ suffix ^ " [label=\"" ^ (class_name) ^ "." ^
| { sova_sexpr = sexpr; sova_var_name = var_name; sova_is_static = false; _ } ->
    let suffix = new_suffix existing_suffix new_index in
    combine_list "obj_var_access" suffix ([string_of_sexpr suffix 0 sexpr] @ [string_of_id suffix 1 var

let rec string_of_sstmt existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function
  SExpr(sexpr) -> get_single_node_generator "expr" suffix string_of_sexpr sexpr
| SReturn(sexpr) -> get_single_node_generator "return" suffix string_of_sexpr sexpr
| SReturnVoid -> combine_list "return" suffix [("returnvoid" ^ suffix, [get_literal_node "returnvoid" su
| SIf(sexpr, sl1, selifs, sl2) -> combine_list "if" suffix ([string_of_sexpr suffix 0 sexpr] @ (mapiplus
| SLoop(sexpr1, sexpr2, sl1) -> combine_list "loop" suffix ([string_of_sexpr suffix 0 sexpr1] @ [string
and
string_of_selif existing_suffix new_index selif_tuple =
 let suffix = new_suffix existing_suffix new_index in
 combine_list "elif" suffix ([string_of_sexpr suffix 0 (fst selif_tuple)] @ (mapiplus 1 (string_of_sstmt

let string_of_sfdecl existing_suffix new_index sfdecl =
  let suffix = new_suffix existing_suffix new_index in
  combine_list "fdecl" suffix ([string_of_id suffix 0 sfdecl.sfname] @ (mapiplus 1 (string_of_bind suff

let string_of_sclassdecl existing_suffix new_index sclassdecl =
  let suffix = new_suffix existing_suffix new_index in
  combine_list "classdecl" suffix ([string_of_id suffix 0 sclassdecl.scname] @ (mapiplus 1 (string_of_s

let string_of_sp_unit existing_suffix new_index =
  let suffix = new_suffix existing_suffix new_index in
  function (* Takes a program unit and returns a Tuple<StartNodeString, List<String>> *)
  SStmt(sstmt) -> get_single_node_generator "stmt" suffix string_of_sstmt sstmt
```

```
| SFdecl(sfdecl) -> string_of_sfdecl suffix 0 sfdecl
| SClassdecl(sclassdecl) -> string_of_sclassdecl suffix 0 sclassdecl

let string_of_sprogram sprogram = (* Takes a program object and returns a Tuple<StartNodeString, List<S
  let suffix = "0" in
  get_multi_node_generator "program" suffix string_of_sp_unit sprogram

let graphviz_string_of_sprogram sprogram =
  "digraph G { \n" ^ (String.concat "\n" (snd (string_of_sprogram sprogram))) ^ "\n}"
```

9.10 scanner.mll

```
(* Scanner for the Boomslang Language *)

{

open Parser

module StringMap = Map.Make(String)

let add_entry map pair = StringMap.add (fst pair) (snd pair) map

let reserved_word_to_token = List.fold_left add_entry StringMap.empty [
  (* Boolean operators *)
  ("not", NOT); ("or", OR); ("and", AND);
  (* Loops and conditionals *)
  ("loop", LOOP); ("while", WHILE); ("if", IF); ("elif", ELIF); ("else", ELSE);
  (* Words related to functions and classes *)
  ("def", DEF); ("class", CLASS); ("self", SELF);
  ("return", RETURN); ("returns", RETURNS);
  ("static", STATIC); ("required", REQUIRED); ("optional", OPTIONAL);
  (* Primitive data types *)
  ("int", INT); ("long", LONG); ("float", FLOAT); ("boolean", BOOLEAN);
  ("char", CHAR); ("string", STRING); ("void", VOID);
  (* Default keyword for intitializing arrays *)
  ("default", DEFAULT);
]

let llvm_illegal_chars = [
  ("%", "pct"); ("&", "amp"); ("\\$", "dol"); ("@", "at"); ("!", "excl");
  ("#", "pound"); ("\\^", "caret"); ("\\*", "star"); ("/", "slash");
  ("~", "tilde"); ("\\?", "qstn"); (">", "gt"); ("<", "lt"); (":", "col");
  ("=", "eq");
]

let replace input_str illegal_char = Str.global_replace (Str.regexp (fst illegal_char)) (snd illegal_cha
let replace_illegal_chars str = List.fold_left replace str llvm_illegal_chars

let convert_slashes str =
  let str = Str.matched_string str in
  let orig_len = String.length str in
  let new_len = orig_len / 2 in
  String.sub str 0 new_len
let strip_firstlast str =
  if String.length str <= 2 then ""
  else String.sub str 1 ((String.length str) - 2)

(* In ocaml 4.08+ you could write let tab_count_stack = Stack.of_seq (List.to_seq [0]) *)
let tab_count_stack = Stack.create ()
let add_zero_to_stack = (Stack.push 0 tab_count_stack); ()
let token_queue = Queue.create ()

let rec enqueue_dedents n = if n > 0 then (Queue.add DEDENT token_queue; (enqueue_dedents (n-1)))

let rec enqueue_indents n = if n > 0 then (Queue.add INDENT token_queue; (enqueue_indents (n-1)))

let count_tabs str = if String.contains str '\t' then String.length str - String.index str '\t' else 0
}


(* Class names in Boomslang must start with a capital letter,
   to distinguish them from identifiers, which must begin
   with a lowercase letter *)
let class_name = ['A'-'Z']['a'-'z' 'A'-'Z']*
let int_literal = ['0'-'9']+

rule tokenize = parse
  [' ' '\r'] { tokenize lexbuf }
(* Mathematical operations *)
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MODULO }
(* Assignment operators *)
| '=' { EQ }
| "+=" { PLUS_EQ }
| "-=" { MINUS_EQ }
| "*=" { TIMES_EQ }
| "/=" { DIVIDE_EQ }
(* Comparison operators *)
| "==" { DOUBLE_EQ }
| "!=" { NOT_EQ }
| ">" { GT }
| "<" { LT }
| ">=" { GTE }
| "<=" { LTE }
(* Multi-line comments *)
| ['\n']+[' ' '\t']*"/#" { multi_comment lexbuf }
| "/#" { multi_comment lexbuf }
(* Misc. punctuation *)
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRACKET }
| ']' { RBRACKET }
| ':' { COLON }
| '.' { PERIOD }
| ',' { COMMA }
| '_' { UNDERSCORE }
| "NULL" { NULL }
(* Literal definitions *)
| int_literal as lit { INT_LITERAL(int_of_string lit) }
| int_literal"L" as lit {
    LONG_LITERAL(Int64.of_string (String.sub lit 0 (String.length lit - 1)))
}
| ['0'-'9']+('.'['0'-'9']+)? | '.'['0'-'9']+ as lit { FLOAT_LITERAL(lit) }
| "true" { BOOLEAN_LITERAL(true) }
| "false" { BOOLEAN_LITERAL(false) }
(* Char literals are single quotes followed by any single character
   followed by a single quote *)
| '\'' [' '-'~'] '\'' as lit { CHAR_LITERAL( (strip_firstlast lit).[0] ) }
(* String literals in Boomslang cannot contain double quotes or newlines.
   String literals are a " followed by any non newline or double quote
   followed by " regex copied from CORAL*)
|  '"' [^'"''\\']* ('\\'_[^'"''\\']* )* '"' as lit { let stripped = (strip_firstlast lit) in let fix_sla
(* Syntactically meaningful whitespace - tabs for indentation only *)
(* Either a single-line comment appears on a line by itself, in which case
   we ignore that line completely, or else it appears at the end of the line,
   in which case we ignore everything after the # before the \n *)
| (['\n']+[' ' '\t']*('#'[^'\n']*))* { tokenize lexbuf }
| ('#'[^'\n']*)?(['\n']+['\t']* as newlines_and_tabs) {
  let num_tabs = (count_tabs newlines_and_tabs) in
  if (Stack.top tab_count_stack) == num_tabs then
    NEWLINE
  else if (Stack.top tab_count_stack) > num_tabs then
    ((enqueue_dedents ((Stack.pop tab_count_stack) - num_tabs); Stack.push num_tabs tab_count_stack); NE
  else
    ((enqueue_indents (num_tabs - (Stack.top tab_count_stack)); Stack.push num_tabs tab_count_stack); NE
}
(* User defined types, i.e. class names *)
| class_name as t { CLASS_NAME(t) }
(* If we see a lowercase letter followed by any letters or digits,
   it could either be the name of a primitive type (e.g. int), or
   a reserved word (e.g. class) or an identifier for a variable. *)
| ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as possible_id {
    if StringMap.mem possible_id reserved_word_to_token
      then StringMap.find possible_id reserved_word_to_token
    else
      IDENTIFIER(possible_id)
  }
| ['+' '-' '%' '&' '$' '@' '!' '#' '^' '*' '/' '~' '?' '>' '<' ':' '=']+ as lit {
  (* convert the weird chars to simpler strings so avoid any LLVM errors later on. *)
  OBJ_OPERATOR((replace_illegal_chars lit))
}
| '_'['+' '-' '%' '&' '$' '@' '!' '#' '^' '*' '/' '~' '?' '>' '<' ':' '=']+ as lit {
  (* convert the weird chars to simpler strings so avoid any LLVM errors later on. *)
  OBJ_OPERATOR_METHOD_NAME((replace_illegal_chars lit))
}
(* Automatically add a NEWLINE to end of all files.
   All statements in Boomslang must end in a NEWLINE, such that
   ordinarily all valid programs must have a blank line at the end.
   But since this is easy to forget, we automatically add a blank line
   here in case the user forgets. *)
| eof { (Queue.add EOF token_queue); NEWLINE }
| _ as c { raise (Failure("Illegal character: " ^ Char.escaped c)) }


and multi_comment = parse
  "#/" { tokenize lexbuf }
| _ { multi_comment lexbuf }

{
let read_next_token lexbuf =
  if Queue.is_empty token_queue then tokenize lexbuf else Queue.take token_queue
}
```

9.11 semant.ml

```
(* Semantic checking for the Boomslang compiler *)

open Ast
open Sast

type lhsrhs = {
  lhs: typ;
  rhs: typ;
}

module StringMap = Map.Make(String);;
module SignatureMap = Map.Make(struct type t = function_signature let compare = compare end);;
module TypMap = Map.Make(struct type t = typ let compare = compare end);;
module LhsRhsMap = Map.Make(struct type t = lhsrhs let compare = compare end);;
module StringHash = Hashtbl.Make(struct
  type t = string (* type of keys *)
  let equal x y = x = y (* use structural comparison *)
  let hash = Hashtbl.hash (* generic hash function *)
end);;

let built_in_funcs = [
  ({ fs_name = "println"; formal_types = [Primitive(String)] }, Primitive(Void));
  ({ fs_name = "int_to_string"; formal_types = [Primitive(Int)] }, Primitive(String));
  ({ fs_name = "long_to_string"; formal_types = [Primitive(Long)] }, Primitive(String));
  ({ fs_name = "float_to_string"; formal_types = [Primitive(Float)] }, Primitive(String));
  ({ fs_name = "char_to_string"; formal_types = [Primitive(Char)] }, Primitive(String));
  ({ fs_name = "bool_to_string"; formal_types = [Primitive(Bool)] }, Primitive(String));
  ({ fs_name = "int_to_long"; formal_types = [Primitive(Int)] }, Primitive(Long));
  ({ fs_name = "int_to_float"; formal_types = [Primitive(Int)] }, Primitive(Float));
  ({ fs_name = "concat_strings"; formal_types = [Primitive(String); Primitive(String)] }, Primitive(Str
  ({ fs_name = "compare_strings"; formal_types = [Primitive(String); Primitive(String)] }, Primitive(Bo
  ({ fs_name = "check_int_not_zero"; formal_types = [Primitive(Int); Primitive(String)] }, Primitive(Vo
  ({ fs_name = "check_long_not_zero"; formal_types = [Primitive(Long); Primitive(String)] }, Primitive(
  ({ fs_name = "check_float_not_zero"; formal_types = [Primitive(Float); Primitive(String)] }, Primitiv
  ({ fs_name = "check_not_null"; formal_types = [Primitive(Void)] }, Primitive(Void) );
]
```

```
let type_is_nullable = function
  Primitive(_) -> false
| Class(_) -> true
| Array(_) -> false
| NullType -> true

let rec arrays_are_compatible arr1 arr2 = (match arr1 with
  Array(Array(_) as inner1) -> (match arr2 with
      Array(Array(_) as inner2) -> arrays_are_compatible inner1 inner2
    | _ -> false)
| Array(NullType) -> true
| Array(_) -> (arr2 = Array(NullType))
| _ -> false)

let binop_method_name = function
  Plus  -> "_+"
| Subtract -> "_-"
| Times -> "_star"
| Divide -> "_slash"
| Modulo -> "_pct"
| DoubleEq -> "_eqeq"
| BoGT -> "_gt"
| BoLT -> "_lt"
| BoGTE -> "_gteq"
| BoLTE -> "_lteq"
| _ -> raise (Failure("Attempted to use an incompatible binary operator on an object type."))

let are_types_compatible typ1 typ2 =
  ((typ1 = typ2) || (type_is_nullable(typ1) && typ2 = NullType) || (arrays_are_compatible typ1 typ2))

let signature_could_match signature1 signature2 _ =
  if (signature1.fs_name = signature2.fs_name) && ((List.length signature1.formal_types) = (List.length
    List.for_all2 are_types_compatible (signature2.formal_types) (signature1.formal_types)
  else false

(* This function is complicated because if the language has nulls, you can't just look up the
   signature directly. However, since null could match multiple types, if a user calls a func like
   myfunc(null, null), it is impossible to tell if they meant myfunc(MyObject a, MyObject b) or
   myfunc(OtherObject a, OtherObject b). *)
let find_matching_signature signature signatures =
  let matching_signatures_map = SignatureMap.filter (signature_could_match signature) signatures in
  let matching_signatures_list = List.map (fst) (SignatureMap.bindings matching_signatures_map) in
  if List.length matching_signatures_list = 0 then
    raise (Failure("No matching signature found for function call " ^ signature.fs_name))
  else if List.length matching_signatures_list > 1 then
    raise (Failure("The call to " ^ signature.fs_name ^ " is ambiguous."))
  else List.hd matching_signatures_list

(* If the user is calling a function with NULLs as parameters, we need to
   convert the type associated with the NullExpr to the type of the formal
   that is expected. This will help LLVM generate the right kind of null
   pointer. *)
let convert_nulls_in_checked_exprs checked_exprs matching_signature =
  let convert checked_expr typ = match (fst checked_expr) with
```

```
    NullType -> (typ, (snd checked_expr))
  | Array(_) -> (typ, (snd checked_expr))
  | _ -> checked_expr in
  List.map2 convert checked_exprs matching_signature.formal_types

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong. *)

(* Add built-in functions *)
let check original_program =

let rec type_of_identifier v_symbol_tables s =
  match v_symbol_tables with
  [] -> raise (Failure ("undeclared identifier " ^ s))
  | hd::tl -> try StringHash.find hd s
              with Not_found -> (type_of_identifier tl s)
in

let rec dups kind = function (* Stolen from microc *)
      [] -> ()
    | (n1 :: n2 :: _) when n1 = n2 ->
       raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
    | _ :: t -> dups kind t
in

let check_type_is_int = function
  Primitive(Int) -> ()
| _ -> raise (Failure("Expected expr of type int"))
in
let check_type_is_bool = function
  Primitive(Bool) -> ()
| _ -> raise (Failure("Expected expr of type bool"))
in

let add_to_hash hash bind = StringHash.add hash (snd bind) (fst bind) in
let get_hash_of_binds bind_list =
  let hash = (StringHash.create (List.length bind_list)) in
  (List.iter (add_to_hash hash) bind_list); hash
in

let add_built_in map built_in = SignatureMap.add (fst built_in) (snd built_in) map
in
let built_in_func_map = List.fold_left add_built_in SignatureMap.empty built_in_funcs
in

(* Functions to coerce one type into another via a built-in function *)
let wrap_to_string checked_exprs = match checked_exprs with
  [(Primitive(Int), _)] -> (Primitive(String), SCall(SFuncCall("int_to_string", checked_exprs)))
| [(Primitive(Long), _)] -> (Primitive(String), SCall(SFuncCall("long_to_string", checked_exprs)))
| [(Primitive(Float), _)] -> (Primitive(String), SCall(SFuncCall("float_to_string", checked_exprs)))
| [(Primitive(Char), _)] -> (Primitive(String), SCall(SFuncCall("char_to_string", checked_exprs)))
| [(Primitive(Bool), _)] -> (Primitive(String), SCall(SFuncCall("bool_to_string", checked_exprs)))
| [(Primitive(String), _) as sexpr] -> sexpr
| [(_, SNullExpr)] -> (Primitive(String), SCall(SFuncCall("null_to_string", checked_exprs)))
| [(Class(_), _) as sexpr] -> (Primitive(String), SCall(SMethodCall(sexpr, "to_string", [])))
| [(Array(_), _)] -> (Primitive(String), SStringLiteral("Array"))
| _ -> raise (Failure("Expected exactly 1 expression of a non-void primitive, null, or named object typ
in
let wrap_int_to_long checked_exprs = match checked_exprs with
  [(Primitive(Int), _)] -> (Primitive(Long), SCall(SFuncCall("int_to_long", checked_exprs)))
| _ -> raise (Failure("Expected exactly 1 expression of type int."))
in
let wrap_int_to_float checked_exprs = match checked_exprs with
  [(Primitive(Int), _)] -> (Primitive(Float), SCall(SFuncCall("int_to_float", checked_exprs)))
| _ -> raise (Failure("Expected exactly 1 expression of type int."))
in
let coerceable_types = [
  ({ lhs = Primitive(Long); rhs = Primitive(Int) }, wrap_int_to_long);
  ({ lhs = Primitive(Float); rhs = Primitive(Int) }, wrap_int_to_float);
] in
let add_to_map map coerceable_type = LhsRhsMap.add (fst coerceable_type) (snd coerceable_type) map in
let coerceable_types_map = List.fold_left add_to_map LhsRhsMap.empty coerceable_types
in

(* First, figure out all the defined functions *)
let get_signature fdecl = { fs_name = fdecl.fname; formal_types = List.map fst fdecl.formals }
in
let sget_signature sfdecl = { fs_name = sfdecl.sfname; formal_types = List.map fst sfdecl.sformals }
in
let add_fdecl map fdecl =
  let signature = (get_signature fdecl) in
  if SignatureMap.mem signature map then raise (Failure(("Duplicate function signatures detected for " 
  else SignatureMap.add signature fdecl.rtype map
in
let add_signature map = function
  Fdecl(fdecl) -> add_fdecl map fdecl
| _ -> map
in
let function_signatures = List.fold_left add_signature built_in_func_map original_program
in

(* For each class, generate 1-2 constructor signatures, corresponding to the required and optional fiel
let get_bind_from_assign = function
  RegularAssign(typ, name, _) -> (typ, name)
in
let get_required_only_signature classdecl =
  { fs_name = "construct"; formal_types = (List.map (fst) classdecl.required_vars) }
in
let get_required_and_optional_signature classdecl =
  let optional_var_binds = (List.map get_bind_from_assign classdecl.optional_vars) in
  { fs_name = "construct"; formal_types = (List.map (fst) (classdecl.required_vars @ optional_var_binds
in
let get_autogenerated_constructor_signatures classdecl =
  if List.length classdecl.required_vars = 0 && List.length classdecl.optional_vars = 0 then [ {fs_name
  else if List.length classdecl.required_vars > 0 && List.length classdecl.optional_vars = 0 then [ get
  else [(get_required_only_signature classdecl); (get_required_and_optional_signature classdecl)]
in
```

```
let _ = (* Check for duplicate class names *)
let add_classdecl map = function
  Classdecl(classdecl) ->
    if (StringMap.mem classdecl.cname map) then
      raise (Failure("Detected duplicate declarations for class " ^ classdecl.cname))
    else
      StringMap.add classdecl.cname classdecl map
| _ -> map
in
List.fold_left add_classdecl StringMap.empty original_program
in

let get_generic_to_actual actual_class generic_class =
  let generics_length = (List.length generic_class.generics) in
  let actuals_length = (List.length actual_class.generics) in
  if generics_length <> (actuals_length) then
    raise (Failure("Attempted to initialize " ^ generic_class.cname ^ " as " ^ actual_class.cname ^ " bu
  else
    let add_to_map map typ1 typ2 =
      if TypMap.mem typ1 map then (* Duplicate type name found *)
        raise (Failure("Found duplicate generic type " ^ (str_of_typ typ1) ^ " in class " ^ generic_cla
      else TypMap.add typ1 typ2 map in
    List.fold_left2 add_to_map (TypMap.add (Class(generic_class.cname)) (Class(actual_class.cname)) TypM
in
let generic_map =
let add_classdecl map = function
  Classdecl(classdecl) ->
    if ((List.length classdecl.generics) > 0) && classdecl.source_class_name = "" then
      StringMap.add classdecl.cname classdecl map
    else map
| _ -> map
in
List.fold_left add_classdecl StringMap.empty original_program
in
(* This method finds all class declarations of the form
   class MyClass = MyGenericClass(typ1, typ2) and converts
   them to full class declarations. This means looking up the generic
   type class MyGenericClass[T1, T2] and replacing all instances of T1
   with typ1 and all instances of T2 with typ2.
   This takes the entire program (list of p_units) and returns a new
   program that has all the generic classes removed, and all the
   generic class instantiations converted into fully usable classdecls.
   After this point, the original program should no longer be used.
*)
let rec convert_generic_typ generic_to_actual = function
  Primitive(_) as self -> self
| Class(_) as self -> if TypMap.mem self generic_to_actual then
                        TypMap.find self generic_to_actual
                      else self
| Array(typ) -> Array(convert_generic_typ generic_to_actual typ)
| NullType -> NullType
and
convert_generic_assign generic_to_actual = function
  RegularAssign(typ, id, expr) -> RegularAssign((convert_generic_typ generic_to_actual typ), id, (conve
and
convert_generic_bind generic_to_actual bind =
  let typ = (fst bind) in
  let name = (snd bind) in
  ((convert_generic_typ generic_to_actual typ), name)
and
convert_generic_ova generic_to_actual ova =
  let ce expr = convert_generic_expr generic_to_actual expr in
  if ova.ova_class_name <> "" then
    let new_class = (convert_generic_typ generic_to_actual (Class(ova.ova_class_name))) in
    (match new_class with
       Class(new_name) -> { ova_expr = (ce ova.ova_expr); ova_class_name = new_name; ova_var_name = ova
     | _ -> raise (Failure("Somehow converted class type to something that was not a class - this shoul
  else
    { ova_expr = (ce ova.ova_expr); ova_class_name = ova.ova_class_name; ova_var_name = ova.ova_var_nam
and
convert_generic_expr generic_to_actual =
  let ce expr = convert_generic_expr generic_to_actual expr in
  let ces exprs = List.map (convert_generic_expr generic_to_actual) exprs in
  function
  Call(FuncCall(name, exprs)) -> Call(FuncCall(name, (ces exprs)))
| Call(MethodCall(expr1, name, exprs)) -> Call(MethodCall((ce expr1), name, (ces exprs)))
| ObjectInstantiation(old_name, exprs) ->
  let new_class = (convert_generic_typ generic_to_actual (Class(old_name))) in
  (match new_class with
     Class(new_name) -> ObjectInstantiation(new_name, (ces exprs))
   | _ -> raise (Failure("Somehow converted class type to something that was not a class - this should 
| ObjectVariableAccess(ova) -> ObjectVariableAccess((convert_generic_ova generic_to_actual ova))
| ArrayAccess(expr1, expr2) -> ArrayAccess((ce expr1), (ce expr2))
| ArrayLiteral(exprs) -> ArrayLiteral((ces exprs))
| DefaultArray(typ, exprs) -> DefaultArray((convert_generic_typ generic_to_actual typ), (ces exprs))
| Binop(expr1, binop, expr2) -> Binop((ce expr1), binop, (ce expr2))
| Unop(unaryop, expr) -> Unop(unaryop, (ce expr))
| Assign(assign) -> Assign(convert_generic_assign generic_to_actual assign)
| Update(RegularUpdate(name, updateop, expr)) -> Update(RegularUpdate(name, updateop, (ce expr)))
| Update(ObjectVariableUpdate(ova, updateop, expr)) -> Update(ObjectVariableUpdate((convert_generic_ova
| Update(ArrayAccessUpdate((expr1, expr2), updateop, expr3)) -> Update(ArrayAccessUpdate(((ce expr1), (
| IntLiteral(_)
| LongLiteral(_)
| FloatLiteral(_)
| CharLiteral(_)
| StringLiteral(_)
| BoolLiteral(_)
| Id(_)
| Self
| NullExpr as self -> self
and
convert_generic_elif generic_to_actual elif =
  let expr1 = (fst elif) in
  let stmts = (snd elif) in
  ((convert_generic_expr generic_to_actual expr1), (List.map (convert_generic_stmt generic_to_actual) s
and
convert_generic_stmt generic_to_actual =
  let ce expr = convert_generic_expr generic_to_actual expr in
  let css stmts = (List.map (convert_generic_stmt generic_to_actual) stmts) in
  function
  Expr(expr) -> Expr((convert_generic_expr generic_to_actual expr))
| Return(expr) -> Return((convert_generic_expr generic_to_actual expr))
| ReturnVoid -> ReturnVoid
| If(expr1, stmts1, elifs, stmts2) -> If((ce expr1), (css stmts1), (List.map (convert_generic_elif gene
| Loop(expr1, expr2, stmts) -> Loop((ce expr1), (ce expr2), (css stmts))
and
convert_generic_fdecl generic_to_actual fdecl =
  { rtype = (convert_generic_typ generic_to_actual fdecl.rtype);
    fname = fdecl.fname;
    formals = (List.map (convert_generic_bind generic_to_actual) fdecl.formals);
    body = (List.map (convert_generic_stmt generic_to_actual) fdecl.body)
  }
and
convert_instantiation actual_class generic_class =
  let generic_to_actual = get_generic_to_actual actual_class generic_class in
  { cname = actual_class.cname; source_class_name = ""; generics = [];
    static_vars = (List.map (convert_generic_assign generic_to_actual) generic_class.static_vars);
    required_vars = (List.map (convert_generic_bind generic_to_actual) generic_class.required_vars);
    optional_vars = (List.map (convert_generic_assign generic_to_actual) generic_class.optional_vars);
    methods = (List.map (convert_generic_fdecl generic_to_actual) generic_class.methods)
  }
and
get_converted_generic_instantiations = function
  Classdecl(classdecl) as self ->
    if ((List.length classdecl.generics) > 0) && classdecl.source_class_name <> "" then
      if StringMap.mem classdecl.source_class_name generic_map then
        Classdecl((convert_instantiation classdecl (StringMap.find classdecl.source_class_name generic_
      else raise (Failure("Attempted to initialize " ^ classdecl.cname ^ " using generic class " ^ clas
    else if (List.length classdecl.generics) = 0 then
      self
   else Stmt(Expr(NullExpr)) (* We want to ignore these classes. This NullExpr will be filtered below.
| _ as self -> self
in
let program = (List.filter (fun a -> a <> Stmt(Expr(NullExpr))) (List.map get_converted_generic_instant
in

(* Next, figure out the type signature map for all functions on all classes.
   This will build a Map<ClassName, Map<FunctionSignature, RtypeOfFunction>> *)
let class_signatures =
(* Adds all the user defined functions, but also the auto-gen constructor signatures *)
let add_void_func_signature map func_signature = SignatureMap.add func_signature (Primitive(Void)) map
in
let func_signatures_for_class classdecl =
  let original_map = (List.fold_left add_fdecl SignatureMap.empty classdecl.methods) in
  let with_constructors = List.fold_left add_void_func_signature original_map (get_autogenerated_constr
  SignatureMap.add { fs_name = "to_string"; formal_types = []; } (Primitive(String)) with_constructors
in
let add_class_signature map = function
  Classdecl(classdecl) -> if StringMap.mem classdecl.cname map then raise (Failure(("Duplicate classes 
                          else StringMap.add classdecl.cname (func_signatures_for_class classdecl) map
| _ -> map
in
```

```
List.fold_left add_class_signature StringMap.empty program
in
let check_class_exists class_name =
  if StringMap.mem class_name class_signatures then ()
  else raise (Failure("Class name " ^ class_name ^ " was never defined."))
in
let rec check_class_exists_nested = function (* if defining an object array, make sure class is defined
    Class(name) -> check_class_exists name
  | Array(typ)  -> check_class_exists_nested typ
  | _           -> ()
in
(* Next, figure out the types for all variables for classes.
   This will build a Map<ClassName, Map<VariableName, TypeOfVariable>> *)
let class_variable_types =
let add_class_variable map tuple = StringMap.add (fst tuple) (snd tuple) map in
let get_tuple_from_assign = function
  RegularAssign(typ, str, _) -> (str, typ)
in
let get_tuple_from_bind bind = (snd bind, fst bind) in
let add_class_variables map = function
  (* Duplicate class names, duplicate variable names within a class,
     and improper assigns/binds are checked in other parts of the code *)
  Classdecl(classdecl) -> StringMap.add classdecl.cname
    (List.fold_left add_class_variable StringMap.empty
      ((List.map get_tuple_from_assign classdecl.static_vars) @
       (List.map get_tuple_from_bind classdecl.required_vars) @
       (List.map get_tuple_from_assign classdecl.optional_vars))) map
| _ -> map
in
List.fold_left add_class_variables StringMap.empty program
in
let class_static_vars =
let add_class_variable map tuple = StringMap.add (fst tuple) (snd tuple) map in
let get_tuple_from_assign = function
  RegularAssign(typ, str, _) -> (str, typ)
in
let add_class_variables map = function
  (* Duplicate class names, duplicate variable names within a class,
     and improper assigns/binds are checked in other parts of the code *)
  Classdecl(classdecl) -> StringMap.add classdecl.cname
    (List.fold_left add_class_variable StringMap.empty
      (List.map get_tuple_from_assign classdecl.static_vars)) map
| _ -> map
in
List.fold_left add_class_variables StringMap.empty program
in

let rec check_fcall fname actuals v_symbol_tables =
  let checked_exprs = List.map (check_expr v_symbol_tables) actuals in
  let signature = { fs_name = fname; formal_types = List.map (fst) checked_exprs } in
  if fname = "println" && (List.length actuals) = 1 then
    (* Special convenience code to wrap all primitives to become a valid print call *)
    let wrapped_checked_exprs = match checked_exprs with
        [(Primitive(Int), _)] -> [wrap_to_string checked_exprs]
      | [(Primitive(Long), _)] -> [wrap_to_string checked_exprs]
      | [(Primitive(Float), _)] -> [wrap_to_string checked_exprs]
      | [(Primitive(Char), _)] -> [wrap_to_string checked_exprs]
      | [(Primitive(Bool), _)] -> [wrap_to_string checked_exprs]
      | [(NullType, _)] -> [wrap_to_string checked_exprs]
      | [(Class(_), _)] -> [wrap_to_string checked_exprs]
      | [(Array(_), _)] -> [wrap_to_string checked_exprs]
      | _ -> checked_exprs in
    let signature = { fs_name = fname; formal_types = List.map (fst) wrapped_checked_exprs } in
    if SignatureMap.mem signature function_signatures then ((SignatureMap.find signature function_signa
  (* Let len() pass through semant, len() is not a built in func and is in fact implemented as an LLVM
     let codegen handle error checking for len() *)
  else if fname = "len" && List.length actuals = 1 then
    Primitive(Int), SCall (SFuncCall(fname, checked_exprs))
  else
    let matching_signature = find_matching_signature signature function_signatures in
    let null_safe_checked_exprs = convert_nulls_in_checked_exprs checked_exprs matching_signature in
    ((SignatureMap.find matching_signature function_signatures), SCall (SFuncCall(fname, null_safe_check
and
check_mcall_prechecked checked_expr fname checked_actuals =
  match ((fst checked_expr)) with
  Class(class_name) ->
    let class_function_signatures = if StringMap.mem class_name class_signatures
                                    then StringMap.find class_name class_signatures
                                    else raise (Failure(("Class name " ^ class_name ^ " not found ")))
    in
    let signature = { fs_name = fname; formal_types = List.map (fst) checked_actuals } in
    let matching_signature = find_matching_signature signature class_function_signatures in
    let null_safe_checked_exprs = convert_nulls_in_checked_exprs checked_actuals matching_signature in
    ((SignatureMap.find matching_signature class_function_signatures), SCall (SMethodCall(checked_expr,
  | _ -> raise (Failure(("Attempted to call method on something that was not a class")))
and
check_mcall expr fname actuals v_symbol_tables =
  let checked_expr = check_expr v_symbol_tables expr in
  let checked_exprs = List.map (check_expr v_symbol_tables) actuals in
  check_mcall_prechecked checked_expr fname checked_exprs
and

check_call v_symbol_tables = function
  FuncCall(fname, exprs) -> (check_fcall fname exprs v_symbol_tables)
| MethodCall(expr, fname, exprs) -> check_mcall expr fname exprs v_symbol_tables
and

check_object_variable_access v_symbol_tables obj_var_access =
  let expr = obj_var_access.ova_expr in
  let class_name = obj_var_access.ova_class_name in
  let var_name = obj_var_access.ova_var_name in
  let is_static_access = obj_var_access.ova_is_static in
  if is_static_access then
    if StringMap.mem class_name class_static_vars then
      let class_variable_map = StringMap.find class_name class_static_vars in
      if StringMap.mem (var_name) class_variable_map then
        let typ_of_access = StringMap.find (var_name) class_variable_map in
        let sova = { sova_sexpr = (NullType, SNullExpr); sova_class_name = class_name; sova_var_name = v
        (typ_of_access, (SObjectVariableAccess sova))
      else raise (Failure("Could not find a static variable named " ^ (var_name) ^ " in class " ^ class_
    else raise (Failure("Could not find a definition for class name " ^ class_name))
  else
    let checked_expr = check_expr v_symbol_tables expr in
    let object_type = (fst checked_expr) in
    match object_type with
      Class(class_name) -> (* Then check that the variable being accessed on the class actually exists
        if StringMap.mem class_name class_variable_types then
          let class_variable_map = StringMap.find class_name class_variable_types in
          if StringMap.mem (var_name) class_variable_map then
            let typ_of_access = StringMap.find (var_name) class_variable_map in
            let sova = { sova_sexpr = checked_expr; sova_class_name = class_name; sova_var_name = var_na
            (typ_of_access, (SObjectVariableAccess sova))
          else raise (Failure("Could not find a variable named " ^ (var_name) ^ " in class " ^ class_nam
        else raise (Failure("Could not find a definition for class name " ^ class_name))
    | _ -> raise (Failure("Attempted to access variable " ^ (var_name) ^ " on something that isn't an ob
and

check_lhs_is_not_void = function
  Primitive(Void) -> raise (Failure("LHS of assignment cannot be void"))
| NullType -> raise (Failure("LHS of assignment cannot be null"))
| _ -> ()
and
(* Arrays are painful due to empty array literals and array literals containing NULL *)
check_array_assign lhs_name lhs_element_type this_scopes_v_table checked_expr =
  (* *_type is always like Array(Int). *_element_type is the type of the element
     making up the array, e.g. Int *)
  (* if defining an object array, make sure class is defined *)
  let _ = check_class_exists_nested lhs_element_type in
  let lhs_type = Array(lhs_element_type) in
  let rhs_type = (fst checked_expr) in
  match rhs_type with
    Array(_) ->
      if are_types_compatible lhs_type rhs_type then
        let converted_checked_expr = (lhs_type, (snd checked_expr)) in
        ((StringHash.add this_scopes_v_table lhs_name lhs_type); (SRegularAssign(lhs_type, lhs_name, con
      else if (rhs_type = lhs_type) then
        (* If we reached this point, the element type is not allowed to be null and it is non-zero. So
           types need to directly match. *)
        ((StringHash.add this_scopes_v_table lhs_name lhs_type); (SRegularAssign(lhs_type, lhs_name, che
      else
        raise (Failure(("Illegal assignment. LHS was type " ^ (str_of_typ (Array(lhs_element_type))) ^
  | _ -> raise (Failure("RHS of array assignment must be an array."))
and
check_regular_assign lhs_type lhs_name rhs_expr v_symbol_tables =
  let this_scopes_v_table = (List.hd v_symbol_tables) in
  let _ = (check_lhs_is_not_void lhs_type) in
  (if StringHash.mem this_scopes_v_table lhs_name then
    let existing_type = StringHash.find this_scopes_v_table lhs_name in
    if existing_type <> lhs_type then raise (Failure(("Variable " ^ lhs_name ^ " has type " ^ (str_of_ty

  let checked_expr = (check_expr v_symbol_tables rhs_expr) in
  let rhs_type = (fst checked_expr) in
  let lhs_rhs = { lhs = lhs_type; rhs = rhs_type } in
  let _ = (match lhs_type with Class(class_name) -> (check_class_exists class_name) | _ -> ()) in
  match lhs_type with
    Array(lhs_element_type) -> check_array_assign lhs_name lhs_element_type this_scopes_v_table checked
  | _ ->
    if (rhs_type = lhs_type) || ((type_is_nullable lhs_type) && (rhs_type = NullType)) then
      let converted_null = (lhs_type, (snd checked_expr)) in
      ((StringHash.add this_scopes_v_table lhs_name lhs_type); (SRegularAssign(lhs_type, lhs_name, conve
    else if LhsRhsMap.mem lhs_rhs coerceable_types_map then
      let converter = LhsRhsMap.find lhs_rhs coerceable_types_map in
      ((StringHash.add this_scopes_v_table lhs_name lhs_type); (SRegularAssign(lhs_type, lhs_name, (con
    else
      raise (Failure(("Illegal assignment. LHS was type " ^ (str_of_typ lhs_type) ^ " but RHS type was
and

check_assign v_symbol_tables = function
  RegularAssign(lhs_type, lhs_name, rhs_expr) -> (check_regular_assign lhs_type lhs_name rhs_expr v_symb
and

check_regular_update id updateop rhs_expr v_symbol_tables =
  let lhs_type = (type_of_identifier v_symbol_tables id) in
  let checked_expr = check_expr v_symbol_tables rhs_expr in
  let rhs_type = (fst checked_expr) in
  let lhs_rhs = { lhs = lhs_type; rhs = rhs_type } in
  match updateop with
    Eq ->
          if (lhs_type = rhs_type) || ((type_is_nullable lhs_type) && (rhs_type = NullType)) then
            let converted_null = (lhs_type, (snd checked_expr)) in
            (lhs_type, SUpdate (SRegularUpdate(id, updateop, converted_null)))
          else if LhsRhsMap.mem lhs_rhs coerceable_types_map then
            let converter = LhsRhsMap.find lhs_rhs coerceable_types_map in
            ((fst checked_expr), SUpdate (SRegularUpdate(id, updateop, (converter [checked_expr]))))
          else raise (Failure(("Illegal update. LHS was type " ^ (str_of_typ lhs_type) ^ " but RHS type
and
get_sova = function
  SObjectVariableAccess(sova) -> sova
| _ -> raise (Failure("Found something other than an object variable access in an unexpected place."))
and
check_object_variable_update object_variable_access updateop rhs_expr v_symbol_tables =
  let checked_object_variable_access = check_object_variable_access v_symbol_tables object_variable_acce
  let sova = (get_sova (snd checked_object_variable_access)) in
  let lhs_type = (fst checked_object_variable_access) in
  let checked_expr = (check_expr v_symbol_tables rhs_expr) in
  let rhs_type = (fst checked_expr) in
  let lhs_rhs = { lhs = lhs_type; rhs = rhs_type } in
  match updateop with
    Eq ->
          if (lhs_type = rhs_type) || ((type_is_nullable lhs_type) && (rhs_type = NullType)) then
            let converted_null = (lhs_type, (snd checked_expr)) in
            (lhs_type, SUpdate (SObjectVariableUpdate(sova, updateop, converted_null)))
          else if LhsRhsMap.mem lhs_rhs coerceable_types_map then
            let converter = LhsRhsMap.find lhs_rhs coerceable_types_map in
            ((fst checked_expr), SUpdate (SObjectVariableUpdate(sova, updateop, (converter [checked_expr
          else raise (Failure(("Illegal object variable update. LHS was type " ^ (str_of_typ lhs_type)
and
check_array_access_update array_access updateop rhs_expr v_symbol_tables =
  let checked_array_access = check_array_access (fst array_access) (snd array_access) v_symbol_tables in
  let sarray_access = match (snd checked_array_access) with SArrayAccess(tuple) -> tuple | _ -> raise (
  let lhs_type = (fst checked_array_access) in
  let checked_expr = (check_expr v_symbol_tables rhs_expr) in
  let rhs_type = (fst checked_expr) in
  let lhs_rhs = { lhs = lhs_type; rhs = rhs_type } in
  match updateop with
    Eq ->
          if (lhs_type = rhs_type) || ((type_is_nullable lhs_type) && (rhs_type = NullType)) then
            let converted_null = (lhs_type, (snd checked_expr)) in
            (lhs_type, SUpdate (SArrayAccessUpdate(sarray_access, updateop, converted_null)))
          else if LhsRhsMap.mem lhs_rhs coerceable_types_map then
            let converter = LhsRhsMap.find lhs_rhs coerceable_types_map in
            ((fst checked_expr), SUpdate (SArrayAccessUpdate(sarray_access, updateop, (converter [check
          else raise (Failure(("Illegal array update. LHS was type " ^ (str_of_typ lhs_type) ^ " but RH
and
check_update v_symbol_tables = function
  RegularUpdate(id, updateop, expr) -> check_regular_update id updateop expr v_symbol_tables
| ObjectVariableUpdate(object_variable_access, updateop, expr) -> check_object_variable_update object_v
| ArrayAccessUpdate(array_access, updateop, expr) -> check_array_access_update array_access updateop ex
and

check_array_access array_expr int_expr v_symbol_tables =
  (* Check the expr used to index into an array is an int.
     Check that the expr being indexed into is actually an array.
     Checking that the index is not out of bounds has to be done in codegen. *)
  let rhs_checked_expr = (check_expr v_symbol_tables int_expr) in
  let _ = (check_type_is_int (fst rhs_checked_expr)) in
  let lhs_checked_expr = (check_expr v_symbol_tables array_expr) in
  match (fst lhs_checked_expr) with
      Array(typ) -> (typ, SArrayAccess(lhs_checked_expr, rhs_checked_expr))
    | _ -> raise (Failure("Attempted to access something like it was an array, but it was not an array.
and

check_exprs_have_same_type l = match l with
  [] -> ()
| [(_, _)] -> ()
| (typ1, _) :: (typ2, _) :: _ ->
    if (not (are_types_compatible typ1 typ2)) then
      raise (Failure ("Array literal had incompatible types " ^ (str_of_typ typ1) ^ " " ^ (str_of_typ t
    else check_exprs_have_same_type (List.tl l)
and
get_checked_exprs_len = function
  (_, SArrayLiteral(sexprs)) -> List.length sexprs
| _ -> -1
and
check_length expected_length checked_expr =
  if (get_checked_exprs_len checked_expr) <> expected_length then
    raise (Failure("Mismatched array lengths detected inside array literal"))
  else ()
and
check_array_literal exprs v_symbol_tables =
  if (List.length exprs) = 0 then (Array(NullType), (SArrayLiteral []))
  else
  let checked_exprs = List.map (check_expr v_symbol_tables) exprs in
  let _ = check_exprs_have_same_type checked_exprs in
  let non_null_list = List.filter (function p -> p <> NullType) (List.map fst checked_exprs) in
  let element_typ = if List.length non_null_list > 0 then (List.hd non_null_list) else NullType in
  let convert_null typ checked_expr = match (fst checked_expr) with
    NullType -> (typ, (snd checked_expr))
  | _ -> checked_expr in
  let null_safe_checked_exprs = List.map (convert_null element_typ) checked_exprs in
  let expected_length = if List.length null_safe_checked_exprs > 0 then (get_checked_exprs_len (List.hd
  let _ = List.iter (check_length expected_length) null_safe_checked_exprs in
  let typ = Array(element_typ) in (typ, (SArrayLiteral null_safe_checked_exprs))
and

check_expr_is_int_literal = function
  SIntLiteral(_) -> ()
| _              -> raise (Failure ("default arrays must be intitialized with int literals at this time
and
check_default_array typ exprs v_symbol_tables = match typ with
  Array(_) -> let _ = check_class_exists_nested typ in
              let checked_exprs = List.map (check_expr v_symbol_tables) exprs in
              let _ = List.iter (check_type_is_int) (List.map fst checked_exprs) in
              let _ = List.iter (check_expr_is_int_literal) (List.map snd checked_exprs) in
              (typ, SDefaultArray(typ, checked_exprs))
| _        -> raise (Failure ("Attempted to create a default non-array type"))
and

check_unop unaryop expr v_symbol_tables =
  let checked_expr = (check_expr v_symbol_tables expr) in
  let typ = (fst checked_expr) in
  match unaryop with
    Not -> (match typ with
              Primitive(Bool) -> (typ, SUnop(unaryop, checked_expr))
            | _ -> raise (Failure("Attempted to call unary op not on something that wasn't a boolean"))
  | Neg -> (match typ with
              Primitive(Int) | Primitive(Long) | Primitive(Float) -> (typ, SUnop(unaryop, checked_expr)
            | _ -> raise (Failure("Attempted to call unary op - on something that wasn't a number")))
and

coerce_binop_exprs checked_lhs checked_rhs =
  let lhs_type = (fst checked_lhs) in
  let rhs_type = (fst checked_rhs) in
  if lhs_type = Primitive(Int) && rhs_type = Primitive(Long) then
    ((wrap_int_to_long [checked_lhs]), checked_rhs)
  else if lhs_type = Primitive(Long) && rhs_type = Primitive(Int) then
    (checked_lhs, (wrap_int_to_long [checked_rhs]))
  else if lhs_type = Primitive(Int) && rhs_type = Primitive(Float) then
    ((wrap_int_to_float [checked_lhs]), checked_rhs)
  else if lhs_type = Primitive(Float) && rhs_type = Primitive(Int) then
    (checked_lhs, (wrap_int_to_float [checked_rhs]))
  else if lhs_type = Primitive(String) then
    (checked_lhs, (wrap_to_string [checked_rhs]))
  else if rhs_type = Primitive(String) then
```

```
    ((wrap_to_string [checked_lhs]), checked_rhs)
  else
    raise (Failure("No coercion rule found for " ^ (str_of_typ lhs_type) ^ " and " ^ (str_of_typ rhs_ty
and
check_binop_coerced checked_lhs binop checked_rhs =
  let lhs_type = (fst checked_lhs) in
  match binop with
    Plus -> (match lhs_type with
        Primitive(Int) | Primitive(Long) | Primitive(Float) | Primitive(String) ->
          (lhs_type, SBinop(checked_lhs, binop, checked_rhs))
      | Primitive(Char) -> (Primitive(String), SBinop((wrap_to_string [checked_lhs]), binop, (wrap_to_s
      | _ -> raise (Failure("Binop + is not available for type " ^ (str_of_typ lhs_type))))

    | Subtract | Times | Divide | Modulo -> (match lhs_type with
        Primitive(Int) | Primitive(Long) | Primitive(Float) -> (lhs_type, SBinop(checked_lhs, binop, ch
      | _ -> raise (Failure("Binops -, *, /, and % are not available for type " ^ (str_of_typ lhs_type)

    | DoubleEq -> (Primitive(Bool), SBinop(checked_lhs, binop, checked_rhs))

    | BoGT | BoLT | BoGTE | BoLTE -> (match lhs_type with
        Primitive(Int) | Primitive(Long) | Primitive(Float) -> (Primitive(Bool), SBinop(checked_lhs, bi
      | _ -> raise (Failure("Operators > < >= <= are not available for type " ^ (str_of_typ lhs_type)))
    | BoOr | BoAnd -> (match lhs_type with
        Primitive(Bool) -> (Primitive(Bool), SBinop(checked_lhs, binop, checked_rhs))
      | _ -> raise (Failure("or and and must take boolean types, not " ^ (str_of_typ lhs_type))))
and
check_binop lhs binop rhs v_symbol_tables =
  let checked_lhs = (check_expr v_symbol_tables lhs) in
  let checked_rhs = (check_expr v_symbol_tables rhs) in
  let lhs_type = (fst checked_lhs) in
  let rhs_type = (fst checked_rhs) in
  (* NULLs are only valid in a ==. As usual, NULLs have to be treated very carefully. *)
  if ((lhs_type = NullType) || (rhs_type = NullType)) && binop = DoubleEq then
    (Primitive(Bool), SBinop(checked_lhs, binop, checked_rhs))
  else (match lhs_type with
     Class(_) -> (try check_mcall_prechecked checked_lhs (binop_method_name binop) [checked_rhs]
                  (* check if the user defined an operator override. if they did , we call it like a me
                  with _ -> if binop = DoubleEq then (Primitive(Bool), SBinop(checked_lhs, binop, check
                            else raise (Failure("Attempted to use a binop on an object that is not supp
  | _ -> (if lhs_type = rhs_type then
            check_binop_coerced checked_lhs binop checked_rhs
          else
            let lhs_rhs = coerce_binop_exprs checked_lhs checked_rhs in
            check_binop_coerced (fst lhs_rhs) binop (snd lhs_rhs))
  )
and

check_object_instantiation class_name exprs v_symbol_tables =
  let checked_exprs = List.map (check_expr v_symbol_tables) exprs in
  if StringMap.mem class_name class_signatures then
    (* See if there is a constructor matching this signature *)
    let signatures_in_class = StringMap.find class_name class_signatures in
    let signature = { fs_name = "construct"; formal_types = List.map fst checked_exprs } in
    try (let matching_signature = find_matching_signature signature signatures_in_class in
      let null_safe_checked_exprs = convert_nulls_in_checked_exprs checked_exprs matching_signature in
      (Class(class_name), SObjectInstantiation(class_name, null_safe_checked_exprs)))
    with Failure(_) -> (raise (Failure("Attempted to initialize class " ^ class_name ^ " using a type s
  else raise (Failure("Attempted to initialize class " ^ class_name ^ " that does not exist."))
and

check_expr v_symbol_tables = function
  IntLiteral(i) -> (Primitive(Int), SIntLiteral(i))
| LongLiteral(l) -> (Primitive(Long), SLongLiteral(l))
| FloatLiteral(f) -> (Primitive(Float), SFloatLiteral(f))
| CharLiteral(c) -> (Primitive(Char), SCharLiteral(c))
| StringLiteral(s) -> (Primitive(String), SStringLiteral(s))
| BoolLiteral(b) -> (Primitive(Bool), SBoolLiteral(b))
| Id(id_str) -> (type_of_identifier v_symbol_tables id_str, SId(id_str))
| NullExpr -> (NullType, SNullExpr)
| Self -> (type_of_identifier v_symbol_tables "self", SSelf)
| Call(call) -> check_call v_symbol_tables call
| ObjectInstantiation(class_name, exprs) -> check_object_instantiation class_name exprs v_symbol_tables
| ObjectVariableAccess(object_variable_access) -> check_object_variable_access v_symbol_tables object_va
| ArrayAccess(array_name, expr) -> check_array_access array_name expr v_symbol_tables
| ArrayLiteral(exprs) -> check_array_literal exprs v_symbol_tables
| DefaultArray(typ, exprs) -> check_default_array typ exprs v_symbol_tables
| Binop(lhs_expr, binop, rhs_expr) -> check_binop lhs_expr binop rhs_expr v_symbol_tables
| Unop(unaryop, expr) -> check_unop unaryop expr v_symbol_tables
| Assign(assign) ->
    (let checked_assign = (check_assign v_symbol_tables assign) in
     match checked_assign with
       SRegularAssign(lhs_type, _, _) -> (lhs_type, SAssign(checked_assign))
     | _ -> raise (Failure("Found unexpected assign type")))
| Update(update) -> check_update v_symbol_tables update
and

check_elif v_symbol_tables expected_rtype elif =
  let checked_cond = (check_expr v_symbol_tables (fst elif)) in
  let _ = check_type_is_bool (fst checked_cond) in
  let checked_stmt_list = check_stmt_list ((StringHash.create 10)::v_symbol_tables) expected_rtype (snd
  (checked_cond, checked_stmt_list)
and
check_if if_cond_expr if_stmt_list elif_list else_stmt_list v_symbol_tables expected_rtype =
 (* check that all the conds are boolean *)
  let checked_if_cond = (check_expr v_symbol_tables if_cond_expr) in
  let _ = check_type_is_bool (fst checked_if_cond) in
  let checked_if_stmt_list = check_stmt_list ((StringHash.create 10)::v_symbol_tables) expected_rtype i
  let checked_elifs = List.map (check_elif v_symbol_tables expected_rtype) elif_list in
  let checked_else_list = check_stmt_list ((StringHash.create 10)::v_symbol_tables) expected_rtype else
  SIf(checked_if_cond, checked_if_stmt_list, checked_elifs, checked_else_list)
and

check_expr_is_update = function
  Update(_) -> ()
| NullExpr -> ()
| _ -> raise (Failure("Loop was expecting an update expr or a null expr"))
and
check_loop update_expr cond_expr stmt_list v_symbol_tables expected_rtype =
  let checked_cond = (check_expr v_symbol_tables cond_expr) in
  let _ = check_type_is_bool (fst checked_cond) in
  let _ = check_expr_is_update update_expr in
  let checked_update_expr = (check_expr v_symbol_tables update_expr) in
  let checked_stmt_list = check_stmt_list ((StringHash.create 10)::v_symbol_tables) expected_rtype stmt_
  SLoop(checked_update_expr, checked_cond, checked_stmt_list)
and

check_stmt v_symbol_tables expected_rtype = function
  Expr(expr) -> SExpr (check_expr v_symbol_tables expr)
| ReturnVoid -> (match expected_rtype with
                  Primitive(Void) -> SReturnVoid
                | NullType -> raise (Failure("Found return statement in unexpected place (i.e. a statem
                | _ -> raise (Failure("Expected a void return but found a " ^ (str_of_typ expected_rtyp
               )
| Return(expr) -> (match expected_rtype with
                    Primitive(Void) -> raise (Failure("Found a non-void return inside a function that r
                  | NullType -> raise (Failure("Found return statement in unexpected place (i.e. a stat
                  | _ ->
                    let checked_expr = (check_expr v_symbol_tables expr) in
                    let actual_rtype = fst checked_expr in
                    if expected_rtype <> actual_rtype then raise (Failure(("Mismatch between expected a
                    else SReturn (checked_expr)
                  )
| If(if_cond_expr, if_stmt_list, elif_list, else_stmt_list) -> (check_if if_cond_expr if_stmt_list elif_
| Loop(update_expr, cond_expr, stmt_list) -> check_loop update_expr cond_expr stmt_list v_symbol_tables
and
elif_always_returns elif =
  let stmt_list = (snd elif) in
  stmts_always_return stmt_list
and
all_are_true l = (* Check if all elements of a list are true *)
  not (List.mem false l)
and
stmt_always_returns = function
  Expr(_) -> false (* expressions are not returns *)
| ReturnVoid -> true
| Return(_) -> true (* Return statements always return *)
| If(_, if_stmt_list, elif_list, else_stmt_list) ->
    (* Checking if ifs always return is more complicated. *)
    if List.length else_stmt_list > 0 then
      (stmts_always_return if_stmt_list) && (all_are_true (List.map elif_always_returns elif_list)) &&
    else false (* Technically, if there is an elseless if, it always returns if the if statement is alw
                  and the if_stmt_list always returns, or if there is an elseless if+elifs, it always r
                  if+elifs are always true and their stmt lists always return. But this is too complica
                  since we may not know if an if will always return true, as the expr in an if can be g
                  runtime. *)
| Loop(_, _, _) -> false (* Similar to the above, a loop always returns if its body always returns, but
                            knowing whether the loop is guaranteed to be entered, which relies on informat
                            not be known at compile time. *)
and
(* Given a stmt_list, determine if it always returns.
   This is needed for the following semantic checks:
   -If a statement always returns, then any code after it is unreachable.
   -If a function has a non-void return, then its statement list must always return. *)
stmts_always_return = function
  [] -> false
| hd::tl -> let first_statement_returns = (stmt_always_returns hd) in
            if first_statement_returns && (List.length tl > 0) then
              raise (Failure("Found unreachable code."))
            else if first_statement_returns && (List.length tl == 0) then
              true
            else (* First statement did not always return, so check if the next ones did. *)
              stmts_always_return tl
and
check_stmt_list v_symbol_tables expected_rtype = function
  [ReturnVoid as s] -> [check_stmt v_symbol_tables expected_rtype s]
| [Return(_) as s] -> [check_stmt v_symbol_tables expected_rtype s]
| ReturnVoid :: _ -> raise (Failure("Nothing can follow a return statement"))
| Return(_) :: _ -> raise (Failure("Nothing can follow a return statement"))
| s :: stmt_list ->
    (* This let is neeeded to ensure the first statement is evaluated
       before the rest of the statements in the list. Our language is
       sequential from top to bottom but by default OCaml has :: as
       right associative *)
    let fst_stmt = check_stmt v_symbol_tables expected_rtype s in
    fst_stmt :: check_stmt_list v_symbol_tables expected_rtype stmt_list
| [] -> []
and

check_binds (kind : string) (binds : bind list) = (* check_binds was stolen from microc *)
  List.iter (function
      (Primitive(Void), b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
    | (NullType, b) -> raise (Failure ("illegal null " ^ kind ^ " " ^ b))
    | (Class(class_name), _) -> check_class_exists class_name
    | _ -> ()) binds;
  dups kind (List.sort (fun (a) (b) -> compare a b) (List.map (snd) binds))
and
check_fdecl is_in_class v_symbol_tables fdecl = (ignore (check_binds ("fdecl " ^ fdecl.fname) fdecl.for
  (let all_branches_return = stmts_always_return fdecl.body in
   if fdecl.rtype <> Primitive(Void) && (not all_branches_return) then
     (* If the function return type is not void, it must return in every branch *)
     raise (Failure("Function " ^ fdecl.fname ^ " had a non-void return but had branches that never retu
   (* Check that return type of constructor is always void *)
   else if (is_in_class && fdecl.fname = "construct" && fdecl.rtype <> Primitive(Void)) then
     raise (Failure("Found constructor that had rtype " ^ (str_of_typ fdecl.rtype) ^ " rather than void
   else if (is_in_class && fdecl.fname = "to_string" && ((List.length fdecl.formals) = 0) && fdecl.rtype
     raise (Failure("Found to_string that had rtype " ^ (str_of_typ fdecl.rtype) ^ " rather than string
   else
     { srtype = fdecl.rtype; sfname = fdecl.fname; sformals = fdecl.formals; sbody = check_stmt_list ((
and

get_names_from_assign = function
  RegularAssign(_, name, _) -> name
and
get_all_class_variable_names classdecl = (List.map get_names_from_assign classdecl.static_vars) @ (List
and
get_sstmt_from_bind cname bind =
  let typ = (fst bind) in
  let name = (snd bind) in
  let sova = { sova_sexpr = (Class(cname), SSelf); sova_class_name = ""; sova_var_name = name; sova_is_s
  SExpr (typ, SUpdate(SObjectVariableUpdate(sova, Eq, (typ, SId(name)))))
and
get_sstmt_from_assign_with_default v_symbol_tables cname = function
  RegularAssign(typ, name, expr) ->
    let sova = { sova_sexpr = (Class(cname), SSelf); sova_class_name = ""; sova_var_name = name; sova_i
    let checked_expr = (check_expr v_symbol_tables expr) in
    let lhs_type = typ in
    let converted_null = (lhs_type, (snd checked_expr)) in
    SExpr (typ, SUpdate(SObjectVariableUpdate(sova, Eq, converted_null)))
and
get_required_only_constructor v_symbol_tables classdecl =
  let body = (List.map (get_sstmt_from_bind classdecl.cname) classdecl.required_vars) @ (List.map (get_s
  { srtype = Primitive(Void); sfname = "construct"; sformals = classdecl.required_vars; sbody = body }
and
get_required_and_optional_constructor classdecl =
  let optional_var_binds = (List.map get_bind_from_assign classdecl.optional_vars) in
  let body = (List.map (get_sstmt_from_bind classdecl.cname) classdecl.required_vars) @ (List.map (get_s
  { srtype = Primitive(Void); sfname = "construct"; sformals = (classdecl.required_vars @ optional_var_b
and
get_autogenerated_constructors v_symbol_tables classdecl =
  if List.length classdecl.required_vars = 0 && List.length classdecl.optional_vars = 0 then [ { srtype
  else if List.length classdecl.required_vars > 0 && List.length classdecl.optional_vars = 0 then [get_r
  else [(get_required_only_constructor v_symbol_tables classdecl); (get_required_and_optional_construct
and
signature_is_not_yet_defined map sfdecl =
  let signature = (sget_signature sfdecl) in
  not (SignatureMap.mem signature map)
and
add_autogenerated_constructors v_symbol_tables classdecl checked_fdecls =
  (* This will add the autogenerated constructors to the checked classdecl.
     However, if there is already a constructor defined with the same signature, we don't
     throw an error or overwrite it here. The user defined constructor takes precedence over
     the auto-generated one. *)
  let autogenerated_constructors = get_autogenerated_constructors v_symbol_tables classdecl in
  let original_func_signatures = (List.fold_left add_fdecl SignatureMap.empty classdecl.methods) in
  checked_fdecls @ (List.filter (signature_is_not_yet_defined original_func_signatures) autogenerated_co
and
get_to_string_from_assign class_name = function
  RegularAssign(typ, name, _) ->
    let checked_expr = (typ, SObjectVariableAccess({ sova_sexpr = (Class(class_name), SId("self")); sova
    let inner_lhs = (Primitive(String), SStringLiteral(name ^ ":")) in
    let inner_rhs = (wrap_to_string [checked_expr]) in
    let inner = (Primitive(String), SBinop(inner_lhs, Plus, inner_rhs)) in
    let outer = (Primitive(String), SStringLiteral("\n")) in
    (Primitive(String), SBinop(inner, Plus, outer))
and
get_to_string_from_bind class_name bind =
  let typ = (fst bind) in
  let name = (snd bind) in
  let checked_expr = (typ, SObjectVariableAccess({ sova_sexpr = (Class(class_name), SId("self")); sova_
  let inner_lhs = (Primitive(String), SStringLiteral(name ^ ":")) in
  let inner_rhs = (wrap_to_string [checked_expr]) in
  let inner = (Primitive(String), SBinop(inner_lhs, Plus, inner_rhs)) in
  let outer = (Primitive(String), SStringLiteral("\n")) in
  (Primitive(String), SBinop(inner, Plus, outer))
and
get_default_to_string classdecl =
  let binops = (List.map (get_to_string_from_assign classdecl.cname) classdecl.static_vars) @ (List.map
  let combine_binops orig new_element = (Primitive(String), SBinop(orig, Plus, new_element)) in
  let predicate = (Primitive(Bool), SBinop( (Class(classdecl.cname), SId("self")) , DoubleEq, (NullType
  let check_null = (SIf(predicate, [SReturn((Primitive(String), SStringLiteral("NULL")))], [], [])) in
  let binops_combined = List.fold_left combine_binops (Primitive(String), SStringLiteral(classdecl.cname
  { srtype = Primitive(String); sfname = "to_string"; sformals = []; sbody = [check_null; SReturn(binops
and
add_to_string_if_not_present classdecl checked_fdecls =
  let original_func_signatures = (List.fold_left add_fdecl SignatureMap.empty classdecl.methods) in
  checked_fdecls @ (List.filter (signature_is_not_yet_defined original_func_signatures) [get_default_to_
and
convert_to_static_assign class_name = function
  SRegularAssign(lhs_typ, lhs_name, rhs_expr) -> SStaticAssign(class_name, lhs_typ, lhs_name, rhs_expr)
| _ -> raise (Failure("Found unexpected assignment type in class " ^ class_name))
and
check_classdecl v_symbol_tables classdecl =
  (* First check for duplicates in the class variables *)
  (dups ("classdecl " ^ classdecl.cname) (List.sort (fun (a) (b) -> compare a b) (get_all_class_variabl
  (* Then check all the assigns and bindings are correct *)
  let new_v_symbol_tables = (StringHash.create 10)::v_symbol_tables in (* class gets a new level of sym
  let checked_static_vars = List.map (convert_to_static_assign classdecl.cname) (List.map (check_assign
  let checked_required_vars = (ignore (check_binds ("classdecl " ^ classdecl.cname) classdecl.required_
  let checked_optional_vars = List.map (check_assign new_v_symbol_tables) classdecl.optional_vars in
  (* Inside this class, make sure "self" by itself points to this class type *)
  let _ = StringHash.add (List.hd new_v_symbol_tables) "self" (Class(classdecl.cname)) in
  let checked_fdecls = List.map (check_fdecl true new_v_symbol_tables) classdecl.methods in
  { scname = classdecl.cname; sstatic_vars = checked_static_vars; srequired_vars = checked_required_var
and

check_p_unit v_symbol_tables = function
  Stmt(stmt) -> SStmt (check_stmt v_symbol_tables (NullType) stmt)
| Fdecl(fdecl) -> SFdecl (check_fdecl false v_symbol_tables fdecl)
| Classdecl(classdecl) -> SClassdecl (check_classdecl v_symbol_tables classdecl)
in

List.map (check_p_unit [StringHash.create 10]) program
```

9.12 _tags

```
# Include the llvm and llvm.analysis packages while compiling
true: package(llvm), package(llvm.analysis)

# Enable almost all compiler warnings
true : warn(+a-4)
```

9.13 stdlib/arraylist.boom

```
class ArrayList[T]:
        optional:
                int size = 0
                T[] arr = default T[10]

        def add(T element):
                if self.size >= len(self.arr):
                        # TODO dynamic sized arrays are needed for this
                        #T[] copy = default T[len(self.arr) * 2]
                        T[] copy = default T[100]
                        int i = 0
                        loop i+=1 while i < len(self.arr):
                                copy[i] = self.arr[i]
                        self.arr = copy
                        self.add(element)
                else:
                        self.arr[self.size] = element
                        self.size += 1

        def get(int i) returns T:
                return self.arr[i]
```

9.14 stdlib/binarytree.boom

```
class Node[T]:
        required:
                T data

        optional:
                Node left = NULL
                Node right = NULL

        def print_level_order():
                int h = self.height()
                int i = 1
                loop i+=1 while i<=h:
                        self.print_given_level(i)

        def height() returns int:
                if self==NULL:
                        return 0
                else:
                        int lheight = self.left.height()
                        int rheight = self.right.height()
                        if lheight>rheight:
                                return lheight+1
                        else:
                                return rheight+1

        def print_given_level(int level):
                if self==NULL:
                        return
                if level==1:
```

```
                println(self.data)
        elif level>1:
                self.left.print_given_level(level-1)
                self.right.print_given_level(level-1)
```

9.15 stdlib/helloworld.boom

```
println("Hello, world! Welcome to Boomslang.")
```

9.16 stdlib/linkedlist.boom

```
class LinkedList[T]:
        required:
                T element

        optional:
                LinkedList next = NULL

        def _+(LinkedList other) returns LinkedList:
                # append the other list to this one
                if self.next == NULL:
                        self.next = other
                        return self
                else:
                        self.next + other
                        return self

        def _+(T new_element) returns LinkedList:
                if self.next == NULL:
                        self.next = LinkedList(new_element, NULL)
                        return self
                else:
                        self.next + LinkedList(new_element, NULL)
                        return self

        def _++(LinkedList other):
                if self.next == NULL:
                        self.next = other
                else:
                        self.next ++ other

        def _++(T new_element):
                if self.next == NULL:
                        self.next = LinkedList(new_element, NULL)
                else:
                        self.next ++ LinkedList(new_element, NULL)

        def _==(LinkedList other) returns boolean:
                # Custom defined equality function
                if self == NULL:
                        return other == NULL
                elif other == NULL:
                        return false
                else:
                        return (self.element == other.element) and (self.next == other.next)

        def size() returns int:
                int size = 1
                LinkedList next_temp = self.next
                loop while (next_temp != NULL):
                        size += 1
                        next_temp = next_temp.next
                return size

        def to_string() returns string:
                string temp = ""
                if self == NULL:
                        return "NULL"
                elif self.next == NULL:
                        return temp + self.element
                else:
                        return temp + self.element + " -> " + self.next.to_string()
```

9.17 stdlib/math.boom

```
def gcd(int a, int b) returns int:
        loop while a != b:
                if a > b:
                        a = a - b
                else:
                        b = b - a
        return a

def pow(int a, int b) returns int:
        if b == 0:
                return 1
        if b == 1:
                return a
        int i = 0
        int ans = 1
        loop i+=1 while i<b:
                ans *= a
        return ans
```

10 Test Suite

The following is a large collection of tests (over 250 tests in total). Each test file (.boom) is labelled by its file name before the test body and is followed by its expected output file name (.err/.out) and output file body.

```
fail-arrays1.boom:

int[] arr = [1,2,3,"string"]




fail-arrays1.err:
```

```
Fatal error: exception Failure("Array literal had incompatible types int string")
```

fail-arrays2.boom:

```
int[] arr = [1,4, get_string()]

def get_string() returns string:
return "not an int!"
```

fail-arrays2.err:

```
Fatal error: exception Failure("Array literal had incompatible types int string")
```

fail-arrays3.boom:

```
int[] arr = default string[2]
```

fail-arrays3.err:

```
Fatal error: exception Failure("Illegal assignment. LHS was type int[] but RHS type was string[]")
```

fail-arrays4.boom:

```
int myint = default int
```

fail-arrays4.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-assign1.boom:

```
int i = 42
```

```
i = 10
boolean b = true
b = false
i = false # Fail: assigning a bool to an integer
```

fail-assign1.err:

```
Fatal error: exception Failure("Illegal update. LHS was type int but RHS type was boolean")
```

fail-assign2.boom:

```
int i = 0
boolean b = true
b = 48 # Fail: assigning an integer to a bool
```

fail-assign2.err:

```
Fatal error: exception Failure("Illegal update. LHS was type boolean but RHS type was int")
```

fail-assign3.boom:

```
def myvoid() returns void:
return

int i = 0
i = myvoid() # Fail: assigning a void to an integer
```

fail-assign3.err:

```
Fatal error: exception Failure("Illegal update. LHS was type int but RHS type was void")
```

fail-assigning-to-undeclared-class.boom:

```
MyClass foo = NULL
```

fail-assigning-to-undeclared-class.err:

```
Fatal error: exception Failure("Class name MyClass was never defined.")
```

fail-bad-array-assign.boom:

```
string[] arr = ["one", "two", "three"]
arr[1] = 50.0
```

fail-bad-array-assign.err:

```
Fatal error: exception Failure("Illegal array update. LHS was type string but RHS type was float")
```

fail-bad-class1.boom:

```
class MyClass:
required:
MyOtherClass c
```

fail-bad-class1.err:

```
Fatal error: exception Failure("Class name MyOtherClass was never defined.")
```

fail-bad-method-call.boom:

```
(2+2).foo()
```

fail-bad-method-call.err:

```
Fatal error: exception Failure("Attempted to call method on something that was not a class")
```

fail-bad-obj-var-access.boom:

```
(2+2).foo
```

fail-bad-obj-var-access.err:

```
Fatal error: exception Failure("Attempted to access variable foo on something that isn't an object.")
```

fail-bad_array_init.boom:

```
def my_func(int x) returns string:
return "hey"

int[] my_array = [1, (2+2), my_func(1), my_func(1) + 1, 2 * (3 + 1)]
```

fail-bad_array_init.err:

```
Fatal error: exception Failure("Array literal had incompatible types int string")
```

fail-bad_constructor_1.boom:

```
class MyClass:
static:
my_object.x = 5
```

fail-bad_constructor_1.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-bad_constructor_2.boom:

```
class MyClass:
static:
void x = NULL
```

fail-bad_constructor_2.err:

```
Fatal error: exception Failure("LHS of assignment cannot be void")
```

fail-bad_indentation_1.boom:

```
int x = 5
def myfunc(int x, MyObject foo) returns string:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
elif x > 20:
println("hey")
elif x > 30:
println("hey")
println("hey")
else:
println("hey")




int x = 50
```

fail-bad_indentation_1.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-bad_indentation_2.boom:

```
int x = 5
def myfunc(int x, MyObject foo) returns string:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
```

```
elif x > 20:
println("hey")
elif x > 30:
println("hey")
println("hey")
else:
println("hey")



int x = 50
```

fail-bad_indentation_2.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-bad_loop.boom:

```
loop x+2 while:
2+2
```

fail-bad_loop.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-bad_negation.boom:

```
boolean x = true
-x
```

fail-bad_negation.err:

```
Fatal error: exception Failure("Attempted to call unary op - on something that wasn't a number")
```

```
fail-bad_not.boom:

int x = 5
not x
```

```
fail-bad_not.err:

Fatal error: exception Failure("Attempted to call unary op not on something that wasn't a boolean")
```

```
fail-bad_returns_1.boom:

def my_func() returns int:
if x > 5:
println("hey")
else:
return 5
```

```
fail-bad_returns_1.err:

Fatal error: exception Failure("Function my_func had a non-void return but had branches that never retu
```

```
fail-bad_returns_2.boom:

def my_func() returns int:
println("hey")
```

```
fail-bad_returns_2.err:

Fatal error: exception Failure("Function my_func had a non-void return but had branches that never retu
```

```
fail-constructors-must-not-have-returns.boom:
```

```
class MyClass:
def construct() returns int:
return 5
```

```
fail-constructors-must-not-have-returns.err:

Fatal error: exception Failure("Found constructor that had rtype int rather than void.")
```

```
fail-dead1.boom:

def foo() returns int:
int i = 15
return i
i = 32 # Error: code after a return
```

```
fail-dead1.err:

Fatal error: exception Failure("Found unreachable code.")
```

```
fail-duplicate_function_args.boom:

def my_func(int x, float x) -> returns int:
return 5
```

```
fail-duplicate_function_args.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-emoji-char.boom:

char c = ''
```

```
fail-emoji-char.err:

Fatal error: exception Failure("Illegal character: \'")
```

```
fail-empty-arrays1.boom:

def myfunc(int[] arr):
println("hey")

def myfunc(float[] arr):
println("hey")

# this is ambiguous
myfunc([])
```

```
fail-empty-arrays1.err:

Fatal error: exception Failure("The call to myfunc is ambiguous.")
```

```
fail-expr1.boom:

int a = 1
boolean b = true

def foo(int c, boolean d) returns void:
int dd = 2
boolean e = false
a = a + c
c = c - a
a = a * 3
c = c / 2
d = d + a # Error: boolean + int
```

```
fail-expr1.err:

Fatal error: exception Failure("No coercion rule found for boolean and int")
```

```
fail-expr2.boom:
```

```
int a = 1
boolean b = true

def foo(int c, boolean d) returns void:
int d = 3
boolean e = false
b = b + a # Error: bool + int
```

fail-expr2.err:

```
Fatal error: exception Failure("Variable d has type boolean but you attempted to assign it to type int"
```

fail-expr3.boom:

```
int a = 1
float b = 2.12

def foo(int c, float d) returns void:
int d = 3
float e = 4.12
b = b + a # Error: float + int
```

fail-expr3.err:

```
Fatal error: exception Failure("Variable d has type float but you attempted to assign it to type int")
```

fail-float1.boom:

```
if -3.5 and true: # Float with AND?
println("Error")
```

fail-float1.err:

```
Fatal error: exception Failure("No coercion rule found for float and boolean")
```

fail-float2.boom:

```
if -3.5 and 2.5: # Float with AND?
println("Error")
```

fail-float2.err:

```
Fatal error: exception Failure("or and and must take boolean types, not float")
```

fail-for1.boom:

```
def foo() returns int:
int i = 0
loop i += 1 while true:
println("forever")

i = 0
loop i += 1 while i < 10:
if (i == 3):
return 42

i = 0
loop j += 1 while i < 10:
println("Error") # j undefined
return 0
```

fail-for1.err:

```
Fatal error: exception Failure("undeclared identifier j")
```

fail-for2.boom:

```
int i = 0
loop i += 1 while j < 10: # j undefined
println("Error")
```

fail-for2.err:

```
Fatal error: exception Failure("undeclared identifier j")
```

fail-for3.boom:

```
int i = 0
loop i += 1 while i: # i is an integer, not Boolean
println("Error")
```

fail-for3.err:

```
Fatal error: exception Failure("Expected expr of type bool")
```

fail-for4.boom:

```
int i = 0
loop j += 1 while i < 10: # j is undefined
println("Error")
```

fail-for4.err:

```
Fatal error: exception Failure("undeclared identifier j")
```

fail-for5.boom:

```
int i = 0
loop i += 1 while i < 10:
foo() # Error: no function foo
```

fail-for5.err:

```
Fatal error: exception Failure("No matching signature found for function call foo")
```

fail-func1.boom:

```
def bar() returns int:
return 0

def bar() returns void:
return # Error: duplicate function bar
```

fail-func1.err:

```
Fatal error: exception Failure("Duplicate function signatures detected for bar")
```

fail-func2.boom:

```
def bar(int a, boolean b, int a) returns int: # Error: duplicate formal a in bar
return 0
```

fail-func2.err:

```
Fatal error: exception Failure("duplicate fdecl bar a")
```

fail-func3.boom:

```
def bar(int a, void b, int c) returns void: # Error: illegal void formal b
return
```

fail-func3.err:

```
Fatal error: exception Failure("illegal void fdecl bar b")
```

fail-func4.boom:

```
def println(string b) returns void:
return
```

fail-func4.err:

```
Fatal error: exception Failure("Duplicate function signatures detected for println")
```

fail-func5.boom:

```
def bar() returns int:
void b = 2 # Error: illegal void local b
return 0
```

fail-func5.err:

```
Fatal error: exception Failure("LHS of assignment cannot be void")
```

fail-func6.boom:

```
def foo(int a, boolean b) returns void:
return
foo(42) # Wrong number of arguments
```

fail-func6.err:

```
Fatal error: exception Failure("No matching signature found for function call foo")
```

fail-func7.boom:

```
def foo(int a, boolean b) returns void:
return
foo(42, true, false) # Wrong number of arguments
```

fail-func7.err:

```
Fatal error: exception Failure("No matching signature found for function call foo")
```

fail-func8.boom:

```
def foo(int a, boolean b) returns void:
return

def bar() returns void:
return

foo(42, bar()) # int and void, not int and bool
```

fail-func8.err:

```
Fatal error: exception Failure("No matching signature found for function call foo")
```

fail-func9.boom:

```
def foo(int a, boolean b) returns void:
return

foo(42, 42) # Fail: int, not bool
```

fail-func9.err:

```
Fatal error: exception Failure("No matching signature found for function call foo")
```

fail-func-with-undeclared-class.boom:

```
def myfunc(MyClass foo):
return
```

fail-func-with-undeclared-class.err:

```
Fatal error: exception Failure("Class name MyClass was never defined.")
```

fail-generic1.boom:

```
class MyGeneric[K]:
required:
K k
class MyActual = MyGeneric(int, string)
```

fail-generic1.err:

```
Fatal error: exception Failure("Attempted to initialize MyGeneric as MyActual but 1 types were expected
```

fail-generic2.boom:

```
class MyGeneric:
required:
int x
class MyGeneric[G]:
required:
G g
```

fail-generic2.err:

```
Fatal error: exception Failure("Detected duplicate declarations for class MyGeneric")
```

fail-generic3.boom:

```
class MyGeneric[G]:
required:
G g
class MyActual = MyGeeneric(int)
```

fail-generic3.err:

```
Fatal error: exception Failure("Attempted to initialize MyActual using generic class MyGeeneric, but no
```

fail-generic4.boom:

```
class MyGeneric[G, G]: # can't duplicate the type name
```

```
required:
G g

class MyActual = MyGeneric(int, int)
```

fail-generic4.err:

```
Fatal error: exception Failure("Found duplicate generic type Class_G_ in class MyGeneric")
```

fail-generic5.boom:

```
class MyGeneric[G]:
required:
G g
def myfunc(MyGeneric g):
return
```

fail-generic5.err:

```
Fatal error: exception Failure("Class name MyGeneric was never defined.")
```

fail-global1.boom:

```
int c = 0
boolean b = true
void a = 3 # global variables should not be void

def foo() returns int:
return 0
```

fail-global1.err:

```
Fatal error: exception Failure("LHS of assignment cannot be void")
```

fail-if1.boom:

```
if (true):
int a = 0
if (false):
int b = 1
else:
int c = 2
if (42): # Error: non-bool predicate
int d = 3
```

fail-if1.err:

```
Fatal error: exception Failure("Expected expr of type bool")
```

fail-if2.boom:

```
if (true):
foo # Error: undeclared variable
```

fail-if2.err:

```
Fatal error: exception Failure("undeclared identifier foo")
```

fail-if3.boom:

```
if (true):
42
else:
bar # Error: undeclared variable
```

fail-if3.err:

```
Fatal error: exception Failure("undeclared identifier bar")
```

fail-invalid-self-return.boom:

```
class MyClass:
```

```
def foo() returns int:
return self
```

fail-invalid-self-return.err:

```
Fatal error: exception Failure("Mismatch between expected and actual return type")
```

fail-invalid_arithmetic.boom:

```
5 *
```

fail-invalid_arithmetic.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-invalid_array_access.boom:

```
int x = 5
x[5]
```

fail-invalid_array_access.err:

```
Fatal error: exception Failure("Attempted to access something like it was an array, but it was not an a
```

fail-invalid_array_literal_1.boom:

```
[1, 2, 3,]
```

fail-invalid_array_literal_1.err:

```
Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_array_literal_2.boom:

[[1, 2, 3]
```

```
fail-invalid_array_literal_2.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_assignment_1.boom:

int =
```

```
fail-invalid_assignment_1.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_char_literal.boom:

char c = ''
```

```
fail-invalid_char_literal.err:

Fatal error: exception Failure("Illegal character: \'")
```

```
fail-invalid_double_eq.boom:

x ==
```

fail-invalid_double_eq.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-invalid_floating_point.boom:

```
float v = 1.
```

fail-invalid_floating_point.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-invalid_function_call.boom:

```
myfunc(1,2,)
```

fail-invalid_function_call.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-invalid_long_assignment.boom:

```
long f0oo = 5000LL
```

fail-invalid_long_assignment.err:

```
Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_multi_comment.boom:

comment #/
```

```
fail-invalid_multi_comment.err:

Fatal error: exception Failure("undeclared identifier comment")
```

```
fail-invalid_newlines.boom:


 = x
```

```
fail-invalid_newlines.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_null_assignment.boom:

NULL = int x
```

```
fail-invalid_null_assignment.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_object_usage.boom:

myobject.
```

```
fail-invalid_object_usage.err:

Fatal error: exception Parsing.Parse_error



fail-invalid_plus_eq.boom:

int x += 5




fail-invalid_plus_eq.err:

Fatal error: exception Parsing.Parse_error



fail-invalid_primitive_type.boom:

short x = 500




fail-invalid_primitive_type.err:

Fatal error: exception Parsing.Parse_error



fail-invalid_self_usage.boom:

self.




fail-invalid_self_usage.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_single_comment.boom:

/ comment
```

```
fail-invalid_single_comment.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_times_eq.boom:

int x *= 6
```

```
fail-invalid_times_eq.err:

Fatal error: exception Parsing.Parse_error
```

```
fail-invalid_type_return_1.boom:

def my_func() returns int:
if x > 5:
return
else:
return 5
```

```
fail-invalid_type_return_1.err:

Fatal error: exception Failure("undeclared identifier x")
```

```
fail-invalid_type_return_2.boom:

def my_func() returns int:
if x > 5:
return "string"
```

```
else:
return 5
```

fail-invalid_type_return_2.err:

```
Fatal error: exception Failure("undeclared identifier x")
```

fail-multi_objop.boom:

```
Wfoueb $^@#&@ Wefoudvn %&@@$^ HdfEFow
```

fail-multi_objop.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-nested-array-literals1.boom:

```
[ [[1, 2], [1, 2], [1, 2]]      ,      [[1, 2],[1, 2],[1, 2]]      ,      [[1, 2],[1, 2],[1, 2, 3]]   ]
```

fail-nested-array-literals1.err:

```
Fatal error: exception Failure("Mismatched array lengths detected inside array literal")
```

fail-nested-array-literals2.boom:

```
[ NULL, [] ]
```

fail-nested-array-literals2.err:

```
Fatal error: exception Failure("Array literal had incompatible types NULL Array_NULL_")
```

fail-nonsense.boom:

```
%-$_? !?
```

fail-nonsense.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-nothing_comes_after_return.boom:

```
def myfunc():
return
println("hey")
```

fail-nothing_comes_after_return.err:

```
Fatal error: exception Failure("Found unreachable code.")
```

fail-no-func-within-func.boom:

```
def myfunc():
def myfunc2():
println("hey")
println("hey")
```

fail-no-func-within-func.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-no_matching_signature.boom:

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

myfunc(5)
```

fail-no_matching_signature.err:

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

fail-null_assignment.boom:

```
NULL x = NULL
```

fail-null_assignment.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-print.boom:

```
def println(string b) returns void:
int a = 0
```

fail-print.err:

```
Fatal error: exception Failure("Duplicate function signatures detected for println")
```

fail-return1.boom:

```
def foo() returns int:
return true # Should return int
```

fail-return1.err:

```
Fatal error: exception Failure("Mismatch between expected and actual return type")
```

fail-return2.boom:

```
def foo() returns void:
if (true):
return 42 # Should return void
else:
return
```

fail-return2.err:

```
Fatal error: exception Failure("Found a non-void return inside a function that required a void return.")
```

fail-return-outside-of-func.boom:

```
int x = 5
return
```

fail-return-outside-of-func.err:

```
Fatal error: exception Failure("Found return statement in unexpected place (i.e. a statement outside of
```

fail-self-as-identifier.boom:

```
int self = 5
self
```

fail-self-as-identifier.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-self-on-its-own.boom:

```
self
```

fail-self-on-its-own.err:

```
Fatal error: exception Failure("undeclared identifier self")
```

fail-simple_objop.boom:

```
Wfoueb $^@#&@ Wefoudvn
```

fail-simple_objop.err:

```
Fatal error: exception Parsing.Parse_error
```

fail-void_assignment.boom:

```
void x = 5
```

fail-void_assignment.err:

```
Fatal error: exception Failure("LHS of assignment cannot be void")
```

runtime-err-divide-by-zero1.boom:

```
int x = 0
5 / x
```

runtime-err-divide-by-zero1.out:

```
DivideByZeroException
```

runtime-err-divide-by-zero2.boom:

```
long x = 0L
5L / x
```

runtime-err-divide-by-zero2.out:

```
DivideByZeroException
```

runtime-err-divide-by-zero3.boom:

```
5 / (2.2 - 2.2 + 0.0)
```

runtime-err-divide-by-zero3.out:

```
DivideByZeroException
```

runtime-err-mod-by-zero1.boom:

```
5 % 0
```

runtime-err-mod-by-zero1.out:

```
ModByZeroException
```

runtime-err-mod-by-zero2.boom:

```
class MyClass:
optional:
```

```
long mylong = 0L

MyClass foo = MyClass()
2 % foo.mylong
```

runtime-err-mod-by-zero2.out:

```
ModByZeroException
```

runtime-err-mod-by-zero3.boom:

```
def myfunc() returns float:
return 2.0 - 2.0

1 % myfunc()
```

runtime-err-mod-by-zero3.out:

```
ModByZeroException
```

runtime-err-npe1.boom:

```
class MyClass:
required:
int x

MyClass foo = NULL
foo.x
```

runtime-err-npe1.out:

```
NullPointerException
```

runtime-err-npe2.boom:

```
class MyClass:
```

```
required:
int x

MyClass foo = NULL
foo.x += 5
```

runtime-err-npe2.out:

```
NullPointerException
```

test-add1.boom:

```
def add(int x, int y) returns int:
return x + y

println(add(17, 25))
```

test-add1.out:

```
42
```

test-adding-chars.boom:

```
println('a' + 'b')
string foo = 'c' + 'd'
println(foo)
println(myfunc())
def myfunc() returns string:
return 'e' + 'f' + 'g'
```

test-adding-chars.out:

```
ab
cd
efg
```

test-arith1.boom:

```
println(39 + 3)
```

test-arith1.out:

```
42
```

test-arith2.boom:

```
println(1 + 2 * 3 + 4)
```

test-arith2.out:

```
11
```

test-arith3.boom:

```
def foo(int a) returns int:
return a

int a = 42
a = a + 5
println(a)
```

test-arith3.out:

```
47
```

test-arith4.boom:

```
# Test printing
println(NULL)
println(true)
println(false)
println(50L)
println(2.2)
println(2)
println("hello")
println('c')
println("hello"+'c')
println(true+"hello")
println(1+"hello")
println(2.2+"hello")
println("hello"+5L)
println("hello"+NULL)
println("hello"+"world")
println(("hello " + NULL) + ("! 2+2" + " is equal to ") + (2+2))
/# test multiline comments

get totally ignored

because i am just writing nonsense here

""" to test#/
# Test unops
println(not true)
println(not false)
println(not (3 > 2))
println(-2)
println(-2.2)
println(-5L)
# Int section
println(2+2)
println(5-2)
println(5*5)
println(100/10)
println(5%2)
println(5 == 5)
println(5 != 5)
println(2 > 5)
println(2 < 5)
println(3 >= 2)
println(3 <= 2)
# Long section
println(2L+2L)
println(5L-2L)
println(5L*5L)
println(100L/10L)
println(5L%2L)
println(5L == 5L)
println(5L != 5L)
println(2L > 5L)
println(2L < 5L)
println(3L >= 2L)
println(3L <= 2L)
# Float section
println(2.0+2.0)
println(5.0-2.0)
println(5.0*5.0)
println(100.0/10.0)
println(5.0 % 2.0)
println(5.0 == 5.0)
println(5.0 != 5.0)
println(2.0 > 5.0)
println(2.0 < 5.0)
println(3.0 >= 2.0)
println(3.0 <= 2.0)
# Mixed section
println(5.5 < 10.10)
println(2 < 5.5)
println(2 < 5L)
println(true or false)
println(true and true)
println(true and false)
println(false or false)
println((2 < 5L) and (5.0 > 1))
println(not not (true and false))
println("hello" + "world" + "we" + "are" + "boomslang")
# Test overflows
println(5000000000 + 5000000000)
println(5000000000L + 5000000000L)
# Test equality
int x = 1
println("batch1")
println(x == NULL)
println("hello" == "HELLO")
println("hello" == "hello")
println(NULL == NULL)
println(NULL != NULL)
println("batch2")
println(NULL != 0.0)
println(0 == NULL)
println(false == NULL)
println(true == NULL)
println(0L == NULL)
println("batch3")
println('a' == 'a')
println('a' != 'b')
println('a' != 'a')
println('c' == NULL)
println('c' == "c")  # this is true due to coercion

# Test some crazy stuff
int i = (int x = 5)
println(i)
println(x)

int a = 5
int b = (a = 1)
println(a)
println(b)
```

test-arith4.out:

```
NULL
true
false
50
2.2000
2
hello
c
helloc
truehello
1hello
2.2000hello
hello5
helloNULL
helloworld
hello NULL! 2+2 is equal to 4
false
true
false
-2
-2.2000
-5
4
3
25
10
1
true
false
false
true
true
false
4
3
25
10
1
true
false
false
true
true
false
4.0000
3.0000
25.0000
10.0000
1.0000
true
```

```
false
false
true
true
false
true
true
true
true
true
false
false
true
false
helloworldweareboomslang
1410065408
10000000000
batch1
false
false
true
true
false
batch2
true
false
false
false
false
batch3
true
true
false
false
true
5
5
1
1
```

test-arithmetic_passes.boom:

```
int y = 2 + 3 * 5 / 4
println(y)
```

test-arithmetic_passes.out:

```
5
```

test-arraylist.boom:

```
class ArrayList[T]:
optional:
int size = 0
T[] arr = default T[10]

def add(T element):
if self.size >= len(self.arr):
# TODO dynamic sized arrays are needed for this
#T[] copy = default T[len(self.arr) * 2]
T[] copy = default T[100]
int i = 0
loop i+=1 while i < len(self.arr):
copy[i] = self.arr[i]
self.arr = copy
self.add(element)
else:
self.arr[self.size] = element
self.size += 1

def get(int i) returns T:
return self.arr[i]

class StringArrayList = ArrayList(string)

StringArrayList list = StringArrayList()
int i = 0
loop i+=1 while i < 20:
list.add("string" + i)

println(list.size)
int i = 0
loop i+=1 while i < list.size:
println(list.get(i))
```

test-arraylist.out:

```
20
string0
string1
string2
string3
string4
string5
string6
string7
```

```
string8
string9
string10
string11
string12
string13
string14
string15
string16
string17
string18
string19
```

test-arrays1.boom:

```
int[] arr = default int[10]
int i = 0
loop i += 2 while i < 10:
arr[i] = i
i = 0
loop i += 1 while i < 10:
println(arr[i])

arr = [0,9,8,7,6,5,4,3,2,1]
println("")
println(arr[7])

def give_me_an_array() returns string[]:
return ["hi","I","love","PLT","!"]

string[] foo = give_me_an_array()
string[] bar = give_me_an_array()

println("")
i = 0
loop i += 1 while i < 5:
println(foo[i])
```

test-arrays1.out:

```
0
0
2
0
4
0
6
0
```

```
8
0

3

hi
I
love
PLT
!
```

test-arrays2.boom:

```
int[] arr = [3+2, exp(5,2), 42]

print_arr3(arr)

# this function calculates b^p
def exp(int b, int p) returns int:
if p == 0:
return b
int start = 1
int i = 0
loop i += 1 while i < p:
start *= b
return start

# this function prints out an array of size 3
def print_arr3(int[] a) returns void:
int i = 0
loop i += 1 while i < 3:
println(arr[i])

# make everyting in arr default
arr = default int[3]
println("")
print_arr3(arr)
```

test-arrays2.out:

```
5
25
42

0
0
0
```

test-arrays-in-classes-methods.boom:

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

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

Family doeFamily = Family([["Typo","Stephanie","Jim","Kate"], ["father","mother","son","daughter"]])
doeFamily.updateFathersName("John")
int i = 0
loop i += 1 while i < 4:
println(doeFamily.relation[0][i] + " is " + doeFamily.relation[1][i])
```

test-arrays-in-classes-methods.out:

```
John is father
Stephanie is mother
Jim is son
Kate is daughter
```

test-arrays-in-classes-optional.boom:

```
class MyObject:
optional:
int[] arr = [1,2,3]

MyObject a = MyObject([0,1,2])

int i = 0
loop i += 1 while i < 3:
println(a.arr[i])
```

test-arrays-in-classes-optional.out:

```
0
1
2
```

test-arrays-in-classes-required.boom:

```
class MyObject:

required:
int[] arr

MyObject a = MyObject([0,1,2])

int i = 0
loop i += 1 while i < 3:
println(a.arr[i])
```

test-arrays-in-classes-required.out:

```
0
1
2
```

test-arrays-in-classes-static.boom:

```
class MyObject:
static:
int[] arr = [0,1,2]

MyObject a = MyObject()
int i = 0
loop i += 1 while i < 3:
println(a.arr[i])
```

test-arrays-in-classes-static.out:

```
0
1
2
```

test-arrays-reassignment.boom:

```
int[] arr = [1,2,3]
int[] arrCopy = [2,3,4]
arrCopy = arr
```

```
int i = 0
loop i += 1 while i < 3:
println(arrCopy[i])
```

test-arrays-reassignment.out:

```
1
2
3
```

test-array-default-multidim.boom:

```
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])
```

test-array-default-multidim.out:

```
0
false
0
false
0
false
0
false
0
false
0
false
```

test-array-equality1.boom:

```
int[][] a = [[1,2], [3,4]]
int[][] b = [[1,2], [3,4]]
# this should not be equal
# despite the elements being the same it is a different pointer
println(a == b)
# these should be the same
println(a == a)
println(b == b)
```

test-array-equality1.out:

```
false
true
true
```

test-array_access_passes.boom:

```
int[] array = [0,1,2]
int x = array[2]
println(x)
```

test-array_access_passes.out:

```
2
```

test-array_declaration_passes_1.boom:

```
int[] array = [1, 2, 3, 4, 5, 6]

int i=0
loop i=i+1 while i<6:
println(array[i])
```

test-array_declaration_passes_1.out:

```
1
2
3
4
5
6
```

test-array_declaration_passes_2.boom:

```
[]
```

test-array_declaration_passes_2.out:

test-array_declaration_passes_3.boom:

```
[[1, 2, 3]]
int[][] y = [[1, 2, 3]]
int i = 0
loop i+=1 while i<3:
println(y[0][i])
```

test-array_declaration_passes_3.out:

```
1
2
3
```

test-array_update_succeeds.boom:

```
int[] arr = [1, 2]
arr[0] += 5

println(arr[0])
```

test-array_update_succeeds.out:

```
6
```

test-assigning_to_array_index_succeeds.boom:

```
string[] arr = ["one", "two", "three"]
arr[1] = "newvalue"
int i = 0
loop i=i+1 while i<3:
println(arr[i])
```

```
test-assigning_to_array_index_succeeds.out:

one
newvalue
three
```

```
test-assignment_without_type_passes.boom:

int x = 0
x = 5
println(x)
```

```
test-assignment_without_type_passes.out:

5
```

```
test-auto-tostring1.boom:

class MyGeneric[T]:
static:
T static1 = 0
string static2 = "hey"

required:
float x

optional:
char c = 'h'
int[][] arr = default int[2][5]

class MyClass = MyGeneric(int)
MyClass instance = MyClass(2.0)
println(instance)

MyClass instance2 = NULL
```

```
println(instance2)
```

test-auto-tostring1.out:

```
MyClass:
static1:0
static2:hey
x:2.0000
c:h
arr:Array

NULL
```

test-class_declaration_1.boom:

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


required:
int z
float fOOOO



optional:
boolean boo = true

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

test-class_declaration_1.out:

```
10
1.0000
```

test-class_declaration_2.boom:

```
class MyObject:
```

```
static:
int x = 5
string foo = "bar"


required:
int z
float fOOOO



optional:
boolean boo = true

MyObject foo = MyObject(10,1.0)
println(foo.boo)
println(foo.foo)
println(foo.x)
```

```
test-class_declaration_2.out:

true
bar
5
```

```
test-class_declaration_3.boom:

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


required:
int z
float fOOOO



optional:
boolean boo = true


def mymethod():
self.x = 5

def _++%%(MyObject other) returns MyObject:
return MyObject(1, 5.0)
```

```
MyObject apple = MyObject(10, 1.0, false)
MyObject banana = MyObject(5, 7.0, true)

MyObject appana = apple ++%% banana

println(appana.fOOOO)




test-class_declaration_3.out:

5.0000




test-class_declaration_4.boom:

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


required:
int z
float fOOOO



optional:
boolean boo = true


def mymethod():
self.x = 10








def _++%%(MyObject other) returns MyObject:
return MyObject(1, 5.0, false)
def mymethod2(int x) returns int:
return 5

MyObject mo = MyObject(1,1.0)
int x = mo.mymethod2(7)
println(x)
```

test-class_declaration_4.out:

```
5
```

test-class_declaration_5.boom:

```
class MyObject:
def _++%%(MyObject other) returns MyObject:
return MyObject()
def mymethod2(int x) returns int:
return 5

MyObject a = MyObject()
MyObject b = MyObject()

MyObject x = a ++%% b

println(x.mymethod2(4))
```

test-class_declaration_5.out:

```
5
```

test-class_declaration_6.boom:

```
class MyObject:
static:
int x = 5

MyObject foo = MyObject()
println(foo.x)
```

test-class_declaration_6.out:

```
5
```

test-class_declaration_7.boom:

```
class MyObject:
required:
int x

MyObject foo = MyObject(1)
println(foo.x)
```

test-class_declaration_7.out:

```
1
```

test-class_declaration_8.boom:

```
class MyObject:
optional:
int x = 5
MyObject mo = MyObject()
println(mo.x)
```

test-class_declaration_8.out:

```
5
```

test-class_declaration_9.boom:

```
class MyObject:
static:
int x = 5
required:
int y

println(MyObject(10).y)
```

test-class_declaration_9.out:

```
10
```

```
test-class_declaration_10.boom:

class MyObject:
static:
int x = 5
optional:
int y = 5

MyObject mo = MyObject()
println(mo.y)
MyObject no = MyObject(7)
println(no.y)
```

```
test-class_declaration_10.out:

5
7
```

```
test-class_declaration_11.boom:

class MyObjectTwo:
required:
string x


optional:
string z = "foo"
MyObjectTwo ot = MyObjectTwo("two")
println(ot.z)
println(ot.x)
MyObjectTwo ot = MyObjectTwo("two","foo2")
println(ot.z)
println(ot.x)
```

```
test-class_declaration_11.out:

foo
two
foo2
two
```

test-complicated_program_1.boom:

```
int x = 5
def myfunc(int x, float foo) returns string:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
elif x > 20:
println("hey")
elif x > 30:
println("hey")
println("hey")
else:
println("hey")
return "hey"



int x = 50

string str = myfunc(32, 234.8)
println(str)
```

test-complicated_program_1.out:

```
hey
```

test-complicated_program_2.boom:

```
int x = 5
def myfunc(int x, float foo) returns string:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
return "hey"
elif x > 20:
println("hey")
return "hey"
elif x > 30:
println("hey")
println("hey")
return "hey"
```

```
else:
println("hey")
return "hey"
else:
return "hey"


def myfunc2(int x, float foo) returns string:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
elif x > 20:
println("hey")
elif x > 30:
println("hey")
println("hey")
else:
println("hey")
return "hey"


string str1 = myfunc(x, 5234.2425)
println(str1)
int x = 50
string str2 = myfunc2(x, 5345.3)
println(str2)
```

test-complicated_program_2.out:

```
hey
hey
```

test-complicated_program_3.boom:

```
int x = 5
def myfunc(int x, float foo) returns void:
int y = 5
int z = 7
if x > 5:
if x > 10:
int z = 20
println(z)
elif x > 20:
println("hey")
elif x > 30:
println("hey")
println("hey")
else:
println("hey")
println(z)

myfunc(15,20.83)
myfunc(7,20.76)
```

```
test-complicated_program_3.out:

20
hey
7
```

```
test-constructor_overwriting_is_allowed.boom:

class MyClass:
required:
int x
def construct(int x):
println("i am overwriting the default constructor")

MyClass foo = MyClass(5)
```

```
test-constructor_overwriting_is_allowed.out:

i am overwriting the default constructor
```

```
test-double_eq.boom:

int x = 2
println(3 == x)
x = 3
println(3 == x)
```

```
test-double_eq.out:

false
true
```

```
test-empty-arrays1.boom:

def myfunc(int[] x):
println("this works now")

myfunc([])

def myfunc(int[][] x):
println("this also works")

myfunc([[], [], [], []])
myfunc([[1,2], [1,2], [1,2]    ])

int[][] arr1 = default int[0][0]
int[][] arr2 = [[]]
println(arr1)
println(arr2)
```

```
test-empty-arrays1.out:

this works now
this also works
this also works
Array
Array
```

```
test-empty_program_passes.boom:
```

```
test-empty_program_passes.out:
```

```
test-float1.boom:

float a = 3.14159267
println(a)
```

```
test-float1.out:

3.1416
```

```
test-float2.boom:

float a = 3.14159267
float b = -2.71828
float c = a + b
println(c)
```

```
test-float2.out:

0.4233
```

```
test-float3.boom:

def testfloat(float a, float b) returns void:
println(a + b)
println(a - b)
println(a * b)
println(a / b)
println(a == b)
println(a == a)
println(a != b)
println(a != a)
println(a > b)
println(a >= b)
println(a < b)
println(a <= b)

float c = 42.0
float d = 3.14159
testfloat(c, d)
testfloat(d, d)
```

```
test-float3.out:

45.1416
38.8584
```

```
131.9468
13.3690
false
true
true
false
true
true
false
false
6.2832
0.0000
9.8696
1.0000
true
true
false
false
false
true
false
true
```

test-func1.boom:

```
def add(int a, int b) returns int:
return a + b

int a = add(39, 3)
println(a)
```

test-func1.out:

```
42
```

test-func2.boom:

```
def fun(int x, int y) returns int:
return 0

int i = 1
fun(i = 2, i = i+1)
println(i)
```

test-func2.out:

```
3
```

test-func3.boom:

```
def printem(int a, int b, int c, int d) returns void:
println(a)
println(b)
println(c)
println(d)

printem(42,17,192,8)
```

test-func3.out:

```
42
17
192
8
```

test-func4.boom:

```
def add(int a, int b) returns int:
int c = a + b
return c

int d = add(52, 10)
println(d)
```

test-func4.out:

```
62
```

test-func5.boom:

```
def foo(int a) returns int:
```

```
return a

def main() returns int:
return 0
```

test-func5.out:

test-func6.boom:

```
def bar(int a, boolean b, int c) returns int:
return a + c

println(bar(17, false, 25))
```

test-func6.out:

```
42
```

test-func7.boom:

```
int a = 0

def foo(int c) returns void:
a = c + 42

foo(73)
println(a)
```

test-func7.out:

```
115
```

test-func8.boom:

```
def foo(int a) returns void:
```

```
println(a + 3)

foo(40)
```

test-func8.out:

```
43
```

test-func9.boom:

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

foo(40)
```

test-func9.out:

```
43
```

test-func10.boom:

```
def factorial(int n) returns int:
if n < 0:
println("invalid input!")
return -1
if n == 0:
return 1
return n * factorial(n - 1)

println(factorial(5))
println(factorial(0))
println(factorial(-20))
```

test-func10.out:

```
120
1
invalid input!
```

```
-1
```

test-func11.boom:

```
def printfib(int n) returns void:
int i = 0
int prev = 1
int curr = prev
loop i += 1 while i < n:
if (i == 0) or (i == 1):
println(1)
else:
int tmp = curr
curr += prev
prev = tmp
println(curr)

printfib(10)
```

test-func11.out:

```
1
1
2
3
5
8
13
21
34
55
```

test-func12.boom:

```
def myadd(int a, int b) returns int:
return a + b

def myadd(int a, int b, int c) returns int:
return a + b + c

println(myadd(myadd(4, 5), myadd(5, 7, 2)))
```

test-func12.out:

```
23
```

test-func13.boom:

```
int x = myfunc("foo")
println(x)
println(myfunc("bar"))

def myfunc(string x) returns int:
int y = 500
if y == 500:
if y == 500:
/# just throwing a comment in

to make sure
comments are working inside functions
#/
if y == 500:
loop y+=1 while y < 600:
return y
return -1

def myotherfunctuion(int a, int b) returns string:
# here's a comment for no real reason
return "unused"
```

test-func13.out:

```
500
500
```

test-func14.boom:

```
int x = myfunc(get_a_string())
println(x)
println(myfunc(get_a_string()))

def get_a_string() returns string:
return "heres a string"

def myfunc(string x) returns int:
int y = 500
if y == 500:
```

```
if y == 500:
/#
another pointless comment

#/
if y == 500:
loop y+=1 while y < 600:
if y > 550:
int z = y
return z
elif y == 550:
int z = y
return z
elif y == 550:
return y
return -1

def myotherfunctuion(int a, int b) returns string:
# here's a comment for no real reason
return "unused"
```

test-func14.out:

```
550
550
```

test-func15.boom:

```
# test mutual recursion for functions

println(a(0))

def a(int x) returns int:
int z = 5 + 5
if x > 5:
return 5
else:
return b(x+1)

def b(int x) returns int:
if x > 5:
return 10
else:
return a(x+1)
```

test-func15.out:

```
5
```

test-func16.boom:

```
int x = myfunc(get_a_string())
println(x)
println(myfunc(get_a_string()))

def get_a_string() returns string:
return "heres a string"

def myfunc(string x) returns int:
int y = 500
if y == 500:
if y == 500:
/#
another pointless comment

#/
if y == 500:
loop y+=1 while y < 600:
if y > 550:
int z = y
return z
elif y == 551:
int z = y
elif y == 550:
return y
else:
2+2
return -1

def myotherfunctuion(int a, int b) returns string:
# here's a comment for no real reason
return "unused"
```

test-func16.out:

```
550
550
```

test-func17.boom:

```
int x = myfunc(get_a_string())
println(x)
println(myfunc(get_a_string()))

def get_a_string() returns string:
return "heres a string"

def myfunc(string x) returns int:
int y = 500
if y == 500:
if y == 500:
/#
another pointless comment

#/
if y == 500:
loop y+=1 while y < 600:
if y > 550:
int z = y
return z
elif y == 550:
int z = y
return z
elif y == 550:
return y
else:
return y
return -1

def myotherfunctuion(int a, int b) returns string:
# here's a comment for no real reason
return "unused"
```

test-func17.out:

```
500
500
```

test-gcd.boom:

```
def gcd(int a, int b) returns int:
loop while a != b:
if (a > b):
a = a - b
else:
b = b - a
return a
```

```
println(gcd(2,14))
println(gcd(3,15))
println(gcd(99,121))
```

test-gcd.out:

```
2
3
11
```

test-generics1.boom:

```
class KeyValue[K, V]:
required:
K key
V value

def get_key() returns K:
return self.key

def get_value() returns V:
return self.value

def get_key(V value) returns K:
return self.key

def get_value(K key) returns V:
return self.value

class StringIntKeyValue = KeyValue(string, int)

StringIntKeyValue kv1 = StringIntKeyValue("hey1", 1)
StringIntKeyValue kv2 = StringIntKeyValue("hey2", 2)
IntStringKeyValue kv3 = IntStringKeyValue(3, "reversed1")
IntStringKeyValue kv4 = IntStringKeyValue(4, "reversed2")
println(kv1.key)
println(kv1.value)
println(kv1.get_key())
println(kv1.get_value())
println(kv1.get_key(kv1.value))
println(kv1.get_value(kv1.key))

println(kv2.key)
println(kv2.value)
println(kv2.get_key())
println(kv2.get_value())
println(kv2.get_key(kv2.value))
println(kv2.get_value(kv2.key))

println(kv3.key)
println(kv3.value)
println(kv3.get_key())
println(kv3.get_value())
println(kv3.get_key(kv3.value))
println(kv3.get_value(kv3.key))

println(kv4.key)
println(kv4.value)
println(kv4.get_key())
println(kv4.get_value())
println(kv4.get_key(kv4.value))
println(kv4.get_value(kv4.key))

class IntStringKeyValue = KeyValue(int, string)
```

test-generics1.out:

```
hey1
1
hey1
1
hey1
1
hey2
2
hey2
2
hey2
2
3
reversed1
3
reversed1
3
reversed1
4
reversed2
4
reversed2
4
reversed2
```

test-generics2.boom:

```
class LinkedList[T]:
```

```
required:
T element

optional:
LinkedList next = NULL

def _+(LinkedList other) returns LinkedList:
# append the other list to this one
if self.next == NULL:
self.next = other
return self
else:
self.next + other
return self

def _+(T new_element) returns LinkedList:
if self.next == NULL:
self.next = LinkedList(new_element, NULL)
return self
else:
self.next + LinkedList(new_element, NULL)
return self

def _++(LinkedList other):
if self.next == NULL:
self.next = other
else:
self.next ++ other

def _++(T new_element):
if self.next == NULL:
self.next = LinkedList(new_element, NULL)
else:
self.next ++ LinkedList(new_element, NULL)

def _==(LinkedList other) returns boolean:
# Custom defined equality function
if self == NULL:
return other == NULL
elif other == NULL:
return false
else:
return (self.element == other.element) and (self.next == other.next)

def size() returns int:
int size = 1
LinkedList next_temp = self.next
loop while (next_temp != NULL):
size += 1
next_temp = next_temp.next
return size

def to_string() returns string:
string temp = ""
```

```
if self == NULL:
return "NULL"
elif self.next == NULL:
return temp + self.element
else:
return temp + self.element + " -> " + self.next.to_string()

class IntLinkedList = LinkedList(int)

IntLinkedList mylist = IntLinkedList(1)
mylist + 2 + 3 + 4
println(mylist.size())
println(mylist)
println(mylist.next)
println(mylist.next.next)
println(mylist.next.next.next)

IntLinkedList nulllist = mylist.next.next.next.next
println(nulllist)
println(mylist.next.next.next.next)

class FloatLinkedList = LinkedList(float)
FloatLinkedList mylist2 = FloatLinkedList(1.0)
mylist2 + 2.0
println(mylist2.size())
println(mylist2.to_string())

FloatLinkedList mylist3 = FloatLinkedList(3.0, NULL)
mylist2 + mylist3
println(mylist2.size())
println(mylist2)

# Check equality
IntLinkedList int2 = IntLinkedList(1)
int2 ++ 2
int2 ++ 3
int2 ++ 4
println(mylist == int2) # should be true
IntLinkedList int3 = IntLinkedList(1) + 2 + 3
println(mylist == int3) # should be false
int3 = int3 + 4
println(mylist == int3) # should be true
```

test-generics2.out:

```
4
1 -> 2 -> 3 -> 4
2 -> 3 -> 4
3 -> 4
4
```

```
NULL
NULL
2
1.0000 -> 2.0000
3
1.0000 -> 2.0000 -> 3.0000
true
false
true
```

test-hello.boom:

```
println("Hello, world! Welcome to Boomslang.")
```

test-hello.out:

```
Hello, world! Welcome to Boomslang.
```

test-if1.boom:

```
int x = (2 + 2) * 2
println("this is the first thing")
#string unused = "unused"
string unused = "unused"
if x == 8:
int unused = 0
println("this is nested, and correct"+unused)
int z = x + 1
if z > 100:
# comment
println("this should not be hit")
println("nor should this")
println("this should be hit - elseless if")
if z < 20:
/# multiline
comment
with crazy tabs


                   and non-tab stuff
       #/
println("here is another nested if")
if z < 15:
println("but we can go deeper")
else:
```

```
println("this is an else that won't be hit")
if z == 9:
println("this won't be hit because it's inside the else")
println("this should be printed too because it's not in the else")
elif z == 9:
     # comment before
/# another multi-line #/
#comment after
#one more
float unused = 900.0
   /#
a really weird one
     #/
println("this is an elif that shouldnt be hit")
#comment
elif z == 9:
println("this is true but won't be hit")
elif z < 10:
println("this is true but won't be hit")
elif z > 100:
println("this is false but won't be hit")
println("above we had an if elifs with no else, which is fine")
if x == 8:
println("now let's try if/elif/else")
elif x == 8:
println("ignored")
elif x == 8:
println("ignored")
else:
println("ignored")
if x == 8:
if x == 8:
if x == 8:
if x == 8:
if x == 8:
if x == 9:
println("x was 9")
else:
println("this was nested very deep but who cares?")
else:
println("this else does not matter")


if true:
println("true")
if false:
println("false")
if false:
println("false")
else:
println("true")
if false:
println("false")
elif true:
```

```
println("true")
else:
println("false")

if (2+2+2+2) == 2 * 4 and 4 == 4:
println("this should be printed")

println("here is some more stuff")
println("more junk")
string foo = "bar" + "baz"
println(foo)
```

test-if1.out:

```
this is the first thing
this is nested, and correct0
this should be hit - elseless if
here is another nested if
but we can go deeper
this should be printed too because it's not in the else
above we had an if elifs with no else, which is fine
now let's try if/elif/else
this was nested very deep but who cares?
true
true
true
this should be printed
here is some more stuff
more junk
barbaz
```

test-if2.boom:

```
if (true):
println(42)
else:
println(8)
println(17)
```

test-if2.out:

```
42
17
```

test-if3.boom:

```
if (false):
println(42)
println(17)
```

test-if3.out:

```
17
```

test-if4.boom:

```
if (false):
println(42)
else:
println(8)
println(17)
```

test-if4.out:

```
8
17
```

test-if5.boom:

```
def cond(boolean b) returns int:
int x = 0
if (b):
x = 42
else:
x = 17
return x

println(cond(true))
println(cond(false))
```

test-if5.out:

```
42
17
```

test-if6.boom:

```
def cond(boolean b) returns int:
int x = 10
if (b):
if (x == 10):
x = 42
else:
x = 17
return x

println(cond(true))
println(cond(false))
```

test-if6.out:

```
42
10
```

test-local2.boom:

```
def foo(int a, boolean b) returns int:
int c = a
boolean d = false
return c + 10

println(foo(37, false))
```

test-local2.out:

```
47
```

test-long_initialization_passes.boom:

```
long lo = 500000000000L
```

```
println(lo)
```

test-long_initialization_passes.out:

```
500000000000
```

test-loop1.boom:

```
int i = 0
loop i+=1 while i<5:
println(i)
println(42)
```

test-loop1.out:

```
0
1
2
3
4
42
```

test-loop2.boom:

```
int i = 0
if true:
if true:
if not false:
loop while i<5:
# comment
println(i)
i += 1

int a = 0
int b = 0
loop a+=1 while a<3:
loop b+=1 while b<3:
println(a + " " + b)
b = 0
println(42)
```

test-loop2.out:

```
0
1
2
3
4
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
42
```

test-loop_1.boom:

```
int x = 0
loop x+=1 while x<100:
println("hey")
```

test-loop_1.out:

```
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
```

```
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
```

```
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
hey
```

test-math.boom:

```
def gcd(int a, int b) returns int:
loop while a != b:
if a > b:
a = a - b
else:
b = b - a
return a

def pow(int a, int b) returns int:
if b == 0:
return 1
if b == 1:
return a
int i = 0
int ans = 1
loop i+=1 while i<b:
ans *= a
return ans

println(gcd(10, 10))
println(gcd(100, 15))
println(pow(2, 2))
println(pow(3, 3))
println(pow(3, 0))
println(pow(100, 1))
```

test-math.out:

```
10
5
4
27
1
100
```

test-multi_comments.boom:

```
/#comments
comments#/
```

test-multi_comments.out:

test-nested-array-literals1.boom:

```
class MyObject:
static:
int x = 0


println([ [NULL, NULL], [MyObject(), MyObject()] ])
```

test-nested-array-literals1.out:

```
Array
```

test-nested-array-literals2.boom:

```
println([ [[1, 2], [1, 2], [1, 2]]      ,      [[1, 2],[1, 2],[1, 2]]      ,      [[1, 2],[1, 2],[1, 2]
```

test-nested-array-literals2.out:

```
Array
```

test-newlines_1.boom:

test-newlines_1.out:

test-newlines_2.boom:

test-newlines_2.out:

test-newlines_3.boom:

```
int x = 5

int y = 6
```

test-newlines_3.out:

test-null_assignment.boom:

```
class MyObject:
def myfunc():
println("hey")

MyObject x = NULL
x.myfunc()
```

test-null_assignment.out:

```
hey
```

test-objects1.boom:

```
class MyClass:
#adding some comments
required:
int one
# adding some comments
string two
MyClass recursive1

optional: /# more comments #/
float three = 3.0
float four = 4.0
float five  = 5.0 # adding some comments
char six = '6'
MyClass recursive2 = MyClass() # adding some comments

def construct(): /# here
are some more comments designed to be annoying #/
println("called custom constructor")

def to_string() returns string: /#
 another one #/
```

```
# adding some comments
return "this is my custom tostring"
# adding some comments
def method1(int a, int b, int z) returns int:
println("called method1") /#

more    stupid
comments
#/
int q = a + b + z
int z = q
return z

# adding some comments
def method2():
    # adding some comments
println("called method2")

println("printing some random stuff")
MyClass foo = MyClass(1, "two", MyClass())
println(foo)
println(foo.one)
println(foo.two)
println(foo.three)
println(foo.four)
println(foo.five)
println(foo.six)
MyClass recursive1 = foo.recursive1
MyClass recursive2 = foo.recursive2
println(recursive1)
println(recursive2.to_string())
println(foo.method1(1, 2, 3))
foo.method2()
```

test-objects1.out:

```
printing some random stuff
called custom constructor
called custom constructor
this is my custom tostring
1
two
3.0000
4.0000
5.0000
6
this is my custom tostring
this is my custom tostring
called method1
6
called method2
```

test-objects2.boom:

```
class MyClass:
#adding some comments
static:
string staticx = "foo"
int staticy = 5

required:
int one
# adding some comments
string two
MyClass recursive1

optional:
float three = 3.0
float four = 4.0
float five  = 5.0 # adding some comments
char six = '6'
MyClass recursive2 = MyClass() # adding some comments

def construct():
println("called custom constructor")

def to_string() returns string:
# adding some comments
return "this is my custom tostring"
# adding some comments
def method1(int a, int b, int z) returns int:
println("called method1")
int q = a + b + z
int z = q
return z

def return_self() returns MyClass:
return self

# adding some comments
def method2():
    # adding some comments
println("called method2")

println("printing some random stuff")
MyClass foo1 = MyClass(1, "two", MyClass())
println(foo1)
println(foo1.one)
println(foo1.two)
println(foo1.three)
println(foo1.four)
println(foo1.five)
println(foo1.six)
MyClass recursive1 = foo1.recursive1
MyClass recursive2 = foo1.recursive2
println(recursive1)
println(recursive2.to_string())
println(foo1.method1(1, 2, 3))
foo1.method2()

MyClass foo2 = MyClass(1, "two", MyClass())
println(foo2.staticx)
println(foo2.staticy)
foo1.staticx = "new value"
foo2.staticy = 5000
println(foo1.staticy)
println(foo2.staticx)
println(MyClass.staticx)
println(MyClass.staticy)
MyClass.staticx = "successfully updated static var"
MyClass.staticy *= 2
println(MyClass.staticx)
println(MyClass.staticy)
MyClass copy = foo1.return_self()
println(copy)
```

```
test-objects2.out:

printing some random stuff
called custom constructor
called custom constructor
this is my custom tostring
1
two
3.0000
4.0000
5.0000
6
this is my custom tostring
this is my custom tostring
called method1
6
called method2
called custom constructor
called custom constructor
foo
5
5000
new value
new value
5000
successfully updated static var
10000
```

```
this is my custom tostring
```

test-objects3.boom:

```
class MyClassO:
#adding some comments
static:
MyClassT myclass2 = MyClassT(1, MyClassO(), 4)
int staticy = 5

def return_self() returns MyClassO:
return self

class MyClassT:
required:
int q
MyClassO other

optional:
int x = 6

def func(int unused) returns int:
if self.x > 5:
loop self.x+=1 while self.x<10:
println(self.x)
return self.x
return 10000

def func(int unused) returns int:
return 12345

class WeirdClass:
required:
int x
def construct() returns void:
println("i forgot to set x - what will happen?")
println(self.x)

MyClassO object1 = MyClassO()
MyClassT object2 = MyClassT(1, object1.return_self())
println(object2.func(12345))
MyClassT object3 = object1.myclass2
println(object3.func(12345))

WeirdClass weird_class = WeirdClass()
println(weird_class.x)
```

test-objects3.out:

```
6
7
8
9
10
10000
i forgot to set x - what will happen?
0
0
```

test-objects4.boom:

```
class MyClass:
required:
MyClass myclass
int x

def construct():
# do nothing
2 + 2

def getself() returns MyClass:
return self

def printx():
println(self.x)

def to_string() returns string:
return "this is my custom to string"



MyClass innermost = MyClass()
MyClass foo = MyClass(MyClass(MyClass(MyClass(MyClass(MyClass(innermost, 1), 2), 3), 4), 5), 6)
println(foo)
println(foo.myclass.myclass.getself().getself().getself().myclass.x)
println(foo.getself().getself().x)
println(foo.getself().myclass.myclass.getself().myclass)
println(foo.myclass.myclass)
foo.getself().getself().printx()

println(MyClass())
println(MyClass().x)
MyClass().printx()
println(MyClass().to_string())
```

```
test-objects4.out:

this is my custom to string
3
6
this is my custom to string
this is my custom to string
6
this is my custom to string
0
0
this is my custom to string
```

```
test-objects5.boom:



class MyClass:
required:
MyClass myclass
int x

def getself(MyClass myclass) returns MyClass:
return self

def printx(MyClass myclass, MyClass myclass2):
println(self.x)

def to_string() returns string:
if self == NULL:
return "NULL"
else:
return "this is my custom to string"



MyClass innermost = MyClass(NULL, 0)
MyClass foo = MyClass(MyClass(MyClass(MyClass(MyClass(MyClass(innermost, 1), 2), 3), 4), 5), 6)
println(foo)
println(foo.myclass.myclass.getself(NULL).getself(NULL).getself(NULL).myclass.x)
println(foo.getself(NULL).getself(NULL).x)
println(foo.getself(NULL).myclass.myclass.getself(NULL).myclass)
println(foo.myclass.myclass)
foo.getself(NULL).getself(NULL).printx(NULL, NULL)

println(MyClass(NULL, 0))
println(MyClass(NULL, 5000).x)
MyClass(NULL, 5001).printx(NULL, NULL)
println(MyClass(NULL, 0).to_string())
```

```
MyClass myobject = NULL
println(myobject)
MyClass myobject2 = MyClass(NULL, 0)
myobject2 = NULL
println(myobject2)
println(NULL == myobject)
println(myobject == NULL)
println(NULL == myobject2)
println(myobject2 == NULL)
println(myobject == myobject2)
```

test-objects5.out:

```
this is my custom to string
3
6
this is my custom to string
this is my custom to string
6
this is my custom to string
5000
5001
this is my custom to string
NULL
NULL
true
true
true
true
true
```

test-object_assignment_passes.boom:

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

def myfunc(int x) returns string:
return "hey"

MyObject foo = MyObject(2, myfunc(2+2))
```

test-object_assignment_passes.out:

test-object_function_call.boom:

```
class MyObject:
def myfunction(int a, string b, boolean c):
println("hey")

int a = 5
string b = "foo"
boolean c = false
MyObject myobject = MyObject()
myobject.myfunction(a, b, c)
```

test-object_function_call.out:

```
hey
```

test-object_variable_access.boom:

```
class MyObject:
static:
int x = 5

MyObject myobject = MyObject()
int a = myobject.x
println(a)
```

test-object_variable_access.out:

```
5
```

test-object_variable_assignment.boom:

```
class MyObject:
required:
boolean fOOo12345
```

```
MyObject myobject = MyObject(false)

println(myobject.fOOo12345)
myobject.fOOo12345 = true
println(myobject.fOOo12345)
```

test-object_variable_assignment.out:

```
false
true
```

test-object_variable_with_expression.boom:

```
class MyObject:
static:
boolean is_this_true = true

MyObject myobject = MyObject()
boolean x = true and myobject.is_this_true

println(x)
```

test-object_variable_with_expression.out:

```
true
```

test-objoperator_ambiguity_1.boom:

```
class Horse:
def _+%(Horse other) returns int:
return 5

Horse yak = Horse()
Horse saddle = Horse()
int winnie = 8 + yak +% saddle + 6
println(winnie)
```

test-objoperator_ambiguity_1.out:

```
19
```

test-objoperator_decl_1.boom:

```
class Horse:
def _+*(Horse other) returns Horse:
return Horse()
Horse yak = Horse()
Horse saddle = Horse()
Horse winne = yak+*saddle
```

test-objoperator_decl_1.out:

test-objoperator_decl_2.boom:

```
class Horse:
required:
int x
def _^&%$(Horse other) returns Horse:
return Horse(other.x+self.x)

def _$%(Horse other) returns Horse:
return Horse(other.x*self.x)

Horse yak = Horse(2)
Horse saddle = Horse(3)
Horse poop = Horse(5)
Horse winnie = yak ^&%$ saddle $% poop
println(winnie.x)
```

test-objoperator_decl_2.out:

```
25
```

test-ops1.boom:

```
println(1 + 2)
println(1 - 2)
println(1 * 2)
println(100 / 2)
println(99)
println(1 == 2)
println(1 == 1)
println(99)
println(1 != 2)
println(1 != 1)
println(99)
println(1 < 2)
println(2 < 1)
println(99)
println(1 <= 2)
println(1 <= 1)
println(2 <= 1)
println(99)
println(1 > 2)
println(2 > 1)
println(99)
println(1 >= 2)
println(1 >= 1)
println(2 >= 1)
```

test-ops1.out:

```
3
-1
2
50
99
false
true
99
true
false
99
true
false
99
true
true
false
99
false
true
99
false
true
true
```

test-ops2.boom:

```
println(true)
println(false)
println(true and true)
println(true and false)
println(false and true)
println(false and false)
println(true or true)
println(true or false)
println(false or true)
println(false or false)
boolean a = not false
println(a)
boolean b = not true
println(b)
println(-10)
int c = -42
int d = -c
println(d)
```

test-ops2.out:

```
true
false
true
false
false
false
true
true
true
false
true
false
-10
42
```

test-print-emoji.boom:

```
println("")
```

test-print-emoji.out:

test-print-int.boom:

```
println(2)
```

test-print-int.out:

```
2
```

test-self_access_with_field.boom:

```
class Foo:
optional:
string x = "str"

def myfunc() returns string:
return self.x

Foo f = Foo("hello")
println(f.myfunc())
```

test-self_access_with_field.out:

```
hello
```

test-self_access_with_function.boom:

```
class MyClass:

def myfunction(int x):
println("hey")

def myfunction2():
self.myfunction(2 % 3)
MyClass m = MyClass()
```

```
m.myfunction(6)
m.myfunction2()
```

```
test-self_access_with_function.out:

hey
hey
```

```
test-self_assignment_with_field.boom:

class MyClass:
required:
int x

def myfunc():
self.x = 5

MyClass a = MyClass(15)
println(a.x)
a.myfunc()
println(a.x)
```

```
test-self_assignment_with_field.out:

15
5
```

```
test-self_with_expression.boom:

class MyClass:
required:
int foo

def myfunc() returns int:
int x = self.foo * 5
return x

MyClass mc = MyClass(5)
println(mc.myfunc())
```

test-self_with_expression.out:

```
25
```

test-simple_assignment_passes_1.boom:

```
int x = 5

println(x)
```

test-simple_assignment_passes_1.out:

```
5
```

test-simple_assignment_passes_2.boom:

```
float yEs = .5

println(yEs)
```

test-simple_assignment_passes_2.out:

```
0.5000
```

test-simple_assignment_passes_3.boom:

```
def myfunc(int x) returns string:
return "hey"

string foo = myfunc(1+1+2+3+5)

println(foo)
```

test-simple_assignment_passes_3.out:

```
hey
```

test-simple_assignment_passes_4.boom:

```
float yEs = -.5

println(yEs)
```

test-simple_assignment_passes_4.out:

```
-0.5000
```

test-simple_func_print.boom:

```
def foo(int a) returns void:
println("hey")
return

foo(3)
```

test-simple_func_print.out:

```
hey
```

test-single_comments_1.boom:

```
#comments
```

test-single_comments_1.out:

test-single_comments_2.boom:

```
int x = 2
int foo = 2+2 #comment


int y = 2
```

test-single_comments_2.out:

test-single_comments_3.boom:

```
int x = 2
int foo = 2+2 # comment
int y = 2

println(x)
println(foo)
println(y)
```

test-single_comments_3.out:

```
2
4
2
```

test-string_esc_1.boom:

```
def print_extras(string str):
println(str)

print_extras("backslashes \o")
```

test-string_esc_1.out:

```
backslashes o
```

test-string_esc_2.boom:

```
println("\"")
```

test-string_esc_2.out:

```
"
```

test-string_esc_3.boom:

```
println("\\")
println("\\\\")
println("\\\\\\")
```

test-string_esc_3.out:

```
\
\\
\\\
```

test-string_esc_4.boom:

```
println("



")
```

test-string_esc_4.out:

test-string_esc_5.boom:

```
println("\n\n\n")
```

test-string_esc_5.out:

```
nnn
```

test-string_esc_6.boom:

```
println("hey i'm nikhil!!! && ** CAPS")
```

test-string_esc_6.out:

```
hey i'm nikhil!!! && ** CAPS
```

test-string_esc_7.boom:

```
println("\\n\\n\\n")
```

test-string_esc_7.out:

```
\n\n\n
```

test-update1.boom:

```
int x = 2
int y = 2
x += y
x *= y
x /= y
```

```
x -= y
x -= 4
println(x)
```

test-update1.out:

```
-2
```

test-valid_array_succeeds.boom:

```
def my_func(int x) returns int:
return x

int[] my_array = [1, (2+2), my_func(1), my_func(1) + 1, 2 * (3 + 1)]

int i = 0
loop i=i+1 while i<5:
println(my_array[i])
```

test-valid_array_succeeds.out:

```
1
4
1
2
8
```

test-valid_char_literal_1.boom:

```
char c = 'a'

println(c)
```

test-valid_char_literal_1.out:

```
a
```

test-valid_char_literal_2.boom:

```
char c = '''
```

test-valid_char_literal_2.out:

test-valid_construct_succeeds.boom:

```
class MyClass:
def print_5():
println("5")

MyClass my_object = MyClass()
MyClass x = my_object
x.print_5()
```

test-valid_construct_succeeds.out:

```
5
```

test-valid_divide_eq_with_object.boom:

```
class MyObject:
required:
int heY
MyObject myobject = MyObject(5)
println(myobject.heY /= 20)
```

test-valid_divide_eq_with_object.out:

```
0
```

test-valid_minus_eq.boom:

```
long fOo = 10L
fOo -= 500L
```

test-valid_minus_eq.out:

test-valid_neg_succeeds.boom:

```
int x = -1
x = -x
println(x)
```

test-valid_neg_succeeds.out:

```
1
```

test-valid_not_succeeds.boom:

```
boolean x = true
not x

println(not x)
println(x)
```

test-valid_not_succeeds.out:

```
false
true
```

test-valid_returns_succeeds.boom:

```
def my_func(int x) returns int:
if x > 5:
println("hey")
else:
return 5
return 1

println(my_func(7))

println(my_func(5))
```

```
test-valid_returns_succeeds.out:

hey
1
5
```

```
test-valid_self_return_succeeds.boom:

class MyClass:
def foo() returns MyClass:
return self
```

```
test-valid_self_return_succeeds.out:
```

```
test-valid_statement_without_newline_succeeds_1.boom:

println(2 + 3 * 5 / 4)
```

```
test-valid_statement_without_newline_succeeds_1.out:

5
```

test-valid_statement_without_newline_succeeds_2.boom:

```
println("hello world")
```

test-valid_statement_without_newline_succeeds_2.out:

```
hello world
```

test-valid_string_literal.boom:

```
string foo = "fooooo000oo !@$%^&*()_-+={}|[]:;/<,.>"

println(foo)
```

test-valid_string_literal.out:

```
fooooo000oo !@$%^&*()_-+={}|[]:;/<,.>
```

test-var1.boom:

```
int a = 42
println(a)
```

test-var1.out:

```
42
```

test-var2.boom:

```
int a = 0
def foo(int c) returns void:
a = c + 42
foo(73)
println(a)
```

test-var2.out:

```
115
```

test-var3.boom:

```
int a = 42
int a = 53
println(a)
```

test-var3.out:

```
53
```

test-while1.boom:

```
int i = 5
loop i -= 1 while i > 0:
println(i)
println(42)
```

test-while1.out:

```
5
4
3
2
1
42
```

test-while2.boom:

```
def foo(int a) returns int:
int j = 0
loop a -= 1 while a > 0:
j = j + 2
return j
```

```
println(foo(7))
```

```
test-while2.out:

14
```

11 Git Log

```
commit 7a133ceb920c1d02c2483bf74ce122305fa5c4cf
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Mon Apr 26 10:16:15 2021 -0700

    Add language tutorial to the main README (#114)

    * Update README.md

commit daf5e028ae3a975760090dbbf9d72d424608952d
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Mon Apr 26 01:53:58 2021 -0700

    Cleanup compiler warnings and add tests (#113)

    * fixed compile errors

    * made built in function map immutable

    * deleted [] tests

    * added more tests

commit c0b4ac81d3505bc87822e3859e4ed69e3c130c0b
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Mon Apr 26 00:59:30 2021 -0700

    Added node standard lib and tests (#112)

    * initial commit

    * thingy

    * remove tab

    * commit

    * finished tests

    * synced and added binarytree to stdlib
    * reformat

    Co-authored-by: nikhilmin-k <nmk2146@columbia.edu>
    Co-authored-by: Robert Kim <robert23kim@gmail.com>

commit e6fe4552a4bb4be9190ff9ee99aba8e198ef4101
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Apr 25 17:03:06 2021 -0700

    Add standard library, automated to strings, and tests (#111)

commit d8ce79d386b527f587f4ea2b155c639f1f157ccb
Author: robert23kim <robert23kim@gmail.com>
Date:   Sun Apr 25 17:47:39 2021 -0400

    Add tests for arrays (#110)

    * added basic arrays in classes tests

    * methods, two dim arr and default in classes

    * arrays wip

    * arrays wip: created arrays represented as structs

    * got array literal and array assign to work

    * implemented arrays in functions, array declaration, array access, array assginment

    * support for multidim arrays

    * len() for arrays wip

    * implemented len()

    * bug fixes

    * got default arrays to work for single arrays

    * default n-dim arrays work

    * added tests

    * some cleanup

    Co-authored-by: Nathan Cuevas <nathanjcuevas@gmail.com>

commit 35aa795b40c0286244123942ef69c1f30601e85c
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Sun Apr 25 14:35:42 2021 -0700

    Implemented most of the functionality for arrays (#109)

commit 890833731873ea9f798853c9b195585b9d6c6bd6
```

```
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Sun Apr 25 02:12:13 2021 -0700

    Update string literal parsing (#107)

    * Added divide by zero and null checks

    * initial commit

    * completed string character escape functionality

    Co-authored-by: David Steiner <dsteiner93@gmail.com>
    Co-authored-by: nikhilmin-k <nmk2146@columbia.edu>

commit 882415dbdfd5bdc39c83294269ca165c29e21cb5
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sat Apr 24 10:17:27 2021 -0700

    Added divide by zero and null checks (#106)

commit a03ea77b3ea8ad00e5ca2e839543dbe9cde8ac62
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Thu Apr 22 19:12:11 2021 -0700

    Add more tests and remove no longer needed Python test suite (#102)

    * added tests from test_move to a new branch of main, objoperators still ambiguous and arrays tests

    * added all tests from run_tests.py

    * added extra print statements to some tests, removed run_tests.py and Makefile pythontest command

    * changed all relevant tests to have a useful printable output

    * extra test updates

    * added print statement to every remaining relevant test

    * edited array tests

    * forgot a file

    Co-authored-by: nikhilmin-k <nmk2146@columbia.edu>

commit 9dea5e36f92f6a719c3e90b3ea3994b28c6fbcf1
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Thu Apr 22 13:30:20 2021 -0700

    Add support for generics in classes (#105)

    * Add support for generics in classes

    * Fix object operators and update tests
```

```
commit 62d919b5dbc3a56bb8e65aa1a7e299c8bb95ad54
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Wed Apr 21 19:03:19 2021 -0700

    Update arrays to be unsized and allow array access on exprs (#104)

commit 1d19f9a52a8a8b6fb1cc4da433cc557a0db1df65
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Wed Apr 21 10:12:21 2021 -0700

    Add support for arrays (#103)

commit c5c4ac2d12948d4eb357daf1d7b036158dc82037
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Tue Apr 20 17:32:43 2021 -0700

    Add basic support for nulls (#101)

commit 4c769c377e6373fc6fe9383a16d73e969bd3a76b
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Tue Apr 20 09:27:48 2021 -0700

    Enhance object support by allowing nested expressions instead of just identifiers (#99)

commit b67ede4e4a3fd8cacf004c094b595584c7b6f545
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Mon Apr 19 17:07:13 2021 -0700

    Move failure tests from Python to shell script suite (#98)

    * added run_test.py negative tests. positive tests are in the test_move repository and will be adde

    * finally completed with failure case tests

    * added failing string literal test

    * removed invalid string literal fails tests

    * merged with main and tested on parser/semant updates

    * removed moved tests from run_tests.py

    Co-authored-by: nikhilmin-k <nmk2146@columbia.edu>

commit 7bb22fdcb57326d5637e14fdfc632a66ed9218f2
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Apr 18 12:32:30 2021 -0700

    Add support for user defined classes (#87)

commit dc7ef4ce5fc6cdc8f999600c114b36a780a5f913
Author: robert23kim <robert23kim@gmail.com>
Date:   Sat Apr 17 13:22:03 2021 -0400
```

```
    Add more tests (#86)

    * added 10 more tests

    * finish up test success

    * added 11 more tests

    * added more tests

    * added remaining tests and fixed one return err msg

commit df4f77084d11b2ed114f22486bf6e28578e66cfb
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Mon Apr 12 17:30:52 2021 -0700

    Fix bugs with ifs and loops, add support for adding chars, and add tests (#84)

    * Fix bugs with ifs and loops, add support for adding chars, and add tests

commit 79881807312357c8c0737e017444fa26ce0d7470
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Sun Apr 11 21:54:21 2021 -0700

    Add support for functions (#83)

    * function decls wip

    * fdecls works for non-void functions

    * added support for fdecls, passes func all current func tests (1-9)

    * added more function tests (all pass) cleaned up codegen a little

    * can now call funcs before they are declared, fixed broken test scripts

    * made style changes per David's code review

commit 417f249adfb5fa572633c1ea70f70a8ec2b6d36e
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sat Apr 10 10:57:03 2021 -0700

    Add tests and reorganize test directory (#82)

commit ceba51004c4b656dcff00fd7464121b0c2bcf26e
Author: robert23kim <robert23kim@gmail.com>
Date:   Sat Apr 10 12:55:28 2021 -0400

    Implemented ifs and loops (#81)

    * debug

    * session with David if loops
    * Implemented loops, refactored, fixed bugs, added tests

    Co-authored-by: David Steiner <dsteiner93@gmail.com>

commit 25817ad7a3d1ea6e33139a99abfce9abcca9e2dc
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Thu Apr 8 09:58:03 2021 -0700

    Finish all literals, binops, unops, built-ins, and assignments (#80)

commit cb18dd34b4473600df85fc56d0b4e9114dc4b4b2
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Tue Apr 6 23:40:32 2021 -0700

    Added Basic Integer Arithmetic (#79)

    Co-authored-by: Nathan Cuevas <nathanjcuevas@gmail.com>
    Co-authored-by: nikhilmin-k <nmk2146@columbia.edu>
    Co-authored-by: David Steiner <dsteiner93@users.noreply.github.com>

commit b3bdc461e198c2406d35daa84e6fa0d882c899e6
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Tue Apr 6 17:18:04 2021 -0700

    Fix bug in semantic checker where statements were evaluated in wrong order (#78)

commit 00e9b7feeaf1a738e0811dc0edf6aff31063392d
Author: robert23kim <robert23kim@gmail.com>
Date:   Tue Apr 6 18:34:15 2021 -0400

    Test Suite Setup (#77)

    * started adding basic tests and set up shell script

    * added test files to tests folder

    * added about 10 more tests

    * minor changes

    * added helloworld.boom back

    * removed microc references

    * fixed print fail

    * added 9 more tests

    * changed error message

commit b88c255a051e23c5992d93871052af7f14ba15e3
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Sun Apr 4 21:51:37 2021 -0700
```

```
    Support for Library Function Calls (#76)

    * added boilerplate for FuncCall support

    * got println with strings to work again using more scaleable codegen architecture

    * made the builder functions take in a 'builder' param

    * added the built in function map

    * added support for println and int_to_string

    * put the library funcs in one .c file

commit f51bc1a384275a001730be1ee5dbf5d17a961f5e
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Apr 4 10:50:25 2021 -0700

    Add support for NULL and automatic to_string on classes (#75)

commit 4981232bd439304c7c70ad75f87129287240c72b
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sat Apr 3 17:07:43 2021 -0700

    Add support for autocasting, self, and simplify ast (#74)

    * Add support for autocasting, self, and simplify ast

    * Add basic support for nulls

commit ec5114860920a2e63c5637c8e1534c87ec832af7
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Fri Apr 2 16:01:13 2021 -0700

    Add nearly complete sast and semantic checker and fix some bugs (#61)

    * Add nearly complete sast and semantic checker and fix some bugs

    * Add autogenerated constructors as part of sast generation

    * Add semantic check to ensure functions always return and don't have unreachable code

commit 3e00e0e719cee2272331c4676beec588b260c3e0
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Mar 28 15:25:02 2021 -0700

    Rename language to Boomslang and other enhancements (#60)

    * Rename language to Boomslang, automatically add NEWLINE to end of each file, and add easier way t

    * Update docs and change boom script to boomc

commit de786e598c98057e59083bb4ea16d75b5cb44250
Author: David Steiner <dsteiner93@users.noreply.github.com>
```

```
Date:   Wed Mar 24 11:11:40 2021 -0700

    Allow printing of arbitrary strings, enhance sast, semant, codegen, Makefile, and README (#58)

    * Allow printing of arbitrary strings, enhance sast and codegen and improve Makefile

    * Update the README

commit 6d363b6b9e9590cf5280fae1494d320ac85aedce
Author: nathanjcuevas <42158119+nathanjcuevas@users.noreply.github.com>
Date:   Tue Mar 23 23:42:11 2021 -0700

    Add support for Hello World program (#57)

    - Add support for a simple hello world program
    - Have the project compiler with OCaml 4.05.0 (the version that docker uses)

    Co-authored-by: Robert Kim <robert23kim@gmail.com>

commit 506b1905b1508276f344b4221910fe322a7abd58
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Wed Mar 17 17:31:41 2021 -0700

    Add pretty printer for AST and other improvements (#55)

    * Adds a pretty printer for the AST that integrates with graphviz to generate graphical representati

    * Adds unit tests for loops

    * Adds the final possibilities for our class declaration so every possible permutation is covered

    * Moves obj operators from binops to method calls in the AST

commit 5d7aa6b31c7e931f5409ccf91eb6a34709afee9b
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Tue Mar 16 11:32:03 2021 -0700

    Add support for INDENT/DEDENT and make class declarations more flexible (#53)

    Co-authored-by: Nathan Cuevas nathanjcuevas@gmail.com

commit e0b98584d6ddc6813a6d0bf6e32cfffd7a7bfe5e
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Mar 14 18:21:44 2021 -0700

    Define AST and integrate with parser (#48)

    * pair programming session ast ml dev

    * added arrays and objects ast

    * func ast and minor bug fixes

    * Finish AST and fix compilation errors
    * Add more generated files to gitignore

    * Update ast for clarity/specificity

    Co-authored-by: Robert Kim <robert23kim@gmail.com>

commit 3cbe58831d2d5f05b2c54d6bfd7494147db26f5f
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Wed Mar 3 19:01:58 2021 -0800

    Add project documents to repo (#42)

commit 7565e558aeeb923581c64a425cb57c89cc10ca15
Author: nikhilmin-k <nikhilmin.k@gmail.com>
Date:   Wed Feb 24 20:49:37 2021 -0800

    Add support for infix operators between objects (#41)

    * old stuff doesn't work

    * finished adding object operator functionality

    Co-authored-by: David Steiner <dsteiner93@users.noreply.github.com>

commit 91320f0b063529475e85fe945b848810eaa3c198
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Wed Feb 24 19:07:05 2021 -0800

    Add support for char and string literals (#38)

commit 5187c28ad4ac696893369ebdb1b3c3a97e103c69
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Wed Feb 24 08:59:30 2021 -0800

    Attempt to fix bugs for fdecl and if/elif (#37)

commit 27cf68e39324e04673458fa7e898d245d1a05108
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Tue Feb 23 14:49:28 2021 -0800

    Update handling of primitives, assignments, and remove CONSTRUCT token (#36)

    * Update handling of primitives and remove CONSTRUCT token

    * Fix bugs with assign/updates and class declarations

    * Allow assignment without type

commit 5f43f766804cc915c2f9b6aea96e07afa2d480ff
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Mon Feb 22 15:28:54 2021 -0800

    Add proper syntax for objects and self keyword (#34)
```

```
commit 547c538e40e8ca04646ecc5be97efeb97e266f55
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Feb 21 19:22:32 2021 -0800

    Add better support for newlines and make test output cleaner (#33)

commit fe011c7a9f189d8bbe30f8d729acbff83fe35298
Author: robert23kim <robert23kim@gmail.com>
Date:   Sun Feb 21 20:26:46 2021 -0500

    implemented comments (#29)

    * implemented comments

    * move comments to scanner

commit ed825075e5ec03b09f3358c77e9329a163e3960b
Author: robert23kim <robert23kim@gmail.com>
Date:   Sun Feb 21 18:16:14 2021 -0500

    double_eq fix (#30)

    * double_eq fix

    * added double eq tests

commit 1b0c89db003baad2c3ea32b06b2031df6f08c943
Author: robert23kim <robert23kim@gmail.com>
Date:   Sun Feb 21 16:38:32 2021 -0500

    put NULL in expr (#31)

    * put NULL in expr

    * added NULL tests, regex on NULL

commit a1fe1ac702a108bae9927b15b6e7b47d33e26355
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Feb 21 12:11:19 2021 -0800

    Add a working REPL, tests, fix bugs, and add features (#28)

    * Add a working REPL, support for array types and literals, unary minus sign, and fix bugs with pars

    * Fix ambiguity in grammar

    * Add unit testing module

    * Enable more compile warnings, add tests, and add support for empty array literals

    * Add test command to Makefile

commit 81219d63a20698a6fbe9559687cbfcac65eeb15f
```

```
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Fri Feb 19 12:58:28 2021 -0800

    Fix ambiguity and various bugs in the grammar (#21)

commit 99717916a4ad8769033048e51ac4207a2d7b71de
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Thu Feb 18 22:49:47 2021 -0800

    Add initial draft for lexer and parser (#1)

    * Add support in lexer for basic tokens

    * Add initial draft for parser

commit a9a4e620e7871e308267ac48087c63e8b3402776
Author: David Steiner <dsteiner93@users.noreply.github.com>
Date:   Sun Jan 17 15:36:57 2021 -0800

    Initial commit
```