# Scolkam
## Final Report


| Steve K. Cheruiyot | Yekaterina Fomina | Connor P. Hailey | Léopold Mebazaa | Megan O'Neill |
|---|---|---|---|---|
| (skc2143) | (yf2222) | (cph2117) | (lm3037) | (mo2638) |
| System Architect | Language Guru | System Architect | Manager | Tester |


May 12, 2016

# Contents

# Chapter 1

# Introduction

The Scolkam programming language is a general purpose programming language. Scolkam is a subset of the Python programming language. It takes the most basic features of Python (arithmetic operations, control flow, etc.). This language is designed to be a technical exercise rather than a solution to a domain-specific problem.
The input language of Scolkam syntactically similar to the Python programming language. The output of the translator is LLVM code. The language is designed to be easily readable for both novices and experienced programmers alike. Syntactically, the language reads much like a sentence, similar to Python.

# Chapter 2

# Language Tutorial

## 2.1 Prerequisites

There are several software requirements for running the Scolkam language. Since the compiler is written in Ocaml, and compiles down to LLVM, both Ocaml and LLVM must be installed on the target machine. Additionally, Ocaml requires several different packages, which can be installed and managed via Opam, the Ocaml package manager.

There are few incompatibilities, howvever. First off, depending on your insallation, the interpreter that will need to be used is going to be rather `lli` or `lli-3.7`. To work, the minimum version of LLVM you should be using is LLVM 3.7. If your version of `lli` is older than 3.7, you should upgrade, as most programs will not work. `testall.sh` uses the `lli-3.7` interpreter; if it does not work, you should modify the file so that `lli` is used instead. Second, we have not been able to use our interpreter on Mac OS, as there apparently is a system-dependent bug in the LLVM interpreter on that platform. We did our testing on Ubuntu Virtual Machines, akin to the one that Prof. Edwards distributed in class.

## 2.2 Using a Compiler

Type `make` inside the *Scolkam* directory to create a compiler. The compiler takes a Scolkam file with *.sco* extension as input and outputs LLVM code.

Input: test-hello_world.sco

```
def None printIt():
    print("Hello World")
end

printIt()
```

Type the following command to compile the above code:
`./scolkam.native -c tests/test-hello_world.sco | lli-3.7`

The output will be as follows:
`Hello World`

## 2.3 Data Manipulation

### 2.3.1 Declaration and Assignment

#### Variables

To declare a variable, the name of the variable needs to be preceded by its type. There are 5 data types implemented in Scolkam: `str`, `int`, `double`, `tuple` and `None`. At the declaration the variable needs to be initialized as well. Once the variable is declared and initialized, it can be assigned values.

```
str b = ""
b = "hi"

int a = 0
a = 1

tuple c = ()
c = (1,2,3)
```

#### Functions

The function is enclosed in a code block that starts with the keyword `def` and ends with the keyword `end`. The name of the function needs to be preceded with the return type and followed by the list of arguments enclosed in brackets separated by commas. Each argument name has to be preceded with its type. The argument list can be empty if no arguments are required.

```
def float somefunc1(int a, float b):
   return float(a + b)
end
def None somefunc2():
   print("somefunc2")
end
```

### 2.3.2 Control Flow

#### if statement

The `if` statement has `if`, `elif`, and `else` constructs. The keyword `end` completes a code block corresponding to each construct as shown below.

```
int x = 0
if x == 0:
   print("x = 0")
end
elif x == 1:
  print("x = 1")
end
else
   print("x <> 0 and x <> 1")
end
```

#### while loop

The syntax for `while` is similar to most programming languages. The keyword `while` is followed by the condition for loop termination. The instructions in the body of the loop are succeeded by the keyword `end`.

```
int x = 10
while x != 0:
# instructions go here
end
```

### for loop

The following example illustrates the syntax of the `for` loop.

```
int x = 0
for(x = 1; x < 10; x = x + 1):
   print(x)
end
```

# Chapter 3

# Language Manual

## 3.1 Lexical Analysis

A Scolkam program is read by a parser. The lexical analyzer generates a stream of tokens that is passed to the parser. This chapter describes the rules the lexical analyzer follows to break a program into tokens.

### 3.1.1 Line structure

A Scolkam program is divided into a number of logical lines.

#### Logical lines

The end of a logical line is represented by the newline character. Statements cannot cross logical line boundaries except where the newline character is allowed by the syntax. For instance, the newline character is allowed between statements in a compound statement, since a compound statement is considered a logical line. This means that one can find logical lines embedded inside logical lines.

A logical line consists of one or more physical lines by following the implicit line joining rules. The `end` keyword indicates the end of logical line as well. Statements within logical lines can be separated by a semi-colon.

#### Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. Scolkam recognizes only two line termination sequences (using ASCII): LF (linefeed - '\n') and CR LF (carriage return followed by linefeed - '\r \n').

#### Comments

Single-line comments start with a hash character (#) that is not part of a string literal, and are terminated by the end of a physical line. A single-line comment indicates the end of the logical line.

```
str tuple weekend = ("Saturday", "Sunday")
# wish there is no homework on these days
```

Multi-line comments are achieved by inserting a multi-line string enclosed in a pair of three single quotes (''') or in a pair of three double quotes (""").

```
'''
    Expected output:
```

```
    multi-line comments in single quotes
'''
prints("multi-line comments in single quotes")

"""
    Expected output:
    multi-line comments in double quotes
"""
prints("multi-line comments in double quotes")
```

#### Blank lines

A logical line that contains only spaces, tabs, formfeeds and a comment, is ignored. No EOL token is generated for this line.

#### Whitespace

The whitespace characters can be used interchangeably to separate tokens. Recognized whitespace characters are:

- Space
- HT (horizontal tab - '\t')
- FF (form feed - '\f')
- CR (carriage return - '\r')

#### Indentation

Unlike Python, the beginning and the end of a function or a branch would not be defined by indentation. The keyword `end` would indicate the end of the code-block. Therefore, inside a code-block, users can use whatever indentation they want. The following example provides a valid output in Scolkam.

```
int bmi = 20
if bmi < 22:
prints("Your BMI is normal")
  prints("No need to get on a diet")
end
'''
Output:
  Your BMI is normal
  No need to get on a diet
'''
```

### 3.1.2 Identifiers

The valid characters for identifiers are the uppercase and lowercase letters A through Z, the underscore and, except for the first character, the digits 0 through 9. Uppercase and lowercase characters are distinct, so a variable named *var* will be different from a variable named *VAR*. The length of identifiers is unlimited.

To find a valid identifier in Scolkam the following regular expression rule can be used:

```
[A-Za-z_][A-Za-z_0-9]*
```

### 3.1.3 Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers.

```
and       bool      break
class     continue    def
else        end     False
for          if     import
is         None       not
null         or     return
self      True      while
```

We must also add the following identifiers:

```
file    float    int
object    str   tuple
```

Even though they are not keywords, they are, for all practical purposes, reserved identifiers, since they identify the built-in types and objects.

We must also indicate the following identifier:

```
__init__
```

Even though this is not a keyword, it is, for all practical purpose, a system-defined reserved identifier, as it is a function name that can be used only as a constructor for objects.

### 3.1.4 Literals

In lexical analysis, literals of a built-in type are a token type with a grammar rule:

⟨*literal*⟩ ::= ⟨*stringliteral*⟩
| ⟨*integer*⟩
| ⟨*floatnumber*⟩
| ⟨*bool_literal*⟩
| ⟨*none_literal*⟩

#### Integer - `int`

Integer literals are described by the following lexical definitions:

⟨*integer*⟩ ::= ⟨*decimalinteger*⟩ | ⟨*none_literal*⟩
⟨*decimalinteger*⟩ ::= ⟨*nonzerodigit*⟩ ⟨*digit**⟩ | `0`+
⟨*nonzerodigit*⟩ ::= `1`...`9`
⟨*digit*⟩ ::= `0`...`9`

#### Float - `float`

Floating point literals are described by the following lexical definitions:

⟨*floatnumber*⟩ ::= ⟨*pointfloat*⟩ | ⟨*exponentfloat*⟩ | ⟨*none_literal*⟩
⟨*pointfloat*⟩ ::= ⟨*[intpart]*⟩ ⟨*fraction*⟩ | ⟨*intpart*⟩ `.`
⟨*exponentfloat*⟩ ::= (⟨*intpart*⟩ | ⟨*pointfloat*⟩) ⟨*exponent*⟩
⟨*intpart*⟩ ::= ⟨*digit*⟩+
⟨*fraction*⟩ ::= `.` ⟨*digit*⟩+
⟨*exponent*⟩ ::= (`e` | `E`) [`+` | `-`] ⟨*digit*⟩+

### Boolean - `bool`

There are two literal values of the *bool* type, defined according to the following rule:

⟨*bool_ literal*⟩ ::= `True` | `False`

### String - `str`

String values of the `str` type represent sequences of characters. The `str` type stores only ASCII characters. Strings can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as triple-quoted strings in Python). The backslash character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

The grammar rules for the *str* type appear below.

⟨*stringliteral*⟩ ::= ⟨*shortstring*⟩ | ⟨*longstring*⟩ | ⟨*none_ literal*⟩
⟨*shortstring*⟩ ::= ⟨*"'"*⟩ ⟨*shortstringitem*⟩* ⟨*"'"*⟩ | ⟨*'"'*⟩ ⟨*shortstringitem*⟩* ⟨*'"'*⟩
⟨*longstring*⟩ ::= ⟨*"'''"*⟩ ⟨*longstringitem*⟩* ⟨*"'''"*⟩ | ⟨*'"""'*⟩ ⟨*longstringitem*⟩* ⟨*'"""'*⟩
⟨*shortstringitem*⟩ ::= ⟨*shortstringchar*⟩ | ⟨*stringescapeseq*⟩
⟨*longstringitem*⟩ ::= ⟨*longstringchar*⟩ | ⟨*stringescapeseq*⟩
⟨*shortstringchar*⟩ ::= ⟨*any ASCII character except "\" or newline or the quote*⟩
⟨*longstringchar*⟩ ::= ⟨*any ASCII character except <"\">*⟩
⟨*stringescapeseq*⟩ ::= ⟨*"\"*⟩ ⟨*any ASCII character*⟩

The recognized escape sequences are:

| Escape Sequence | Meaning | Notes |
|---|---|---|
| `\newline` | Backslash and newline ignored | |
| `\\` | Backslash (\) | |
| `\'` | Single quote (') | |
| `\"` | Double quote (") | |
| `\f` | ASCII Formfeed (FF) | |
| `\n` | ASCII Linefeed (LF) | |
| `\r` | ASCII Carriage Return (CR) | |
| `\t` | ASCII Horizontal Tab (TAB) | |
| `\v` | ASCII Vertical Tab (VT) | |

Notes:

1. As in Standard C, up to three octal digits are accepted.
2. Unlike in Standard C, exactly two hex digits are required.

---

```
# escape single-quote with backslash
prints("I am 6\'2 tall.")

# escape double-quote with backslash
prints("Leonardo DiCaprio wins best actor Oscar for \"The Revenant\"")
'''
Output:
I am 6'2 tall.
Leonardo DiCaprio wins best actor Oscar for "The Revenant"
'''
```

---

#### Tuple

The elements of a tuple are objects. The objects have to be of the same type. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (singleton) can be constructed by adding a comma after an expression. One expression by itself doesn't create a tuple. Parentheses are required.

```
int tuple tuple1 = (1,) # singleton tuple
int tuple tuple2 = (1,2) # tuple with multiple elements
```

#### None - `None`

*None* type is a special type that has only one literal value *None*. It can be used as a value for any type of object (*int*, *float*, *str*, *tuple*).

⟨*none_literal*⟩ ::= `None`

### 3.1.5 Operators

The following tokens serve as operators in the grammar:

\+ - * / % == != < > <= >= is in and not or

### 3.1.6 Delimiters

The following tokens serve as delimiters in the grammar:

, ; . : = ( ) += -= *= /= %=

### 3.1.7 Scoping Rules

Scolkam is statically scoped. The scope of a variable is set so that it can only be referenced from within the block of code in which it is defined. In Scolkam, the scope of any name can be easily determined by looking at the program.
Functions, variables, and classes have names. There cannot be multiple global variables defined with the same name. An exception will be thrown as shown below.

```
str book = "Compilers"
prints(book)
#returns "Compilers"

str book = "100"
print(book)
# throws an exception: exception Failure("Duplicate global book")
```

A name can be bound to values by:

- variable assignment: ***x*** = *1*
- function argument: *def None foo(int* ***x****)*
- function definition: *def None* ***x****()*
- class definition: *class* ***x***
- 'for' loop: *for* ***x*** *in (1, 2, 3, 4)*

The scope of a name is bound by program, or function, or class.

```
int age = 10
# scope of 'age' is the entire program

def None global():
   print(age)
   # 'age' is not found in the enclosing function
   # it is found in the next outer enclosing space - program, so it prints 10
end

def None local1():
   int age = 3
   # function's own local variable is defined
   print(age)
   # prints 3
end

def None local2(int age):
  print(age)
  # prints the argument value passed, 'age' is local to this function
end

global()
local1()
local2(15)

''' Output:
10
3
15
'''
```

Keywords `def` and `class` create new enclosing space.

## 3.2 Data Types

As in Python, all data is represented by an object. Every object has an identity, a type and a value. These objects are immutable, so all three characteristics of the object do not change once the object has been created. Here, type and class define the same thing.

So, all data is an object. There is, however, a difference between built-in data types, which are built into the interpreter, and the other object types that are defined in the standard library or by the user. We will describe the built-in data types in this section. [1]

### 3.2.1 Type inference

As much as we would like to stick as close as we can to Python's syntax, Python is, in contrast to our language, dynamically-typed. With LLVM, we cannot easily perform JIT compilation, so our language will be statically-typed.

[1] We will not respect Python's standard type hierarchy, as it is described here: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types. In particular, in Python, the types `int`, `float`, etc. are subclasses of a class called `numbers.Number`. This will not be the case here. We will assume that all the built-in data types are types on their own.

```
int varname1 = 23
float varname2 = 4.2
str varname3 = "hello"
int tuple varname4 = (1, 2, 3)
```

### 3.2.2 None

Every program executed has a built-in object that can be accessed through the name `None`. It has a single value, and signifies the absence of value. (Its truth value is false.)

### 3.2.3 int

These objects represent numbers between -2,147,483,648 and 2,147,483,647. These are the equivalent of all binary numbers stored with 32 bits.

```
int var = 1
```

### 3.2.4 bool

These objects represent the truth values `False` and `True`. They are stored in memory as 0 and 1. They can be treated as `int` in computations. For example, `1 + True` returns `2`.

```
bool var = False
# Not "false"!
```

### 3.2.5 float

These objects represent numbers between 1e-37 and 1e37. They should be used to store floating point numbers. These are stored with 64 bits in memory.

```
float var = 0.5
```

### 3.2.6 tuple

The items of a tuple are any Scolkam objects. The items in the tuple have to be of the same class.
We will allow high-level item assignment while preserving the immutability of the objects. This means that operations that are written like `tuple[index] = value` are authorized. We do that by replacing in the parser phase this notation by an assignment that respects the immutability of the objects.

In practice, an instruction under the format:

```
example_tuple[i] = newval
```

will be transformed by the compiler into an instruction like this:

```
example_tuple = example_tuple[0:i] + newval + example_tuple[i+1:]
```

In other words, we simulate object mutability.

### 3.2.7 str

These objects represent a sequence of characters, where each character is encoded in ASCII format.

### 3.2.8 Casting

We will provide various casting operations. See section on Standard Library.

## 3.3 Expressions

This chapter explains the meaning of the elements of expressions in Scolkam. An expression is a code that produces a result or a value. Evaluation rules for expressions described in this chapter are adapted from Python implementation.

### 3.3.1 Literal Expressions

A literal expression evaluates to the value it represents. Here are some examples of literal expressions.

```
42
#evaluates to 42
"some text"
#evaluates to 'some text'
```

The above example shows the expression on the left and the value to which it evaluates on the right.

### 3.3.2 Operator Expressions

#### Binary

A binary expression is where a binary operator applied to two operand expressions. A binary operator takes two arguments. Here are some examples of binary expressions.

```
6 * 7
#evaluates to 42
40 + 2
#evaluates to 42
"Hello, " + "World"
#evaluates to 'Hello, World'
```

The last example demonstrates that some expressions don't evaluate to numbers. This particular expression evaluates to the concatenation of two strings.

In general, binary expressions have the following form:

⟨*binary_expr*⟩ ::= ⟨*expr*⟩ ⟨*bin_op*⟩ ⟨*expr*⟩

When a binary expression is evaluated, first, the operands (left, and then right) are evaluated to a value and replaced with their corresponding values, then *bin_op* (binary operator) is applied to the two resulting values thus getting the value for the entire binary expression. The expression is replaced with the obtained value.

#### Compound

When at least one of the operands is an expression itself, the expression is categorized as a compound expression. Here are some examples of compound expressions.

```
6 * 6 + 6                   #evaluates to 42
32 + 2 + 2                  #evaluates to 36
False and not True          #evaluates to False
```

Parentheses can be used to override the defined Scolkam's order of operations.
Note: A singleton tuple has to have a comma at the end, this allows us to distinguish between a singleton tuple and an expression inside parentheses.

```
int tuple t = (2+1,20)
#tuple
int a = (1+1)
#expression

int b = t[0]
print(a)
print(b)
'''
Output:
2
3
'''
```

Compound expressions enclosed in parentheses have the following form:

⟨*compound_expr*⟩ ::= ( ⟨*expr*⟩ )

To evaluate a compound expression, the main operator has to be identified first by using the order of operations rules. The main operator is the one that gets applied at the end. For example, in 6 * 6 + 6 the main operator is +, thus making this expression an addition expression. This addition expression gets evaluated using the same rules as a binary expression.

### Unary

A unary expression is where a unary operator applied to a single value. Here are some examples of unary expressions:

```
-(6 * 7)        #evaluates to -42
not True        #evaluates to False
```

A unary expression has the following form:

⟨*unary_expr*⟩ ::= ⟨*un_op*⟩ ⟨*expr*⟩

Unary operators are - and `not` . The - operator negates a numerical value, the `not` operator negates a boolean value.

To evaluate a unary expression, the operand expression needs to be evaluated prior to applying the unary operator on it.

### Variable Access

A variable access expression allows you to use the value of a variable you have assigned. Here are some examples of variable access expressions:

```
int result = 42
result        #evaluates to 42
result - 42   #evaluates to 0
```

A variable access expression has the following form:

⟨*var_expr*⟩ ::= ⟨*var_expr*⟩

To evaluate a variable access expression, the bindings are searched for a binding from *var_ expr* to a value. If a binding is found, then the variable access expressions is replaced with the value found. If no such binding is found, then an error is raised.

### Tuple Constructor

A constructor creates a new tuple. Here is an example of a tuple constructor:

```
int tuple = (40, 42)          #a tuple with elements 40 and 42
```

A tuple constructor has the following form:

⟨*tuple_ construct*⟩ ::= (⟨*expr*⟩, ⟨*expr*⟩, ..., ⟨*expr*⟩)

Tuples are evaluated from left to right. Each expression *expr* in the comma-separated list is evaluated to a value which replaces that expression. The values for all expressions in the tuple replace the tuple constructor expression preserving the order. Duplicates are not removed. Once the tuple is constructed, it can only be modified with high-level assignment described in Section 3.6.

### Tuple Element Access

A tuple is a sequence. A sequence stores elements in a fixed order and gives each element a unique index. An element in a tuple can be accessed by using its unique index. Here is an example of a tuple element access expression:

```
int tuple tp = (5, 42, 7)
int a = tp[1]
print(a)
# returns 42 (2nd element)
```

A tuple element access expression has the following form:

⟨*tuple_ access*⟩ ::= ⟨*expr*⟩ [⟨*index_ expr*⟩]

To evaluate a tuple element access expression to a value, the expressions *expr* are evaluated to a value followed by the results of the evaluation of *index_ expr*. If *expr* evaluates to something other than a sequence, then an error is raised. If *index_ expr* does not evaluate to an integer on the interval */-n, n - 1/*(where *n* is the length of the tuple), then an error is raised. If *index_ expr* is negative, *index_ expr* value is replaced with the result of *len(expr) - /index_ expr/*. Accessing index - 1 is equivalent to accessing the last element in the sequence. The tuple access expression is replaced with the value in the sequence at that index. The first value in the sequence is located at index 0.

### Function Call

A function call expression executes a function as a part of a larger expression (for example, a variable assignment). Here is an example of a function call as part of a variable assignment:

```
def int seed():
        return 1
end

int a = 0
a = seed()
print(a)
#returns 1
```

A function call expression has the following form:

⟨*func_ expr*⟩ ::= ⟨*func_ name*⟩[⟨*param_ expr*⟩, ..., ⟨*param_ expr*⟩]

## 3.4 Operators

The operators only work between two operands of the same type. For instance, `3 + 3` is possible, but `3 + 3.0` is not. In order to make this operation possible, a number of casting operations `int()`, `float()`, `str()`, and `bool()` are available (see Standard Library section.) For logical operators, only `bool` objects are accepted.

### 3.4.1 Arithmetic operators

Arithmetic operators return the result of an arithmetic operation performed on a couple of operands. The type of their result is the same as the type of both operands.

#### Addition - +

The + performs an addition with both operands.

```
3.0 + 2.1
# 5.1
```

#### Subtraction - −

The − performs a subtraction with both operands.

```
3.0 − 2.1
# 0.9
```

#### Multiplication - *

The ⋆ performs a multiplication of both operands.

```
3.0 * 2.1
# 6.3
```

#### Division - *

The / performs a division between both operands. Note that the division can be performed only between two operands of the same type, and that the return type is the same as the types of the operands.

```
3.0 / 2.1
# 1.4285714286
3 / 2
# 1
```

#### Modulus - %

The % performs a modulus between both operands. Note that the division can be performed only between two operands of the same type, and that the return type is the same as the types of the operands.

```
3.0 % 2.1  # 0.8999999999999999
3 % 2 # 1
```

### 3.4.2 Comparison

These operators compare values that are bound to expressions or identifiers. These operators fetch the values of these expressions or identifiers, and then put them on a scale to compare them. There are two types of expression: equality operations, that check that two values are the same, and pure comparison operations, that check that a value is lesser or greater than an other.

The equality and inequality operators work with all types. For custom-made objects, the equality is determined by checking the equality of all the attributes of the both objects. (Once again, the two objects that are compared must be of the same class.) The rest of the comparison operators do not work with all types, but only with `int`, `float` and `bool`. List types such as `str` and `tuple` are not compared together on a scale, but you can perform such an operation handy with a `for` loop. (See next section for more.)

#### Equality - ==

The equality operators checks the value equality of two expressions. If the values of two operands are equal, then the expression returns `True`.

```
3 == 2  # False
```

#### Inequality - !=

The equality operators checks the value inequality of two expressions. If the values of two operands are equal, then the expression returns `False`.

```
"Gold" != "GOLD" # True
```

#### Strictly less than - <

The < operator checks if the value of left operand is less than the value of right operand. If it is, then the expression returns `True`.

```
4.1 < 4.0 # False
```

#### Strictly greater than - >

The > operator checks the value of left operand is greater than the value of right operand. If it is, then the expression returns `True`.

```
"Gold" > "GOLD" # Illegal operation
True > False # True (since True is 1 and False is 0.)
```

#### Less than - <=

The <= operator checks the value of left operand is less than the value of right operand. If it is, or if the two operands have the same value, then the expression returns `True`.

```
True <= True # True
int(4.0) <= int(4.6) # True
```

#### Greater than - >=

The >= operator checks the value of left operand is greater than the value of right operand. If it is, or if the two operands have the same value, then the expression returns `True`.

```
3 >= 3 # True
int(4.0) >= int(4.6) # False
```

### 3.4.3 Assignment - =

The assignment operator assigns values from right side operands to left side operand.

```
c = a + b  # assigns the value of a + b into c
```

### 3.4.4 Identity

Identity operators check the equality of two objects by comparing the memory locations. They are therefore different from equality operators.

#### is

The `is` operator evaluates to `True` if the variables on either side of the operator point to the same object and `False` otherwise.

```
int var1 = 3
int var2 = var1
var1 is var2 # True
"Hello" is "Hello" # Probably False, will depend on the compiler
    implementation
```

#### is not

The `is not` operator evaluates to `False` if the variables on either side of the operator point to the same object and `True` otherwise.

### 3.4.5 Logical

Logical operators only work with `bool` objects.

#### Conjunction - and

Equivalent to a conjunction binary operator. If both the operands are `True` then condition becomes `True`.

```
True and False # False
```

#### Disjunction - or

Equivalent to a disjunction binary operator. If any of the two operands are not `False` then condition becomes `True`.

```
bool(2) or False # True
```

### 3.4.6 Unary

Unary operators are operators that do not apply to two, but one operand. We have two operators: −(as in minus) and `not`. The first one works on a `int` or a `float`. The second one works on a `bool`.

#### Minus - −

The minus operators returns the numerical opposite of the operand.

```
− (3 + 4) # −7
```

#### Negation - `not`

Equivalent to a negation binary operator. Used to reverse the logical state of its operand.

```
not False # True
```

## 3.5 Statements

Scolkam distinguishes two types of statements: simple statements and compound statements. Compound statements are different from the simple statements that they can comprise simple statements inside themselves. We will look at both versions separately.

### 3.5.1 Simple Statements

A simple statement is comprised withing a single logical line. Python-inspired Scolkam's grammar for simple statement is:

⟨*simple_stmt*⟩ ::= ⟨*expression_stmt*⟩
| ⟨*assignment_stmt*⟩
| ⟨*augmented_assignment_stmt*⟩
| ⟨*return_stmt*⟩
| ⟨*break_stmt*⟩
| ⟨*continue_stmt*⟩
| ⟨*import_stmt*⟩

Each line represents one type of simple statement that we will look at separately.

#### Expression Statements

Expression statements in Python are used mostly in interactive mode, which we will not support here. When not used in interactive mode, expression statements are usually used to call a procedure (which is a function that returns nothing or `None`.).

```
function_call()
```

#### Assignment Statements

Expressions statements are slightly different from assignment statements, that are "used to bind (re)bind names to values and to modify attributes or items of mutable objects". In contrast to Python, only one name can be bound at a time.

```
var1, var2 = returnstwovalues() # returns an error in Scolkam
int tuple var1 = (3, 4)
# possible in Scolkam, var1 will be a tuple
```

The data type is to be mentioned during assignment statements.

```
int var1 = 2          # var1 is an int
float var1 = 2.0        # var1 is a float
int var2 = 3.0  # Parser error
```

#### Return Statements

Return statements may occur only in a function definition. The expression is evaluated before it is returned. If no expression is present, then "None" is returned.

⟨*return_stmt*⟩ ::= `return` [⟨*expression*⟩]

```
# In a function that has a bool return type
  return False
```

#### Break Statements

Break statements may occur only in a loop. It terminates the execution of the instructions of the nearest enclosing loop.

⟨*break_stmt*⟩ ::= `break`

#### Continue Statements

Continue statements may occur only in a loop. It continues the execution of the instructions of the nearest enclosing loop with the next cycle.

⟨*continue_stmt*⟩ ::= `continue` [⟨*expression*⟩]

### 3.5.2 Compound Statements

#### The `if` statement

⟨*if_stmt*⟩ ::= `if` ⟨*expression*⟩ `:` ⟨*statements*⟩ (`end elif` ⟨*expression*⟩ `:` ⟨*statements*⟩ )* [`end else :` ⟨*statements*⟩] `end`

The `if` statement is used to execute different instructions based on conditions.
The structure here is similar to any generic programming language, with *if*, *else if*, and *else* constructs (although the syntax of *else if* is here `elif`).

```
str string_number = ""
int var = 4
if var == 4:
  string_number = "four"
elif var == 6:
  string_number = "six"
else:
  string_number = "unknown (but definitely not four or six)"
end
prints(string_number)
```

### Loops: while, for

It is worth nothing that `break` and `continue` statements can apply in these statements. (See earlier to see the effects of these statements.)

### The while statement

⟨*while_stmt*⟩ ::= `while` ⟨*expression*⟩ `:` ⟨*statements*⟩ `end`

The `while` statements is used for repeated execution as an expression is true.
The expression is evaluated once. If it is true, then the statements inside the loop are executed until the expression is not true anymore.

```
int a = 40

while (a < 43):
        print(a)
        a = a + 1
end
```

### The for statement

⟨*for_stmt*⟩ ::= `for` ⟨*target_list*⟩ `in` ⟨*expression_list*⟩ `:` ⟨*statements*⟩ `end`

The `for` statement is used to iterate over the elements of a sequence (a string, a tuple). For every element in this sequence, the statements are evaluated once (except for a `break` instruction).
The scope of the variables in `target_list` is the loop. Let us look at this example:

```
def int testIt():
  int i = 0
  for (i = 0; i < 5; i = i + 1):
    print(i)
  end
end

print(42)
testIt()
```

In this example, the loop runs 4 times, and not another number of times. The scope of $i$ is just the loop. The loop makes assignments to the variable $i$ in the target list (in the above example the target list is *(1, 2, 3, 4)*). These assignments overwrite all previous assignments to the variable $i$.

### Function definitions

⟨*funcdef*⟩ ::= `def` [⟨*type*⟩] ⟨*funcname*⟩ `(` [⟨*parameter_list*⟩] `):` ⟨*statements*⟩ `end`
⟨*parameter_list*⟩ ::= (⟨*parameter*⟩)*
| ⟨*parameter*⟩ [`,`] `)`
⟨*parameter*⟩ ::= ⟨*type*⟩ ⟨*name*⟩

Here, <*type*> means the type that the object is. As, we explained earlier, every parameter must have an explicit type. Therefore, the type of the arguments must be explicit in the function definition. Functions can be overloaded, in the sense that there can be multiple functions with the same name, but different argument and return types. Functions do not accept optional arguments.

A function definition, which is an executable statement, defines a function object. Using the name defined in that function definition calls the execution of that function.

`return` statements are generally expect in a function, and terminate them. Here is an example of a function:

```
def float somefunc(float a, float b):
  return float(a + b)
end
```

# Chapter 4

# Project Plan

## 4.1 Planning Process

Our team used an iterative approach to planning. For the first couple of weeks our group met once a week to brainstorm ideas for a programming language. Once we agreed on the idea, we started meeting twice a week. During our meetings we would discuss goals and assign action items for the following week. The selection of goals and action items was driven by the project milestone schedule and the stages of the translator. Various features were planned and either adapted, added, or dropped depending upon time constraints, and ability to implement.

## 4.2 Specification Process

At first, we needed to ensure that all members of the team are familiar with Python and have some understanding of LLVM IR . This phase was necessary for collecting input from all team members on our language design. Once the list of language features was defined, we were ready to document the details of these features, as well as nuances of their implementation. The first draft of the language reference manual was written in parallel with the scanner and parser development. Close collaboration during this initial stage allowed us to revise the language specification as we realized the level of complexity for implementing certain language features.

## 4.3 Development Process

The compiler structure dictated the stages of the development process. The first components to develop were the scanner and parser, followed by the semantic analyzer. The code generation was the final stage of the development process. All components were tested during the development process.

## 4.4 Testing Process

Several tests were originally built based off of the Micro-C compiler testing suite. Following those, numerous other tests were developed based off of the features detailed in the Language Reference Manual. Though a test-first design was implemented, test driven development was not used. Please see Section 6, "Test Plan", for further details.

## 4.5 Programming Style

Although the language is based upon Python, differences do exist. When defining both classes and functions, an "end" keyword is used to designate the end of a class or function. In definitions, no curly brackets are

used, and arguments must be explicitly listed with their return types. Additionally, variables must first be declared with their associated return type, and then assigned a value (please see the Language Tutorial for more information).

## 4.6 Project Timeline

| | |
|---|---|
| February 10th | Language Proposal submitted |
| February 19th | Revised Language Proposal submitted |
| March 7th | Language Reference Manual submitted |
| March 22nd | Scanner development completed |
| April 1st | Parser and AST development completed |
| April 19th | 'Hello World' demo |
| April 28th | Semantic Analyzer completed |
| May 11th | Code Generator completed |
| May 11th | Project Presentation and Final Report submission |

## 4.7 Roles and Responsibilities

Initially assigned roles and responsibilities are summarized below.

| | |
|---|---|
| Connor P. Hailey | System Architect |
| Léopold Mebazaa | Manager |
| Megan O'Neill | Tester |
| Steve K. Cheruiyot | System Architect |
| Yekaterina Fomina | Language Guru |

Even though the main roles stayed the same throughout the project, responsibilities of team members evolved during multiple stages of the project. As needed team members took additional responsibilities to meet the milestone schedule and progress the project further.

| | |
|---|---|
| Connor P. Hailey | Scanner, Parser, Code generator |
| Léopold Mebazaa | Scanner, Parser, Code generator, LRM |
| Megan O'Neill | Test case creation, Testing Script, Final Report |
| Steve K. Cheruiyot | Scanner, Parser, AST, Code generator |
| Yekaterina Fomina | LRM, Scanner, Final Report, Testing Script |

## 4.8 Development Environment

Our programming enviornment was pretty unexceptional. The only notable feature is that we used Vagrant Virtual Machines with Ubuntu distributions in order to have a similar envionrment.

## 4.9 Project Log

```
commit 4aa7fd18257008c815c66253623e4fdeca317d8c
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu May 12 00:02:50 2016 -0400

    You can pass tuples as arguments

commit dfab13b11b33d98e63ec6f519cd31f27ed35bbe8
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 22:22:22 2016 -0400

    lessons learnt

commit a8cf5d26a4ebcea187dd7804274e771813367ae8
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed May 11 21:52:24 2016 -0400

    lesson learned

commit 09effb5add45bec61971ac55dafb9041c2e9fa29
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed May 11 21:52:15 2016 -0400

    lesson learned

commit a6df1bd46d6ab80157b890388e5ede766786a265
Merge: f102b89 9d60c47
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 21:30:24 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit f102b8911780afec584699c804986e4fac7c8938
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 21:29:13 2016 -0400

    suppressing warnings

commit 9d60c47f46abb9c6b142b719dd51aea9374508ad
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 21:09:09 2016 -0400

    updates to Final Report

commit 63211ca7a413519f64a5ee13e3cd94966e95e819
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 20:37:35 2016 -0400

    implementing assignment to tuple elements e.g b[1] = 23

commit 54f128f9cf7830ea0d834d170d015b975d30e6f8
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 17:24:24 2016 -0400

    moveTower example
```

```
commit b2a41375ed1f4ed31e9d396ad20cae67f5c9f870
Merge: c3acfc0 10a8acc
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 17:04:09 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit c3acfc058094934103b78ec76fe0601ed1d33a0e
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 17:03:36 2016 -0400

    arithmetic fix

commit 10a8accfa562c886f7f22be2e2f88e8ffb59a8b0
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 16:59:47 2016 -0400

    standard library beginning

commit 5b0ca9a8e7120313ad37614c67b5e41a3dac3b53
Merge: bd3d2a7 f722d3d
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 16:46:29 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit bd3d2a7e0f1d19f33bef2cbdfaebb54878213a3e
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 16:46:04 2016 -0400

    string literals concatenation

commit f722d3db3285dc252f868df0fc8d0f923adf32b3
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 16:45:50 2016 -0400

    moveTower, print tuple

commit 2899db1a2925687bece847d8f42bd95f3b85bf11
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 15:14:23 2016 -0400

    testing toint function

commit ca3afc6fa17b26974405d7865c3581b9ddc72ee6
Merge: 39200c7 6b68ad3
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed May 11 16:36:50 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit 39200c7d8ab19e867e64e002693d2558a49037fb
Author: Connor Hailey <connorhailey@gmail.com>
```

```
Date:   Wed May 11 16:36:40 2016 -0400

    str concat

commit 1dabd8b8c8b52ebe7bfcbb8b14f655a96a5c08fa
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed May 11 16:35:46 2016 -0400

    str concat

commit 6b68ad33f5b525ab0e6581796bea791e3270d086
Merge: 000e7f3 7d6a118
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 16:10:44 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit 000e7f3c0b7c46935c39ae2bf54bb58e15c63b33
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 16:10:39 2016 -0400

    Break and continue solved

commit 7d6a1187ffc6980e81173e51ea43c01cbb09690a
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 15:03:06 2016 -0400

    more testing...

commit 19ee603f68cd264b342420fe62cbee5715901b9b
Merge: 5027cd8 f4f95d6
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 14:45:28 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit 5027cd8d73b84ffa42d9dec0b8093d7ffa3252d3
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 14:45:20 2016 -0400

    toint converts to integer

commit f4f95d63c30c66f13a654eb5a412858b0f28b849
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 14:42:56 2016 -0400

    more testing stuff

commit e3521b7f2395882e34d8817f9679935aa8d44cec
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 14:27:24 2016 -0400

    Tuples work
```

```
commit 06d12977de41c1c07061c934834d0f673b90fba4
Merge: 558e9d2 206e4f3
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 13:35:39 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam
    "Merge"

commit 558e9d21a135a10055ffc08f2343b4158b71004e
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 13:35:26 2016 -0400

    TUple works

commit 206e4f3ebb064f97793061f03d9bcb441e1905b8
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 12:39:33 2016 -0400

    testing fibonacci

commit 3a65f44d33c5ab00162b4eab6a7f780292d67831
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 12:26:33 2016 -0400

    more testing of tuple access

commit a29b822c064d65843158feaeab3500281f74c761
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 12:13:25 2016 -0400

    testing tuple declaration and access

commit 06435c6a323acd002936fe67a8d77c5af055465e
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 12:02:18 2016 -0400

    adding floating pt operations

commit b0d8e0e7b3a45c81a03cfe5d9ab75dcd685785a7
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 11:53:48 2016 -0400

    testing print int, for loop

commit 1e5acec236ca5903059c2c282688747ec768ede9
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Wed May 11 11:40:30 2016 -0400

    testing scope, float addition and print

commit f859ce7dcbf4de66dc44db53b449464cca03411d
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 10:45:29 2016 -0400
```

```
    Fixed element bug

commit ed2752c535103159e3b63ee1871df2604a28c42c
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed May 11 10:41:28 2016 -0400

    stuff

commit 5d12116f73db3e791d2f8e8fab6167abe81538a9
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed May 11 03:06:01 2016 -0400

    implementing tuples e.g str tuple t = ("I", "am", "a", "string", "tuple")

commit 51b515012c56f4e4a57ae8092135ca1cebf57703
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Tue May 10 17:15:09 2016 -0400

    Added Test Plan, Lang Tutorial and Project Plan sections

commit 4735075226e6dfdc38e98e8cba209ee5022dd0e5
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Mon May 9 14:27:46 2016 -0400

    updates to LRM

commit 980a5e5697428fde9ac40e79056d56b7f1d77849
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Mon May 9 14:21:24 2016 -0400

    adding Final Report doc

commit 52e2d35e832d70ff23a7653e75e9a628b6ab352d
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun May 8 23:01:37 2016 -0400

    House cleaning

commit ccb644356ac63bda748d6fae3b458bd787501107
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Sat May 7 19:37:32 2016 -0400

    5/7 test results

commit 392146e7f1ebc9caccb612ac7eb31f0df1e121de
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Sat May 7 19:34:40 2016 -0400

    test scripts cleanup

commit a2a94ec6585b204a4b9e707c2c4ec49844af2994
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Sat May 7 18:30:42 2016 -0400
```

```
    restoring creation of .ll files

commit 56b14e28950ab2561e802832f6c5aea43ea595db
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 17:59:01 2016 -0400

    Better AST printing

commit f54a6195f22ea5b69b89e6355d736bd85b26fd0d
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 17:34:31 2016 -0400

    Fixed bug stemming from previous bug fix (arg problem)

commit 93132bb3a2a65c7ab169230165cbe6754c5504ea
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 17:26:06 2016 -0400

    Fixed a bug that made something between parentheses considered a tuple

commit 88945e8b52b655ea80b55056cf9a284e2badc036
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 17:12:32 2016 -0400

    Bug fixes

commit 5a07a0fd878d810ea45cf8c84cec04ed45835cd6
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 16:27:55 2016 -0400

    Esthetic fixes

commit 7a5b990433952f90d5eb46e1ebf04433fe4c5692
Merge: 72f2ae4 808bebd
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 15:58:34 2016 -0400

    Added Declaration among statements2

commit 72f2ae4689af4d5fd85f2bb4f4610656934440c8
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 15:57:41 2016 -0400

    Added Declaration among statements

commit 808bebd087b8dba41c398cd504b8c547dd50b977
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sat May 7 12:06:22 2016 -0400

    can access tuple elements. changed to int instead of expr list

commit ad9d20aeb81f3c0f8706104d92f49aa7f3240412
Merge: 3d2e034 c3ce20a
Author: Steve Cheruiyot <skc2143@columbia.edu>
```

```
Date:   Sat May 7 06:49:41 2016 -0400

    implementing variable assignment in declarations

commit 3d2e03431621bbe702d5b4a3982469f1322a178d
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Sat May 7 06:43:51 2016 -0400

    implementing variable assignment in declarations

commit c3ce20aabefc38d2cdd3d5501bd7b4be59b34c12
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 00:37:50 2016 -0400

    Merge issues fixed

commit b1636e3b73757039ee666cb0d45b57d685d02b8e
Merge: 94f9fe2 28959a7
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat May 7 00:22:15 2016 -0400

    Merge complete

commit 94f9fe28517bce7dea63fdf608af2d57fc702a8e
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Fri May 6 23:39:16 2016 -0400

    Elifs handled correctly - 2

commit 71a1d8990d1a7843b5f9ff2f90076139db6c2300
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Fri May 6 23:38:34 2016 -0400

    Elifs handled correctly

commit 28959a7ee28b87f81c16ff1851469ea2bad6068f
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Fri May 6 19:24:39 2016 -0400

    access, only for 0th element at this time. will make it dynamic tmrrw

commit 2c2ab109efd20d670c1830fd265733b52756ad9a
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Fri May 6 18:47:17 2016 -0400

    created tuples. now to access them

commit dfc2272cad6fa05805629f57709327f575c921f5
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Fri May 6 18:47:01 2016 -0400

    created tuples successfully. now for accessing them

commit 84c17d5ed92153d40eaf28efc78d163b4b955db3
```

```
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Fri May 6 18:05:32 2016 -0400

    bug fixes

commit 7f4033baf9fda693d9d6bc45620b6e7791ca64aa
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Fri May 6 16:49:21 2016 -0400

    additions to semantic

commit 00e132eed710310acaebe1643c6d5ade87287128
Merge: 12878a9 faf1ee4
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Thu Apr 28 16:22:17 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit 12878a9ba1425b66a82ec4187929e09fe78fb8b0
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Thu Apr 28 16:20:18 2016 -0400

    improving semantic analyzer

commit faf1ee408dc8386eca3663fa1e3d945e8c9b855b
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Wed Apr 27 12:23:26 2016 -0400

    move avl tree file into bst file

commit 84994eef0e8c68156e7eedd9e1a622c9981d9e67
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Wed Apr 27 10:50:51 2016 -0400

    update bst to conform to LRM

commit d2c24dd9eba6225a19488081c631049e09d915a7
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Mon Apr 25 05:35:58 2016 -0400

    adding elif and object type

commit ad73a55e81982abbccc88eb75610aad27483d0f0
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 22 16:15:32 2016 -0400

    updated readme

commit 36cdfbeaf4f745cd5a6790fa1c8dbb89d6453bb7
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 22 16:11:06 2016 -0400

    testing doc update
```

```
commit 378ed87e8a3a7383d7fff0a5536d02e0165eab22
Merge: 5f9cefa cfbed83
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Tue Apr 19 11:01:36 2016 -0400

    something2

commit 5f9cefae617d3799804eb60324a66c1abef3aa1e
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Tue Apr 19 10:56:59 2016 -0400

    something

commit cfbed83d01f09743dd30f70f9adc8176eb553deb
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Sun Apr 17 15:14:03 2016 -0400

    test results along with additional tests

commit c8ee58f060fe8a98d56d222b8fb0413cf5107a93
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Mon Apr 11 19:13:55 2016 -0400

    fixing reversed stmtns and improving semantic analyzer

commit a0378c26b360b4653be1af6577c63e77f17af099
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Apr 10 12:15:05 2016 -0400

    update some test syntax

commit 27b993a0702013dd393376e7e6f1c7f0773f02ec
Merge: 9be7af7 2449185
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Apr 10 11:32:58 2016 -0400

    Merge branch 'tests'

    Conflicts:
        Makefile

commit 9be7af7bef564ec19e78e883854bc33f8c3f8f46
Merge: e48c26a 4827e71
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat Apr 9 15:21:45 2016 -0400

    Merge branch 'master' of https://github.com/lemeb/scolkam

commit e48c26a218611524ce06b0ec949b679877f83ae5
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sat Apr 9 15:21:28 2016 -0400

    HelloWorld_README
```

```
commit 191f0f4d8145fb5b7d44ea704db540c929bd7e13
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 19:49:41 2016 -0400

    Added REPL instructions

commit 4827e71e02554f12db456be0b09750f6bbf2f7af
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Thu Apr 7 19:42:14 2016 -0400

    fixing multiline comments and removing import stmts

commit f4ca705dd8d2c0821983170178849094b5a54a0a
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 19:31:59 2016 -0400

    Added todolist

commit 128d7bc6fb506ae790563cf0f23f789db2c1592c
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 19:28:16 2016 -0400

    Added instructions for testing pipelines

commit c0366bc4c28d4e981ac933be1b457d4ba88f4345
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 19:04:01 2016 -0400

    Add clean_ast

commit 3f5a49b81bdb31c57f5152a701b3e00c2010c863
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 18:01:14 2016 -0400

    Fixed the merge

commit 704f8774beb0980b3c0b8746de86ad43e0235c6b
Merge: 337f0a5 0cf3875
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 17:56:10 2016 -0400

    Merge branch 'debug_steve'

commit 0cf3875b004115fb9cd4943d5aa0911e72f9e839
Merge: 4356228 337f0a5
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 17:55:54 2016 -0400

    Merge with master

commit 337f0a5466d64508f17ce49b10bb190ebd7df395
Merge: 95f1642 d72dc9f
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 17:31:12 2016 -0400
```

```
    Merge branch 'helloworld'

commit d72dc9fa8678f4dc400584eaf55e355ed4414fc0
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 17:30:45 2016 -0400

    Resolve merge conflict 2

commit 95f16424cd516b51bae91c9fd8076d28c1a949f9
Merge: 5925cfb d115c7e
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Thu Apr 7 17:29:06 2016 -0400

    Merge pull request #8 from lemeb/helloworld

    Helloworld becomes the stable infra

commit d115c7ee0882bcb039b22463efc269a42d42903c
Merge: dc906e1 5925cfb
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Thu Apr 7 17:28:04 2016 -0400

    Resolve merge conflict

commit dc906e1105b6b830e6bc39ea4b14119b5f7d5c5c
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed Apr 6 18:51:00 2016 -0400

    It works! Also, you can put as many blank lines as you want! Also, you don
       't need to write a main function!

commit 97105faebf697f54dfe5eb99f8731605232867c0
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed Apr 6 15:04:14 2016 -0400

    Better indentation

commit 969c45bd1fd7f0b720d384a1c032f2ac1a2fa18b
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed Apr 6 09:11:35 2016 -0400

    hello world works

commit 5fb60fecfa012baafcc650d3323232c4c7dc3619
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed Apr 6 00:04:14 2016 -0400

    hello world with ll working

commit 6b08cddfd7d9510377afc7cb7c9bc785ff75f14c
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Wed Apr 6 00:03:54 2016 -0400
```

```
    hello world without ll working

commit 4356228f8507463da3fc56f4715db2bcfddf10e8
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed Apr 6 03:36:47 2016 +0000

    Beginning

commit ca411a91e113f52d73636cd40bea0e37c094860e
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Apr 5 22:39:34 2016 -0400

    working codegen

commit 82ce18af940bebea4f11245f1f050ba4815a8e0b
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Apr 5 21:58:23 2016 -0400

    debugging stuff

commit 5925cfbe33fb01fc02145fdc2974e7f826307d83
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Wed Apr 6 01:55:17 2016 +0000

    Modified the scolkam.ml file to check if it works (it doesn't)

commit a0fd58d6ee8bacb93ea7447fcb29fbeffb6950b6
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Tue Apr 5 19:48:47 2016 -0400

    attempt at hello

commit 484eaa7d50489d8e95a35f7d0d7abb54c90a73b2
Author: KathyFomin <kathy.fomin@gmail.com>
Date:   Mon Apr 4 18:41:49 2016 -0400

    minor changes to comply with LRM

    specifying type for variables, making bool values start with a capital
    letter, replacing ^ (power) symbol with * (multiplication)

commit 7d96ac1af98d2d95a3df801df8006d301e238815
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Apr 3 18:53:13 2016 -0400

    adding semantic analyzer version 1

commit 61aecfebd454abed85d8b03674ab77712f1cd91f
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Apr 3 20:49:14 2016 +0000

    Code cleaning so that codegen.ml works

commit eb75a4d2fae873fbca79a94ea5ba0385e6d4377e
```

```
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Apr 3 20:22:28 2016 +0000

    Added *.err in the make clean

commit e5fde5f47437b14d47930d6cbecdf20debc28360
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Apr 3 13:36:13 2016 -0400

    fixing class issue in parser

commit 24491857bd87eba3213d32937ba628af3d94a810
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Apr 3 13:27:33 2016 -0400

    tuple tests

commit 048ce83b0d16d3b1c967bb974bb4868801ca8cde
Merge: 0b84cd0 c5bdb3e
Author: Oneill38 <megan.oneill38@gmail.com>
Date:   Sun Apr 3 12:40:41 2016 -0400

    Merge pull request #7 from lemeb/tests

    Tests

commit c5bdb3eb0563cc3014bbbba1f18a3c559a827a75
Merge: 456f1a4 0b84cd0
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Apr 3 12:37:53 2016 -0400

    Merge branch 'master' into tests

    Conflicts:
        tests/test-misc_test.sco

commit 0b84cd091b338babf17b2e67efa7db25ecf10952
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Sat Apr 2 00:17:42 2016 -0400

    addressing TA's comments

commit dddffec69ac76379dc61283031d75198b35649d3
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Fri Apr 1 23:48:46 2016 -0400

    adding parentheses for print and type for class declaration

commit a9c89766b06052920f2221af5e6dcc4ec89e4d6d
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Fri Apr 1 22:11:55 2016 -0400

    adding parentheses for print and type for tuple declaration
```

```
commit b76867faa44e68092144b0a0387afe6bed7833c1
Author: Kathy Fomin <kathy@kathys-air.nirvanasoft.com>
Date:   Fri Apr 1 16:47:52 2016 -0400

    added parentheses for print

    all functions need to be surrounded with parentheses

commit 456f1a427cc95667086d38ebd146e47867e121a2
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 1 16:06:50 2016 -0400

    rename new tests, add to makefile

commit 2b40db209ccf076fd19031b1bf327a97eff85fc0
Merge: 914d59e 91a1743
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 1 15:39:13 2016 -0400

    Merge branch 'token_check' into tests

commit 91a17437b75f4987f80ef2fa391e8707bc876734
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 1 15:38:42 2016 -0400

    updated test syntax

commit 6e416de70cb35adaa16c260098752bb9c18b904d
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Fri Apr 1 15:37:27 2016 -0400

    add shell script to test parser/scanner

commit a1e66fb9bd85ee696a1ebf7378ed38f9d1fbedcb
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Fri Apr 1 01:59:06 2016 -0400

    adding tuples, element access and list declarations

commit 914d59e8fdc2f50936a1029e2331880fd233cc2a
Merge: 38ad827 a109889
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Tue Mar 29 19:39:07 2016 -0400

    Merge branch 'master' into tests

commit a109889c890b214d4a1d8193836acccc4995e819
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Mar 29 16:40:06 2016 -0400

    fixing empty line issue in parser and changing a few test files to match
       scolkam syntax

commit 95ed676941d6fa7ad7ae98a716f3c682374c46e8
```

```
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Tue Mar 29 13:15:08 2016 -0400

    added test log to see what issues are

commit 6c8a05c12765d76b2457008bb1b497edc73846e4
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Tue Mar 29 12:58:32 2016 -0400

    added ability to test files

commit a650b0e5e6f0c6364922e793144fde9ee442f47b
Merge: acc7cb1 9ccaa2f
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Tue Mar 29 02:29:57 2016 -0400

    Merge pull request #6 from lemeb/parser

    Improving on scanner-parser testing

commit 9ccaa2f6c6a6275326ef15d3575a13fe69970ec9
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Mar 29 02:17:55 2016 -0400

    a better parser-scanner tester

commit 658eb9164b7a1e950d4e97e6ee2e468b12b0ba7d
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Mon Mar 28 17:07:06 2016 -0400

    parser testing script

commit acc7cb133d3ea387b55ddd176b2ab1f807543878
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Mon Mar 28 20:56:21 2016 +0000

    Added the first REPL instructions that you need to execute to have LLVM
       and the AST working

commit fefa8b635f03aa769691215858ed0a73051cc62a
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Mon Mar 28 20:44:12 2016 +0000

    Tasklist for Semantic Analyser

commit 38ad8278bde51ef3ddb0cbe8082896d5f85b3267
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Mon Mar 28 15:49:11 2016 -0400

    arith & augment tests

commit e95f658cd3558fad4d5e9e79c756700fdd611593
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Mar 27 21:38:28 2016 +0000
```

```
    Housekeeping (the codegen does not have any syntax errors anymore) and
       added function calls

commit a0954f78047732a420de84b422a019488f51fdef
Merge: ad102f3 1b3f3a5
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Mar 27 13:19:50 2016 -0400

    Added unary operations

commit ad102f3795e13300090468fe5d33f2b6db0113a4
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Mar 27 13:18:50 2016 -0400

    Added unary operations

commit 1b3f3a5ad3e519287546345028c669ddb670e091
Merge: 95a12db 0d15ce2
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Mar 27 13:01:17 2016 -0400

    Merge branch 'codeGenTypes'

commit 0d15ce213a9a989c5af6ef28186284bb2813e41c
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Mar 27 13:00:51 2016 -0400

    added string and one types

commit 525e6a4016f77ece83d7d2516e86a34831134330
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Mar 27 12:51:22 2016 -0400

    adding a token generating version of the scanner

commit 95a12dbc6743f6cbca2735f2cfd480b46f5f3953
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Sun Mar 27 12:48:22 2016 -0400

    Removed Physical Equality

commit 84ecf33672f147681a30d8a7e331de88dea8a71f
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Fri Mar 25 04:09:52 2016 +0000

    Began the Code Generation!

commit c3b9a594500949b5f7da37a1765d3860bb8c1b58
Merge: b8d8e56 7b3197b
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Thu Mar 24 15:30:19 2016 -0400

    Merge pull request #5 from lemeb/tests
```

```
    Tests

commit b8d8e561e00946a2cc830eb12812012e2c43ff70
Merge: 781607e 5d4d991
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Thu Mar 24 15:29:57 2016 -0400

    Merge pull request #4 from lemeb/parser

    Parser

commit 7b3197bfdade5775f565a4fa085d204a4ef14465
Merge: ecdff02 8db90db
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Thu Mar 24 11:24:53 2016 -0400

    Merge branch 'tests' of github.com:lemeb/scolkam into tests

commit ecdff02477b9de3664000a1088ea01426c064c8b
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Thu Mar 24 11:24:35 2016 -0400

    another round of tests

commit 8db90db35c69b79b9ed51157f757badb096748bc
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Thu Mar 24 00:09:50 2016 -0400

    changes to comply with LRM

commit 725bb8dc04f60c701335cb94b1f3f31b694c291f
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Wed Mar 23 23:51:16 2016 -0400

    removed colon in function argument list

commit 5d4d99116e6336720e2e2cfad9cf1c8c2ac13b13
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Tue Mar 22 23:51:07 2016 -0400

    bug fixes for scanner and parser

    scanner: removed 'super', fixed float;
    parser: added 'FILE' token

commit 34f063f4bcfbcd6eb4478074a98c932d2288c717
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Tue Mar 22 19:05:05 2016 -0400

    removed keyword 'tuple'

    removed keyword 'tuple' as it is not really used anywhere
```

```
commit 3487cd7cb4d7a3696c0222b67571c6cc6c1922f8
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Mar 22 18:45:51 2016 -0400

    adding tuples

commit e7d138291e7647ff8dd48608d7e767aef1c81c13
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Tue Mar 22 18:31:37 2016 -0400

    Removed unused tokens

    Removed unused tokens - curly brackets and decorator

commit 36004739c39dded584861322aeb7139219a458c8
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Tue Mar 22 17:58:39 2016 -0400

    few fixes on ast.ml

commit 781607ecc7f12fc15e5aa7694efad2fc5f01c095
Merge: 542a335 9b21508
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Tue Mar 22 17:18:07 2016 -0400

    Merge pull request #3 from lemeb/parser

    adding dot notation

commit 9b215080afe1c59e9a5ce5443f1bcf4cd2de598f
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Mon Mar 14 22:23:12 2016 -0400

    adding dot notation

commit a59d411b11fdceefa631b523e950419d2066b7f5
Merge: 496d2cb 542a335
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Thu Mar 10 18:45:00 2016 -0500

    Merge branch 'master' into tests

commit 496d2cb028925d7a47b1f5c202af6f88758aa456
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Thu Mar 10 18:44:41 2016 -0500

    round two microc tests

commit 542a335cea2a46f389bf515c65f2e2ea03326fd6
Merge: 609a223 ba0c5e9
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Wed Mar 9 22:38:03 2016 -0500

    Merge pull request #2 from lemeb/parser
```

```
    Few changes to match LRM

commit ba0c5e934895e4ff78b1b222f36fb1694d02b0eb
Merge: 7ab38b8 ec51d98
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Wed Mar 9 22:34:01 2016 -0500

    Merge branch 'parser' of https://github.com/lemeb/scolkam into parser

    adding changes from the lrm

commit 7ab38b83c7bda3010caffc699931a563a2aa72e9
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Wed Mar 9 22:30:13 2016 -0500

    incorporating changes from the lrm

commit a68a3a9560a1df7b8aba8ba6b9e586154e9fba87
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Mon Mar 7 23:20:18 2016 -0500

    first round of microC tests

commit ca67eaf2d656a94bec0f449bbf4336f2e74636a9
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Mon Mar 7 22:31:27 2016 -0500

    first pass at bst & avl classes

commit ec51d98def40eb882240761f11d1c912130c75e4
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Mon Mar 7 21:59:18 2016 -0500

    Added LRM and cleaning

commit 9b88565fd9df1808879077a412cf880480c0995f
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Mon Mar 7 21:58:12 2016 -0500

    Added LRM

commit 38ba8e6f1e8d83af1d2f93f77a4f0d67afb0eb50
Author: Leopold Mebazaa <lm3037@columbia.edu>
Date:   Mon Mar 7 21:27:12 2016 -0500

    Added Progress Update

commit 609a223b1b292742a77795e7ff580cf8d6f83cc0
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Sun Mar 6 20:46:14 2016 -0500

    Minor changes after review
```

```
    Minor changes after review

commit ad60c792e1f1f3e8a77fd8316f564f222babcfe8
Merge: cc96095 bd67508
Author: Steve Cheruiyot <skc2143@columbia.edu>
Date:   Sun Mar 6 13:35:33 2016 -0500

    Merge pull request #1 from lemeb/parser

    parser accepting classes, functions variable declarations and statements

commit bd67508cc8c4614ed6750c63bc5e01cf96783452
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Mar 6 03:05:47 2016 -0500

    adding classes to parser, string and float literals to scanner

commit cc96095d4037fa79edca53b9d692d121dc3c4b21
Author: Kathy Fomin <kathy@Kathys-MacBook-Air.local>
Date:   Mon Feb 29 12:37:11 2016 -0500

    LRM - Initial commit

    Completed Standard Library. Remaining chapters: Classes and Statements.

commit 8832430b879207efbad70893a81c88fdbfd3179e
Merge: dcbb2d4 2a1dbc6
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Feb 28 19:18:35 2016 -0500

    Merge branch 'tests'

commit 2a1dbc6ecd47ce33948e15c745e31fc9b2d5d290
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Feb 28 19:17:22 2016 -0500

    super basic ll

commit 3b113749baa21ac14f7aaef0320cd412f4da1324
Author: Megan O'Neill <mo2638@columbia.edu>
Date:   Sun Feb 28 18:30:59 2016 -0500

    calculator plus misc tests

commit a82e8879c8d8413e8dfed9d9b8a58efc46479058
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Feb 28 16:18:28 2016 -0500

    declaration --> assignment

commit 226c7e14c05831b0cfa19bfe9b0206b778fa6595
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Feb 28 15:26:58 2016 -0500
```

```
    remove type declaration from functions, remove parentheses from if
       statements, use 'end' keyword to delineate if/else/while and function
       blocks

commit 771ee2f70391c84ca3da10434f183d0f6667734e
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Feb 28 14:38:13 2016 -0500

    parser is accepting basic function declaration

commit 44d26900ee65f70f81e98ac5dc8ab6dca24af82b
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Feb 28 14:19:30 2016 -0500

    changed VOID to NONE

commit 89394683cd78de5373d23f5e57dd830250ebc48c
Author: Connor Hailey <connorhailey@gmail.com>
Date:   Sun Feb 28 14:17:46 2016 -0500

    initial setup of microc derivative with some scolkam syntax

commit dcbb2d47d79e24096ed1a10a0cc846a35f355238
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Feb 28 19:03:52 2016 +0000

    adding end token

commit 541dde216cc86497d0a245ecb1ed8be81ec2b276
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Feb 28 18:58:01 2016 +0000

    adding scanner.mll

commit eac7f4ce85d454344eb63dde202a1d1748ec49e4
Author: Steve Cheruiyot <scheruiyot@columbia.edu>
Date:   Sun Feb 28 17:02:16 2016 +0000

    adding scanner

commit 3ed7695d0bb9b873dc4486794d07f5210ab7d12c
Author: Kathy Fomin <kathy.fomin@gmail.com>
Date:   Fri Feb 19 16:30:25 2016 -0500

    Revised Proposal

    PDF version

commit 0c1e1bb25978700af3bd36ecaae218c7428dfe56
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Fri Feb 19 15:54:57 2016 -0500

    Revised the proposal
```

```
commit 5f5e4872f9d28251511f3dbf0389d06c630f567a
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Fri Feb 19 15:50:28 2016 -0500

    Added the Revised Proposal

commit ceb7a94556b8e7bf668a24f707a6a5b64e3e1215
Author: Leopold Mebazaa <lemebfr@gmail.com>
Date:   Thu Feb 18 21:25:17 2016 -0500

    Initial commit
```

# Chapter 5

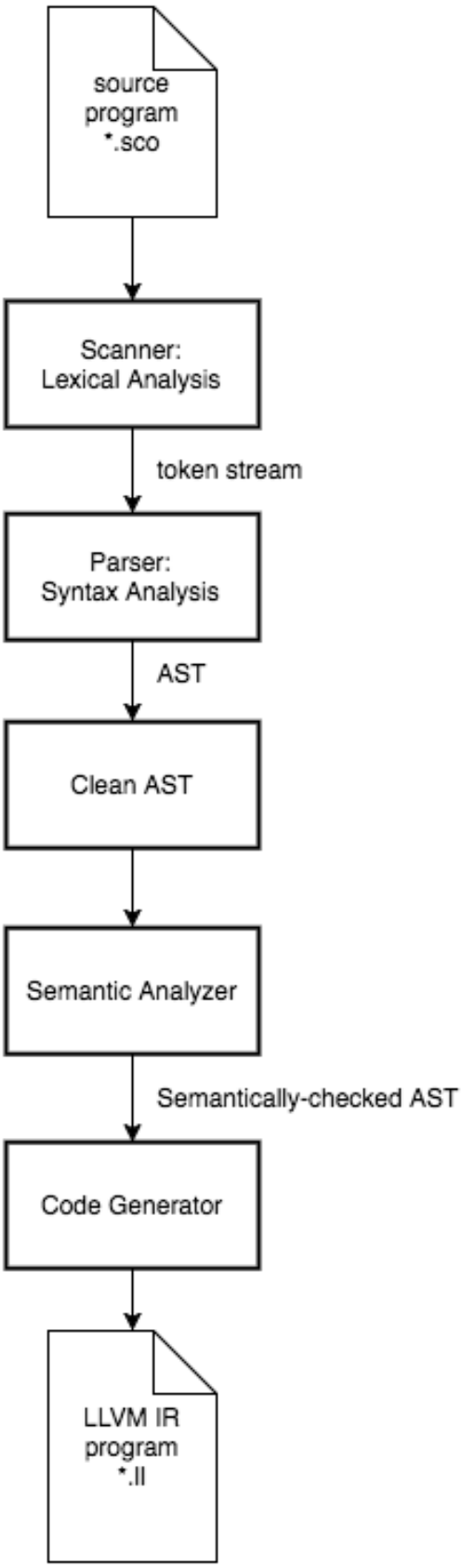
# Architectural Design

## 5.1 Overall architecture

The compiling pipeline is composed of the following modules:

- `scanner.ml` reads the different tokens of the input,
- `parser.mly` transforms this series of token into an Abstract Syntax Tree,
- which `AST.ml` defines (the file also makes some pretty printing functions available),
- `clean_ast.ml` makes a few transformations to the AST so that it is easier to generate code; for instance, it makes that all the statements recored outside of functions are regrouped into a `main` function, which LLVM requires,
- `semant.ml` checks that the AST is proper and that, for instance, type mismatches will not occur, and
- `codegen.ml` generates the LLVM code.

## 5.2 Schema of the architecture


source
program
*.sco
Scanner:
Lexical Analysis
token stream
Parser:
Syntax Analysis
AST
Clean AST
Semantic Analyzer
Semantically-checked AST
Code Generator
LLVM IR
program
*.ll

# Chapter 6

# Test Plan

A variety of testing techniques were utilized to create a robust testing environment for the Scolkam language. Various testing techniques utilized include: testing automation, unit tests, integration tests, positive testing, and negative testing.

Prior to actually testing the code output by the code generator, the parser and scanner was first tested. We created a script to run all the tests through the parser and scanner first, and output if they passed, or if there was a parse error.

Initially our tests are broken up into tests that are meant to pass, and tests that are meant to fail, thereby utilizing positive and negative testing. Negative testing ensures that invalid input is not accepted, and that the language properly rejects the invalid input. Positive testing on the other hand allows us to assess if our assertions and expectations about the language and given programs are correct.

Unit testing was utilized to determine if smaller pieces of our language were behaving as expected, and defined in the Language Reference Manual. Tests were picked out by going through the Language Reference Manual, looking at all the defined features, and creating tests based on those features. Once these smaller tests were verified to be passing, they were then integrated into larger programs, to asses how different pieces of the language behaved together. Integration testing allowed us to slowly piece various units of our language together and assert with confidence that they worked. This combination approach enabled us to write a full fledged demo program.

The automation of testing proved to be an extremely useful tool for developing Scolkam. Automation allowed us to test our entire suite of features, in real time as they were added. We based our automation script off of the one provided in the micro-c compiler code. While team members could see which tests were passing at any time, a weekly list was also sent out with all tests that had previously been passing, to ensure that new code did not break existing code.

The tests were developed by Megan and Kathy, who then reported what was passing and failing to the other team members. The testing automation script was based upon the micro-c testing script, and modified by all team members.

# Chapter 7

# Lessons Learned

## Connor P. Hailey

Having to actually build out the architecture (scanner, parser, ast, semant, codegen) as a opposed to seeing it on a powerpoint flowchart helped to make the concepts concrete. I found doing code generation in LLVM using the Ocaml library was pretty difficult because of how little documentation there is. If I could do it over again, I would choose to generate LLVM IR directly because the resources are much more plentiful.

## Léopold Mebazaa

I did not realize how much, proportionally, is just learning the APIs. I thought that the bulk of the work would be done after we would know how to use LLVM, and that the majority of the work would be to do the conception of the language. Instead, it was mostly struggling to understand a language with a documentation that is confusing and principles that are not easy to understand. OCaml, on the other side, was pretty easy to grasp.

## Megan O'Neill

Building the compiler really taught me that a test driven design approach is best, especially in this case. With so many features to be implemented, it would've been best to write code specifically for the features that we were working on that week, and then waiting until those features were completed, and those tests passed, until writing more tests. Without TDD, it was more difficult to determine which test cases were needed, and which tests were supposed to be passing.

## Steve K. Cheruiyot

Writing a compiler requires careful planning and lots of flexibility as features change along the way. While the limited documentation available for OCaml LLVM bindings posed a challenge, writing parts of both the frontend and the backend of the compiler was a great way to learn and internalize the general structure of the compiler. At the same time I was able to add a powerful functional language, OCaml, to my repertoire.

## Yekaterina Fomina

Since the LRM document is the first formal document describing the language, it is good to have all team members involved in writing it. This will not only ensure that everybody's input is considered but will help to have a clear picture of the intended features of the language. The document ends up being very long and it is unlikely that people will read it in its entirety. This can potentially increase the number of deviations from the original idea.
During 'crunch time', it is very easy to shift your focus on short-term objectives and disregard long-term risks. Spending a bit more time at the beginning of the project on clarifying design aspects of the language

saves a lot of time at the end.
Once all team members agree on the features of the language, the development process should be driven by LRM. If some features listed in LRM cannot be implemented or have to be implemented differently than described in LRM, then it has to be communicated and the updates to LRM need to be done right away before it is forgotten. This will ensure that the document is always up-to-date and team members responsible for development and testing have confidence that they can rely on the document at any given time.

# Chapter 8

# Appendix

### 8.0.1 scanner.mll

```
(* Ocamllex scanner for Scolkam *)

{
    open Parser

    let unescape s =
        Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

let alpha = ['a'-'z' 'A'-'Z']
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')*
let string = '"' ( (ascii | escape)* as s) '"'
let char = ''' ( ascii | digit ) '''
let float = (digit+) ['.'] digit+
let int = digit+
let whitespace = [' ' '\t' '\r']
let return = '\n'

rule token = parse
  whitespace { token lexbuf }
| '#'        { single_comment lexbuf }
| "\'\'\'"   { multi_comment1 lexbuf }
| "\"\"\""   { multi_comment2 lexbuf }
| '('        { LPAREN }
| ')'        { RPAREN }
| '['            { LBRACKET }
| ']'            { RBRACKET }
| ';'        { SEMI }
| ':'        { COLON }
| '\n'       { EOL }
| ','        { COMMA }

(* Operators *)
```

```
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MODULUS }
| "+="     { PLUSEQ }
| "-="     { MINUSEQ }
| "*="     { TIMESEQ }
| "/="     { DIVIDEEQ }
| "%="     { MODULUSEQ }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "and"    { AND }
| "or"     { OR }
| "not"    { NOT }
| "in"     { IN }

(* Control flow *)
| "if"       { IF }
| "elif"     { ELIF }
| "else"     { ELSE }
| "for"      { FOR }
| "while"    { WHILE }
| "break"    { BREAK }
| "continue" { CONTINUE }
| "return"   { RETURN }
| "end"      { END }

(* Data types and atom*)
| "int"      { INT }
| "float"    { FLOAT }
| "str"      { STRING }
| "bool"     { BOOL }
| "True"     { TRUE }
| "False"    { FALSE }
| "None"     { NONE }
| "tuple"        { TUPLE }

(* Functions and Classes and object management *)
| "def"    { FUNCTION }
| "class"  { CLASS }
| "new"    { NEW }

| int as lxm        { INT_LITERAL(int_of_string lxm) }
| float as lxm      { FLOAT_LITERAL(float_of_string lxm) }
| string            { STRING_LITERAL( (unescape s) ) }
| id as lxm         { ID(lxm) }
| eof               { EOF }
| _ as char { raise (Failure("SyntaxError: Invalid syntax -> " ^ Char.escaped
```

```
   char)) }

and multi_comment1 = parse
  "\'\'\'" { token lexbuf }
| _        { multi_comment1 lexbuf }

and multi_comment2 = parse
  "\"\"\"" { token lexbuf }
| _        { multi_comment2 lexbuf }

and single_comment = parse
  '\n' { token lexbuf }
| _    { single_comment lexbuf }
```

### 8.0.2 parser.mly

```
        /* Ocamlyacc parser for Scolkam */

%{
  open Ast
%}

%token CLASS FUNCTION NEW
%token SEMI LPAREN RPAREN COMMA COLON LBRACKET RBRACKET
%token PLUS MINUS TIMES DIVIDE MODULUS ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token PLUSEQ MINUSEQ DIVIDEEQ TIMESEQ MODULUSEQ
%token END RETURN IF ELIF ELSE FOR WHILE IN BREAK CONTINUE
%token INT BOOL FLOAT STRING NONE TUPLE
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token EOL EOF

%right ASSIGN
%left DOT
%left OR
%left AND
%right PLUSEQ MINUSEQ DIVIDEEQ TIMESEQ MODULUSEQ
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULUS
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }
```

```
/* (stmt list * var_decl list) * (func_decl list * class_decl list) */
decls:
               { ([], []), ([], []) }
 | decls stmt  { ($2 :: fst (fst $1), snd (fst $1)), (fst (snd $1), snd (snd
    $1)) }
 | decls vdecl { (fst (fst $1), $2 :: snd (fst $1)), (fst (snd $1), snd (snd
    $1)) }
 | decls fdecl { (fst (fst $1), snd (fst $1)), ($2 :: fst (snd $1), snd (snd
    $1)) }
 | decls cdecl { (fst (fst $1), snd (fst $1)), (fst (snd $1), $2 :: snd (snd
    $1)) }

/* Class declaration */
cdecl:
  CLASS ID LPAREN objects_opt RPAREN COLON EOL cbody END EOL
  { {
    cname = $2;
    cformals = $4;
    cbody = $8;
  } }

cbody:
  /* nothing */ { { vdecls = []; stmts = []; funcs = []; } }
  | cbody vdecl { { vdecls = $1.vdecls @ [$2]; stmts = $1.stmts; funcs = $1.
     funcs; } }
  | cbody stmt  { { vdecls = $1.vdecls; stmts = $1.stmts @ [$2]; funcs = $1.
     funcs; } }
  | cbody fdecl { { vdecls = $1.vdecls; stmts = $1.stmts; funcs = $1.funcs @ [
     $2]; } }

objects_opt:
    /* nothing */  { [] }
  | object_list    { List.rev $1 }

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


/* Function declaration */
fdecl:
  FUNCTION typ ID LPAREN formals_opt RPAREN COLON EOL fbody END EOL
  { {
    typ = $2;
    fname = $3;
    formals = $5;
    fbody = $9;
  } }

fbody:
    /* nothing */ { { f_vdecls = []; f_stmts = []; } }
  | fbody vdecl { { f_vdecls = $1.f_vdecls @ [$2]; f_stmts = $1.f_stmts; } }
  | fbody stmt { { f_vdecls = $1.f_vdecls; f_stmts = $1.f_stmts @ [$2]; } }
```

```
formals_opt:
    /* nothing */  { [] }
  | formal_list    { List.rev $1 }

formal_list:
    typ ID                   { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
    INT      { Int }
  | FLOAT    { Float }
  | BOOL     { Bool }
  | STRING   { String }
  | NONE     { None }
  | typ TUPLE  { Tuple($1, 0) }

/* Variable Declaration */
vdecl:
   typ ID ASSIGN expr EOL { match $1 with
                              Tuple(ty, _) ->
                                (match $4 with
                                  TupleLit l -> (Tuple(ty, List.length l), $2,
                                      $4)
                                  | Call(_,_) -> (Tuple(ty, 0), $2, $4)
                                  | _ -> raise (Failure "You can only define a
                                      tuple with an expression with a tuple
                                     grammar.") )
                            | _ -> ($1, $2, $4)  }

/* Statements */

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    EOL                                     { Nostmt }
  | expr EOL                                { Expr $1 }
  | BREAK EOL                               { Break }
  | CONTINUE EOL                            { Continue }
  | RETURN EOL                              { Return Noexpr }
  | RETURN expr EOL                         { Return $2 }
  | IF internal_if EOL                      { $2 }
  | vdecl                                   { Declaration($1) }
/* Before:
  If (Expr, Stmts to execute if yes, Stmts to execute if no)

Now:
  If (Expr, List of Stmts to execute if yes, [List of Elifs(expr, List of
     stmts to execute if yes)], List of Stmts to execute if no) */

  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN internal_block EOL
                                            { For($3, $5, $7, $9) }
  | FOR expr IN expr internal_block EOL     { ForIn($2, $4, $5) }
```

```
  | WHILE expr internal_block EOL           { While($2, $3) }
  | expr IN expr COLON EOL                  { In($1, $3)}

internal_block:
  | COLON EOL stmt_list END                 { Block(List.rev $3) }

internal_if:
  | expr COLON EOL stmt_list elif_list else_list END
                                            { If($1, Block(List.rev $4),
                                              Block(List.rev $5), Block(List.
                                                  rev $6)) }

elif_list:
    /* nothing */  { [] }
  | elif elif_list { $1 :: $2 }

elif:
    ELIF expr COLON EOL stmt_list          { Elif($2, Block(List.rev $5)) }

else_list:
    /* nothing */      { [] }
  | ELSE COLON EOL stmt_list  { $4 }

/* Expressions */
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    INT_LITERAL                 { IntLit($1) }
  | FLOAT_LITERAL               { FloatLit($1) }
  | STRING_LITERAL              { StringLit($1) }
  | TRUE                        { BoolLit(true) }
  | FALSE                       { BoolLit(false) }
  | ID                          { Id($1) }
  | expr PLUS    expr           { Binop($1, Add,   $3) }
  | expr MINUS   expr           { Binop($1, Sub,   $3) }
  | expr TIMES   expr           { Binop($1, Mult,  $3) }
  | expr DIVIDE  expr           { Binop($1, Div,   $3) }
  | expr MODULUS expr           { Binop($1, Mod,   $3) }
  | ID PLUSEQ    expr           { let id1 = Id($1) in
                                   Assign($1, Binop(id1, Add, $3)) }
  | ID MINUSEQ   expr           { let id1 = Id($1) in
                                   Assign($1, Binop(id1, Sub, $3)) }
  | ID TIMESEQ   expr           { let id1 = Id($1) in
                                   Assign($1, Binop(id1, Mult, $3)) }
  | ID DIVIDEEQ  expr           { let id1 = Id($1) in
                                   Assign($1, Binop(id1, Div, $3)) }
  | ID MODULUSEQ expr           { let id1 = Id($1) in
                                   Assign($1, Binop(id1, Mod, $3)) }
  | expr EQ      expr           { Binop($1, Equal, $3) }
  | expr NEQ     expr           { Binop($1, Neq,   $3) }
  | expr LT      expr           { Binop($1, Less,  $3) }
  | expr LEQ     expr           { Binop($1, Leq,   $3) }
```

```
  | expr GT      expr             { Binop($1, Greater, $3) }
  | expr GEQ     expr             { Binop($1, Geq,   $3) }
  | expr AND     expr             { Binop($1, And,   $3) }
  | expr OR      expr             { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG          { Unop(Neg, $2) }
  | NOT expr                      { Unop(Not, $2) }
  | ID ASSIGN expr                { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN  { Call($1, $3) }
  | LPAREN expr RPAREN               { $2 }
  | LPAREN actuals_list RPAREN       { TupleLit(List.rev $2) }
  | ID LBRACKET expr RBRACKET        { Element($1, $3) }
  | NONE                             { Noexpr }

actuals_opt:
    /* nothing */ { [] }
  | expr          { [$1] }
  | actuals_list  { List.rev $1 }

actuals_list:
    actual_sublist COMMA expr { $3 :: $1 }
  | actual_sublist COMMA      { $1 }

actual_sublist:
    expr                      { [$1] }
  | actual_sublist COMMA expr { $3 :: $1 }
```

### 8.0.3 ast.ml

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

type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater |
   Geq |
          And | Or

type uop = Neg | Not

type expr =
    IntLit of int
  | FloatLit of float
  | StringLit of string
  | BoolLit of bool
  | TupleLit of expr list
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Element of string * expr
  | Noexpr

type typ = Int | Float | Bool | String | None | Tuple of typ * int

type bind = typ * string
```

```
type var_decl = typ * string * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt * stmt
  | Elif of expr * stmt
  | For of expr * expr * expr * stmt
  | ForIn of expr * expr * stmt
  | While of expr * stmt
  | HiddenWhile of expr * stmt * stmt
  | In of expr * expr
  | Declaration of var_decl
  | Break
  | Continue
  | Nostmt

type func_body = {
    f_vdecls: var_decl list;
    f_stmts: stmt list;
  }

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    fbody : func_body;
  }

type class_body = {
    vdecls: var_decl list;
    stmts: stmt list;
    funcs: func_decl list;
  }

type class_decl = {
    cname : string;
    cformals : typ list;
    cbody : class_body;
  }

type program = (stmt list * var_decl list) * (func_decl list * class_decl list
   )

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
```

```
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "and"
  | Or -> "or"

let string_of_uop = function
    Neg -> "-"
  | Not -> "not"

let rec string_of_expr = function
    IntLit(i) -> string_of_int i
  | FloatLit(f) -> string_of_float f
  | StringLit(s) -> s
  | BoolLit(true) -> "True"
  | BoolLit(false) -> "False"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
      ")"
  | TupleLit(el) -> "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")
     "
  | Element(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) -> String.concat "" (List.map string_of_stmt stmts) ^ "end\n"
  | Expr(expr) -> string_of_expr expr ^ "\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ "\n";
  | If(e, s1, elifs, s2) ->  "if " ^ string_of_expr e ^ ":\n" ^
      string_of_stmt s1 ^ string_of_stmt elifs ^ "else:\n" ^ string_of_stmt s2
  | Elif(e, s1) -> "else if " ^ string_of_expr e ^ ":\n" ^ string_of_stmt s1
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ "):\n " ^ string_of_stmt s
  | While(e, s) -> "while " ^ string_of_expr e ^ ":\n " ^ string_of_stmt s
  | HiddenWhile(_, _, _) -> ""
  | ForIn(e1, e2, s) -> "for " ^ string_of_expr e1 ^ " in " ^ string_of_expr
     e2 ^ ":\n" ^ string_of_stmt s
  | In(e1, e2) -> string_of_expr e1 ^ " in " ^ string_of_expr e2 ^ ":\n"
  | Break -> "break"
  | Continue -> "continue"
  | Declaration(typ, id, expr) ->
        let expr = string_of_expr expr in
        let typ = string_of_typ typ in
        "(DECLARATION: " ^ typ ^ " " ^ id ^
        ", EXPR: " ^ expr ^ ")"
  | Nostmt -> ""
```

```
and string_of_typ = function
    Int    -> "Int"
  | Float  -> "Float"
  | Bool   -> "Bool"
  | String -> "Str"
  | None   -> "None"
  | Tuple(t, l)  -> "Tuple[length "^string_of_int l^"]("^string_of_typ t^")"

let string_of_vdecl (t, id, e) = string_of_typ t ^ " " ^ id ^ " = " ^
   string_of_expr e

let string_of_fdecl fdecl =
    "def " ^ string_of_typ fdecl.typ ^ " " ^
    fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
    "):\n" ^
    String.concat "" (List.map string_of_vdecl fdecl.fbody.f_vdecls) ^
    String.concat "" (List.map string_of_stmt fdecl.fbody.f_stmts) ^
    "end\n"

let string_of_cdecl cdecl =
    "class " ^ cdecl.cname ^ "(" ^ String.concat ", " (List.map string_of_typ
       cdecl.cformals) ^
    "):\n" ^
    String.concat "" (List.map string_of_vdecl cdecl.cbody.vdecls) ^
    String.concat "" (List.map string_of_stmt cdecl.cbody.stmts) ^
    String.concat "" (List.map string_of_fdecl cdecl.cbody.funcs) ^ "end\n"

let string_of_program (stmts_and_vdecls, funcs_and_classes) =
  String.concat "\n" (List.map string_of_stmt (fst stmts_and_vdecls)) ^ "\n" ^
  String.concat "\n" (List.map string_of_vdecl (snd stmts_and_vdecls)) ^ "\n"
     ^
  String.concat "\n" (List.map string_of_fdecl (fst funcs_and_classes)) ^ "\n"
      ^
  String.concat "\n" (List.map string_of_cdecl (snd funcs_and_classes))

let rec pt n = match n with 0 -> "\n" | _ -> (pt (n - 1)) ^ "\t"

let rec abstract_expr n = function
    IntLit(i) ->
      "int " ^ string_of_int i
  | FloatLit(f) ->
      "float " ^ string_of_float f
  | StringLit(s) ->
      "string \"" ^ s ^ "\""
  | BoolLit(true) ->
      "bool " ^ "True"
  | BoolLit(false) ->
      "bool " ^ "False"
  | Id(s) ->
      "id " ^ s
  | Binop(e1, o, e2) ->
      (pt n) ^ "binop (" ^
        abstract_expr (n + 1) e1 ^ " "
        ^ string_of_op o ^ " " ^
```

```
      abstract_expr (n + 1) e2 ^ ")"
  | Unop(o, e) ->
     (pt n) ^ "unop  (" ^
      string_of_uop o ^ " " ^ abstract_expr (n + 1) e ^ ")"
  | Assign(v, e) ->
     (pt n) ^ "assign (" ^ v ^ " = " ^
      abstract_expr (n + 1) e ^ ")"
  | Call(f, el) ->
    let abx x = (abstract_expr (n + 1) x) in
     (pt n) ^ "call (" ^ f ^
        "(args: " ^ String.concat ", " (List.map abx el) ^ ")" ^ ")"
  | TupleLit(el) ->
    let abx x = (abstract_expr (n + 1) x) in
     (pt n) ^ "Tuple(" ^ String.concat ", " (List.map abx el) ^ ")"
  | Element(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
  | Noexpr -> ""


let rec abstract_stmt n = function
    Block(stmts) ->
      let abm x = (abstract_stmt (n + 1) x) in
      let content = String.concat "" (List.map abm stmts) in
        (pt n) ^ "(BLOCK: " ^ content ^ ")"
  | Expr(expr) ->
       let content = abstract_expr (n+1) expr in
       (pt n) ^ "(EXPR: " ^ content ^ ")"
  | Return(expr) ->
       let content = abstract_expr (n+1) expr in
       (pt n) ^ "(RETURN: " ^ content ^ ")"

  | If(e, s1, elifs, s2) ->
       let condition  = abstract_expr (n+1) e in
       let then_stmts = abstract_stmt (n+2) s1 in
       let elseifs    = abstract_stmt (n+2) elifs in
       let else_stmts = abstract_stmt (n+2) s2 in
       (pt n) ^ "(IF: " ^
       (pt (n+1)) ^ "(CONDITION: " ^ condition ^ ")" ^
       (pt (n+1)) ^ "(THEN: " ^ then_stmts ^ ")" ^
       (pt (n+1)) ^ "(ELSE IF: " ^ elseifs ^ ")" ^
       (pt (n+1)) ^ "(ELSE: " ^ else_stmts ^ ")"
       ^ ")"

  | Elif(e, s1) ->
       let condition  = abstract_expr (n+1) e in
       let then_stmts = abstract_stmt (n+1) s1 in
       (pt n) ^ "(ELIF: " ^
       "(CONDITION: " ^ condition ^ ")" ^
     (pt n) ^ "(THEN: " ^ then_stmts ^ ")"
       ^ ")"

  | For(e1, e2, e3, s) ->
       let content_beg = abstract_expr (n+1) e1 in
       let condition   = abstract_expr (n+1) e2 in
       let iteration   = abstract_expr (n+1) e3 in
```

```
        let stmts       = abstract_stmt (n+2) s in
        (pt n) ^ "(FOR: " ^ "\n" ^
        (pt (n+1)) ^ "(BEGIN: "      ^ content_beg ^ ")" ^
        (pt (n+1)) ^ "(CONDITION: "  ^ condition ^ ")" ^
        (pt (n+1)) ^ "(ITERATION: "  ^ iteration ^ ")" ^
        (pt (n+1)) ^ "(STATEMENTS: " ^ stmts ^ ")" ^
        ")"

  | While(e, s) ->
        let condition   = abstract_expr (n+1) e in
        let stmts       = abstract_stmt (n+2) s in
        (pt n) ^ "(WHILE: " ^ "\n" ^
        (pt (n+1)) ^ "(CONDITION: "  ^ condition ^ ")" ^
        (pt (n+1)) ^ "(STATEMENTS: " ^ stmts ^ ")" ^
        ")"
  | HiddenWhile(_, _, _) -> ""
  | ForIn(e1, e2, s) ->
        let expr   = abstract_expr (n+1) e1 in
        let tuple  = abstract_expr (n+1) e2 in
        let stmts  = abstract_stmt (n+2) s in
        (pt n) ^ "(FORIN: " ^
                 "(EXPR: "  ^ expr ^ ")" ^
        (pt (n+1)) ^ "(ITERABLE LIST: "  ^ tuple ^ ")" ^
        (pt (n+1)) ^ "(STATEMENTS: " ^ stmts ^ ")" ^
        ")"
  | Declaration(typ, id, expr) ->
        let expr = abstract_expr (n+1) expr in
        let typ = string_of_typ typ in
        (pt n) ^ "(DECLARATION: " ^ typ ^ " " ^ id ^
        ", EXPR: " ^ expr ^ ")"
  | In(_, _) -> "IN"
  | Break -> (pt n) ^ "BREAK"
  | Continue -> (pt n) ^ "CONTINUE"
  | Nostmt -> (pt n) ^ "NOSTMT"


let abstract_func n fdecl =
    let abm x = (abstract_stmt (n + 1) x) in
    let args = String.concat ", " (List.map snd fdecl.formals) in
    (pt n) ^ "(FUNCTION. TYPE: " ^ string_of_typ fdecl.typ ^ ", NAME:" ^
    fdecl.fname ^ ", ARGS: (" ^ args ^ ")" ^
    (pt n) ^ "STATEMENTS: " ^ String.concat "" (List.map abm fdecl.fbody.
       f_stmts) ^
    ")"

let abstract_of_program (stmts_and_vdecls, funcs_and_classes) =
  let abm x = (abstract_stmt 1 x) in
  let abf x = (abstract_func 1 x) in
  let stmts =
    String.concat "\n" (List.map abm (fst stmts_and_vdecls)) in
  let vdecls =
    String.concat ", " (List.map string_of_vdecl (snd stmts_and_vdecls)) in
  let funcs =
    String.concat "\n" (List.map abf (fst funcs_and_classes)) in
```

```
  let classes =
    String.concat "\n" (List.map string_of_cdecl (snd funcs_and_classes)) in
  "(stmts: "   ^ stmts   ^ ") \n" ^
  "(vdecls: "  ^ vdecls  ^ ") \n" ^
  "(funcs: "   ^ funcs   ^ ") \n" ^
  "(classes: " ^ classes ^ ") \n"
```

### 8.0.4 clean_ast.ml

```
open Ast
module S = Semant

let clean ast =

  let rm_nost type_list =
    let type_remove_rec original element = match element with
      | Ast.Nostmt -> original
      | _ -> List.append original [element] in
    List.fold_left type_remove_rec [] type_list in
  let change_function_statment k =
    { Ast.typ = k.Ast.typ;
      Ast.fname = k.Ast.fname;
      Ast.formals = k.Ast.formals;
      Ast.fbody = { Ast.f_vdecls = k.fbody.Ast.f_vdecls;
                Ast.f_stmts = rm_nost k.fbody.Ast.f_stmts}}
  in
  (* Convert global variables to assignemnt statements *)
  let edit_globals g_list =
    let create_stmt vdecl = match vdecl with
      (_, n, e) -> Ast.Expr (Ast.Assign (n, e))
    in
    let rec helper g c = match g with
        [] -> c
      | hd :: tl -> helper tl (create_stmt hd :: c);
    in helper g_list []
  in

  let statements = fst (fst ast) in
  let globals = snd (fst ast) in
  let functions = fst (snd ast) in
  let classes = snd (snd ast) in
  let no_stmt_functions = List.map change_function_statment functions in
  ((statements, globals), (* Statements and globals *)
  ((List.append [{  Ast.typ = Ast.None;
                    fname   = "main";
                    formals = [];
                    fbody   = { Ast.f_vdecls = []; (* empty: this way it doesn
                       't affect scope of globals *)
                              f_stmts = edit_globals globals @ List.rev (
                                 rm_nost statements)}}]
                no_stmt_functions), (* Functions *)
    classes)) (* Classes *)
```

### 8.0.5 semant.ml

```
(* Semantic checking *)

open Ast

exception Error of string

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns NoneType if successful,
    throws an exception if something is wrong.

    Check each global variable, then check each function *)

let check ((_, globals), (functions, classes)) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
        n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a nonetype type *)
  let check_not_none exceptf = function
      (Ast.None, n, _) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  let check_not_none_formals exceptf = function
      (Ast.None, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Raise an exception of the given rvalue type cannot be assigned to
      the given lvalue type *)
  let check_assign lvaluet rvaluet err =
      if lvaluet = rvaluet then lvaluet else raise err
  in

  (**** Checking Global Variables ****)
  List.iter (check_not_none (fun n -> "Illegal NoneType global variable " ^ n)
     ) globals;

  report_duplicate (fun n -> "Duplicate global " ^ n) (List.map (fun (_, n, _)
      -> n) globals);

  (* TODO: Check assign on global variables *)



  (* Function declaration for a named function *)
  let built_in_decls =  StringMap.add "print"
```

```
   { typ = Ast.None; fname = "print"; formals = [(Int, "x")];
     fbody = { f_vdecls = []; f_stmts = [] } } (StringMap.singleton "prints"
  { typ = Ast.None; fname = "prints"; formals = [(String, "x")];
    fbody = { f_vdecls = []; f_stmts = [] } } ) in
let built_in_decls = StringMap.add "toint" { typ = Ast.Int; fname = "toint";
     formals = [(Float, "x")]; fbody = { f_vdecls = []; f_stmts = [] } }
   built_in_decls in
let built_in_decls = StringMap.add "tofloat" { typ = Ast.Float; fname = "
   tofloat"; formals = [(Int, "x")]; fbody = { f_vdecls = []; f_stmts = []
   } }  built_in_decls
  (*(StringMap.singleton "printb" { typ = Ast.None; fname = "printb";
     formals = [(Bool, "x")];
    fbody = { f_vdecls = []; f_stmts = [] } })*)
in

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m
   )
                       built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
      with Not_found -> raise (Failure ("Unrecognized function " ^ s))
in

(****    Check functions    ****)
let check_function func =

  report_duplicate (fun n -> "Duplicate function " ^ n)
    (List.map (fun fd -> fd.fname) functions);

  List.iter (check_not_none_formals (fun n -> "Illegal nonetype formal " ^ n
      ^
    " in " ^ func.fname )) func.formals;

  report_duplicate (fun n -> "Duplicate formal " ^ n ^ " in " ^ func.fname)
    (List.map snd func.formals);

  List.iter (check_not_none (fun n -> "Illegal NoneType local variable " ^ n
      ^
    " in " ^ func.fname)) func.fbody.f_vdecls;

  report_duplicate (fun n -> "Duplicate local variable " ^ n ^ " in " ^ func
     .fname)
    (List.map (fun (_, n, _) -> n) func.fbody.f_vdecls);

  (* Type of each variable (global, formal, or local *)
  let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
    StringMap.empty ((List.map (fun (t, n, _) -> (t, n)) globals) @ func.
       formals @
    (List.map (fun (t, n, _) -> (t, n)) func.fbody.f_vdecls))
  in

  let type_of_identifier s =
    try StringMap.find s symbols
```

```
  with Not_found -> raise (Failure ("Undeclared identifier " ^ s))
in

(* Return the type of an expression or throw an exception *)
let rec expr = function
    IntLit _ -> Int
  | FloatLit _ -> Float
  | BoolLit _ -> Bool
  | StringLit _ -> String
  | Id s -> type_of_identifier s
  | TupleLit l ->
    let first_el = List.hd l in
    let type_el = expr first_el in
    let length = List.length l in
    Tuple (type_el, length)
  | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
    (match op with
        Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int -> Int
      | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Float ->
         Float
      | Add when t1 = String && t2 = String ->
           (match (e1, e2) with
              (Ast.StringLit(_), Ast.StringLit(_)) -> String
            | _ -> raise (Failure ("Only raw strings can be concatenated
               ")))
      | Equal | Neq when t1 = t2 -> Bool
      | Less | Leq | Greater | Geq when t1 = t2 -> Bool
      | And | Or when t1 = Bool && t2 = Bool -> Bool
      | _ -> raise (Failure ("Illegal binary operator " ^ string_of_typ t1
          ^
            " " ^ string_of_op op ^ " " ^ string_of_typ t2 ^ " in " ^
            string_of_expr e))
    )
  | Unop(op, e) as ex -> let t = expr e in
    (match op with
        Neg when t = Int -> Int
      | Neg when t = Float -> Float
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("Illegal unary operator " ^ string_of_uop op
        ^
             string_of_typ t ^ " in " ^ string_of_expr ex)))
  | Assign(var, e) as ex -> let lt = type_of_identifier var
                            and rt = expr e in
            check_assign (type_of_identifier var) (expr e)
            (Failure ("Illegal assignment " ^ string_of_typ lt ^ " = " ^
                      string_of_typ rt ^ " in " ^ string_of_expr ex))
  | Call(fname, actuals) as call ->
      if fname <> "print" then (* The print function can have different
                                  types of arguments *)
     (let fd = function_decl fname in
      if List.length actuals != List.length fd.formals then
        raise (Failure ("Expecting " ^ string_of_int
         (List.length fd.formals) ^ " arguments in " ^ string_of_expr
            call))
```

```
      else
        List.iter2 (fun (ft, _) e -> let et = (match expr e with
                                               (* Since there is a call to
                                               another function, we don't care
                                               about the length *)
                        Ast.Tuple(ty, _) -> Ast.Tuple(ty, 0)
                      |_ -> expr e) in
          ignore (check_assign ft et
            (Failure ("Illegal actual argument found " ^ string_of_typ et
                ^
            " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e))
               ))
          fd.formals actuals;
      fd.typ) else String
  | Element(_, el) as element ->
    if expr el != Int then
    raise (Failure ("Invalid element access in " ^ string_of_expr element)
       )
    else Int
  | Noexpr -> Ast.None
in

let check_variable_assign (t, n, ex) =
  let rt = expr ex in
    ignore(check_assign (type_of_identifier n) (expr ex)
            (Failure ("Illegal assignment " ^ string_of_typ t ^ " := " ^
               string_of_typ rt ^
              " in \'" ^ string_of_typ t ^ " " ^ n ^ " = " ^
                 string_of_expr ex ^ "\'"))); ()
in

let check_bool_expr e = if expr e != Bool
  then raise (Failure ("Expected Boolean expression in " ^ string_of_expr
     e))
  else ()
in

(* Verify a statement or throw an exception *)
let rec stmt = function
    Block sl -> let rec check_block = function
       [Return _ as s] -> stmt s
     | Return _ :: _ -> raise (Failure "nothing may follow a return")
     | Block sl :: ss -> check_block (sl @ ss)
     | s :: ss -> stmt s ; check_block ss
     | [] -> ()
      in check_block sl
  | Expr e -> ignore (expr e)
  | Return e -> let t = (match expr e with (* Since there is a call to
                                               another function, we don't care
                                               about the length *)
                        Ast.Tuple(ty, _) -> Ast.Tuple(ty, 0)
                      |_ -> expr e) in if t = func.typ then () else
      raise (Failure (func.fname ^ " returns " ^ string_of_typ t ^ " but "
          ^
```

```
                        string_of_typ func.typ ^ " was expected."))
      | If(p, b1, b2, b3) -> check_bool_expr p; stmt b1; stmt b2; stmt b3
      | Elif(p, b1) -> check_bool_expr p; stmt b1
      | For(e1, e2, e3, st) ->  ignore (expr e1); check_bool_expr e2;
                                ignore (expr e3); stmt st
      | ForIn(_, e2, st) -> (* Handle nonetype identifier e1 *)
                if (match expr e2 with
                      Tuple(_, _) -> false
                    | _ -> true) && expr e2 != String
                then raise (Failure ("Expected a tuple or a str in " ^
                   string_of_expr e2));
                stmt st
      | While(p, s) -> check_bool_expr p; stmt s
      | HiddenWhile(e, s1, s2) -> ignore(expr e); stmt s1; stmt s2
      | In(e1, e2) -> if (match expr e2 with
                      Tuple(_, _) -> false
                    | _ -> true) then
              raise (Failure ("Expected a tuple in " ^ string_of_expr e1 ^ " "
                  ^ string_of_expr e2))
              else ()
      | Break -> ignore (0)
      | Continue -> ignore (0)
      | Nostmt -> ignore (0)
      | Declaration(_) -> ignore (0)
    in
      ignore (List.map check_variable_assign func.fbody.f_vdecls);
      stmt (Block func.fbody.f_stmts); ()
  in

  (****    Check classes    ****)
  let check_class c =

    List.iter (check_not_none (fun n -> "Illegal NoneType class local " ^ n ^
      " in " ^ c.cname)) c.cbody.vdecls;

    report_duplicate (fun n -> "Duplicate class local " ^ n ^ " in " ^ c.cname
       )
      (List.map (fun (_, n, _) -> n) c.cbody.vdecls);

    List.iter check_function c.cbody.funcs;
in
  List.iter check_class classes;
  List.iter check_function functions;
  report_duplicate (fun n -> "Duplicate class " ^ n) (List.map (fun cd -> cd.
     cname) classes)
```

### 8.0.6 codegen.ml

```
        module L = Llvm
module Fcmp = Llvm.Fcmp
module A = Ast
module StringMap = Map.Make(String)

let translate ((_, globals), (functions, classes)) =
```

```
let context = L.global_context () in
let the_module = L.create_module context "Scolkam"
    and i32_t  = L.i32_type context
    and i8_t   = L.i8_type context
    and i1_t   = L.i1_type context
    and flt_t  = L.double_type context
    and str_t  = L.pointer_type (L.i8_type context)
    and void_t = L.void_type context in

(* Declare printf(), which the print built-in function will call *)
let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in
let prints_t = L.var_arg_function_type str_t [| L.pointer_type i8_t |] in
let prints_func = L.declare_function "puts" prints_t the_module in

let int_format_str    b = L.build_global_stringptr "%d\n" "fmt" b
and float_format_str  b = L.build_global_stringptr "%f\n" "fmt" b
and string_format_str b = L.build_global_stringptr "%s\n" "fmt" b in

(* For the implementation of 'break' and 'continue' *)
let (after_block) = ref (L.block_of_value (L.const_int i32_t 0))
and (before_block) = ref (L.block_of_value (L.const_int i32_t 0)) in

(* Global variables *)
let global_vars = ref (StringMap.empty) in

(* Current function and local variables *)
let (local_vars) = ref StringMap.empty in
let currentf = ref (List.hd functions) in

(* Return the value or the type for a variable or formal argument *)
(* All the tables have the structure (type, llvalue) *)
let name_to_llval n : L.llvalue =
  try (snd (StringMap.find n !local_vars))
  with Not_found -> (snd (StringMap.find n !global_vars))
in

let name_to_type n : A.typ =
  try (fst (StringMap.find n !local_vars))
  with Not_found -> (fst (StringMap.find n !global_vars)) in

(* LLVM types *)
let rec ltype_of_typ = function
    A.Int         -> i32_t
  | A.Float       -> flt_t
  | A.Bool        -> i1_t
  | A.String      -> str_t
  | A.None        -> void_t
  | A.Tuple(t, _) -> L.pointer_type (ltype_of_typ t)

and gen_type = function
    A.IntLit _          -> A.Int
  | A.FloatLit _        -> A.Float
```

```
  | A.BoolLit _         -> A.Bool
  | A.StringLit _       -> A.String
  | A.TupleLit x        -> gen_type (List.hd x)
  | A.Element (e,_)     -> gen_type (A.Id(e))
  | A.Id s              -> (match (name_to_type s) with
                              A.Tuple(t,_) -> t
                            |  _ as ty -> ty)
  | A.Call(s,_)       -> let fdecl =
                           List.find (fun x -> x.A.fname = s) functions in
                           (match fdecl.A.typ with
                             A.Tuple(t,_) -> t
                           |  _ as ty -> ty)
  | A.Binop(e1, _, _)  -> gen_type e1
  | A.Unop(_, e1)     -> gen_type e1
  | A.Assign(s, _)    -> gen_type (A.Id(s))
  | A.Noexpr          -> raise (Failure "corrupted tree - Noexpr as a
     statement")

and lreturn_type ty = match ty with
    A.Tuple (t, _) -> L.pointer_type (ltype_of_typ t)
  | _ -> ltype_of_typ ty

and find_type (t, _) = ltype_of_typ t

and format_str x_type builder =
    let b = builder in
      match x_type with
        A.Int    -> int_format_str b
      | A.Float  -> float_format_str b
      | A.String -> string_format_str b
      | _ -> raise (Failure "Invalid printf type")

in

(* Define each function (arguments and return type) so we can call it *)
let function_decls =
  let function_decl m fdecl =
    let name         = fdecl.A.fname
    and formal_types = Array.of_list (List.map find_type fdecl.A.formals)
  in
    let ftype = L.function_type (lreturn_type fdecl.A.typ) formal_types in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m

  in List.fold_left function_decl StringMap.empty functions
in

  (* Invoke "f builder" if the current block doesn't already
     have a terminal (e.g., a branch). *)
let rec add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (f builder)

(* Return the code of an expression to be built in LLVM *)
```

```
(* This code is partially inspired by the ideas behind the
   memory management in 2015's Dice*)
and expr builder =  let b = builder in
    function
    A.IntLit i -> L.const_int i32_t i
  | A.FloatLit f -> L.const_float flt_t f
  | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | A.StringLit sl -> L.build_global_stringptr sl "string" b
  | A.TupleLit elts ->
      let sizeva = (List.length elts) + 1 in
      let size   = L.const_int i32_t sizeva in
      let ty     = ltype_of_typ
                     (A.Tuple(gen_type(List.hd elts), sizeva)) in
      let arr    = L.build_array_malloc ty size "init1" b in
      let arr    = L.build_pointercast arr ty "init2" b in
      let _      = L.build_bitcast size ty "init3" b in
      let values = List.map (expr b) elts in
      let buildf i v =
        (let arr_ptr =
          L.build_gep arr [| (L.const_int i32_t (i+1)) |] "init4" b in
          ignore(L.build_store v arr_ptr b);)
      in
      List.iteri buildf values; arr
  | A.Element (s, e) ->
      let idx = expr b e in
      let idx = L.build_add idx (L.const_int i32_t 1) "access1" b in
      let arr = expr b (A.Id(s)) in
      let res = L.build_gep arr [| idx |] "access2" b in
      L.build_load res "access3" b
  | A.Noexpr -> L.const_int i32_t 0
  | A.Id s -> let llval = (name_to_llval s) in
                L.build_load llval s b
  | A.Binop (e1, op, e2) ->
      let e1' = expr b e1
      and e2' = expr b e2
      and float_ops = (match op with
          A.Add     -> L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mult    -> L.build_fmul
        | A.Div     -> L.build_fdiv
        | A.Mod     -> L.build_frem
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq     -> L.build_fcmp L.Fcmp.One
        | A.Less    -> L.build_fcmp L.Fcmp.Olt
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Greater -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
      )
      and int_ops = match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
```

```
      | A.Div     -> L.build_sdiv
      | A.Mod     -> L.build_urem
      | A.And     -> L.build_and
      | A.Or      -> L.build_or
      | A.Equal   -> L.build_icmp L.Icmp.Eq
      | A.Neq     -> L.build_icmp L.Icmp.Ne
      | A.Less    -> L.build_icmp L.Icmp.Slt
      | A.Leq     -> L.build_icmp L.Icmp.Sle
      | A.Greater -> L.build_icmp L.Icmp.Sgt
      | A.Geq     -> L.build_icmp L.Icmp.Sge
    and str_ops = match op with
      | A.Add     -> expr b (A.StringLit((A.string_of_expr e1) ^ (A.
          string_of_expr e2)))
      | _ -> (L.const_int i32_t 0)
    in
    if (L.type_of e1' = flt_t || L.type_of e2' = flt_t) then float_ops e1'
        e2' "tmp" builder
    else if ((L.type_of e1' = str_t) && (L.type_of e2' = str_t)) then
       str_ops else int_ops e1' e2' "tmp" builder
| A.Unop(op, e) ->
    let e' = expr b e in
    (match op with
      A.Neg     -> if (L.type_of e' = flt_t) then L.build_fneg else L.
          build_neg
    | A.Not     -> L.build_not) e' "tmp" b
| A.Assign (s, e) -> let e' = expr b e in
                 ignore (L.build_store e' (name_to_llval s) b); e'
| A.Call ("print", [e]) | A.Call ("printb", [e]) ->
  let e' = expr b e in
  let e_type = gen_type e in
  L.build_call printf_func [| ( format_str e_type b) ; e' |]
    "printf" b
| A.Call ("prints", [e]) ->
    L.build_call prints_func [| (expr b e) |]
    "puts" b
| A.Call ("toint", [e]) ->
    let e' = expr b e in
    let e_type = L.string_of_lltype (L.type_of e') in
      if e_type = L.string_of_lltype i32_t
        then raise (Failure "You converted int to int")
      else if e_type = L.string_of_lltype flt_t
        then L.build_fptosi e' i32_t "cast" b
      (* else if e_type = L.string_of_lltype i1_t
        then L.build_bitcast e' i32_t "castbool" b *)
      else raise (Failure ("You cannot convert from this type to int"))
| A.Call ("tofloat", [e]) ->
    let e' = expr b e in
    let e_type = L.string_of_lltype (L.type_of e') in
      if e_type = L.string_of_lltype flt_t
        then raise (Failure "You converted float to float")
      else if e_type = L.string_of_lltype i32_t
        then L.build_sitofp e' flt_t "castint" b
      (* else if e_type = L.string_of_lltype i1_t
        then L.build_bitcast e' flt_t "castbool" b *)
```

```
        else raise (Failure ("You cannot convert from this type to int"))
  | A.Call (f, act) ->
      let (fdef, f_decl) = StringMap.find f function_decls in
      let actuals = List.rev (List.map (expr b) (List.rev act)) in
      let result = (match f_decl.A.typ with A.None -> ""
                                          | _ -> f ^ "_result") in
    L.build_call fdef (Array.of_list actuals) result b

(* Build the code for the given statement; return the b for
   the statement's successor *)
and stmt builder = let b = builder in
    let (the_function, _) = StringMap.find !currentf.A.fname function_decls
       in
    function
    A.Block sl -> List.fold_left stmt b sl
  | A.Expr e ->  ignore (expr b e); b
  | A.Return e ->  ignore (match !currentf.A.typ with
      A.None -> L.build_ret_void b
    | _ -> L.build_ret (expr b e) b); b
  | A.If (predicate, then_stmts, else_if_stmts, else_stmts) ->

    (* Removing the elseifs by recursively replacing the else_statements *)
    let rec remove_elif (_, _, else_if_stmts, else_stmts) =
    (match else_if_stmts with
       A.Block(hd::tl) ->
         let new_predicate, new_then =
           (match hd with
             A.Elif(condition,stmts) -> condition, stmts
           | _ -> raise (Failure "Corrupted tree - Elseif problem")) in
         let new_else_ifs = A.Block(tl) in
         let new_else = remove_elif (new_predicate, new_then, new_else_ifs,
            else_stmts) in
         A.If(new_predicate, new_then, new_else_ifs, new_else)
       | A.Block([]) -> else_stmts
       | _ -> else_stmts) in
    let new_else_stmts = remove_elif (predicate, then_stmts, else_if_stmts,
       else_stmts) in

    let bool_val = expr b predicate in
    let merge_bb = L.append_block context "merge" the_function in

    (* Emit 'then' value. *)
    let then_bb = L.append_block context "then" the_function in
    let then_code = (stmt (L.builder_at_end context then_bb) then_stmts) in
    add_terminal then_code (L.build_br merge_bb);

    (* Emit 'else' value. *)
    let else_bb = L.append_block context "else" the_function in
    let else_code = (stmt (L.builder_at_end context else_bb) new_else_stmts)
        in
    add_terminal else_code (L.build_br merge_bb);

    (* Add the conditional branch. *)
    ignore (L.build_cond_br bool_val then_bb else_bb b);
```

```
    L.builder_at_end context merge_bb
  | A.While (predicate, body) -> stmt b (A.HiddenWhile(predicate, body, A.
     Block([A.Nostmt])))
  | A.HiddenWhile (predicate, body, increment) ->
      let pred_bb = L.append_block context "while" the_function in
      ignore (L.build_br pred_bb b);

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

      let pred_b = L.builder_at_end context pred_bb in
      let bool_val = expr pred_b predicate in

      let increment_bb = L.append_block context "increment" the_function in
      let increment_b = L.builder_at_end context increment_bb in
      let merge_bb = L.append_block context "merge" the_function in

      ignore(before_block := increment_bb);
      ignore(after_block  := merge_bb);
      add_terminal (stmt (L.builder_at_end context body_bb) body)
        (L.build_br increment_bb);
      add_terminal (stmt increment_b increment)
        (L.build_br pred_bb);
      ignore (L.build_cond_br bool_val body_bb merge_bb pred_b);
      L.builder_at_end context merge_bb
  | A.For (e1, e2, e3, body) -> stmt b
      ( A.Block [A.Expr e1 ; A.HiddenWhile (e2, A.Block [body], A.Expr e3) ]
          )
  | A.Nostmt -> ignore (0); b
  | A.Continue ->
    let block = fun () -> !before_block in
    ignore (L.build_br (block ()) b); b
  | A.Break ->
    let block = fun () -> !after_block in
    ignore (L.build_br (block ()) b); b
  | A.Declaration _ -> raise (Failure "Corrupted Tree")
  | A.Elif (_,_) | A.ForIn (_,_,_) | A.In (_) ->
      raise (Failure "Corrupted Tree")

and global_var m (t, n, e) =
  let (f,_) = StringMap.find "main" function_decls in
  let builder = L.builder_at_end context (L.entry_block f) in
  (* Build the first initialization of the variables *)
  let rec init t e = match e with
        A.IntLit _ | A.FloatLit _ | A.BoolLit _ | A.StringLit _ -> expr
           builder e
      | A.TupleLit(_) ->
          let ty     = ltype_of_typ t in
          L.const_ptrtoint (L.const_int i32_t 0) ty
      | _ ->
           match t with
           A.Int   -> expr builder (A.IntLit(0))
         | A.Float -> expr builder (A.FloatLit(0.0))
         | A.String -> expr builder (A.StringLit(""))
         | A.Bool   -> expr builder (A.BoolLit(true))
```

```
          | A.Tuple(_, _) -> (init t (A.TupleLit([])))
          | A.None -> expr builder (A.Noexpr)
  in
  let tuple = (t, (L.define_global n (init t e) the_module)) in
  StringMap.add n tuple m

(* Fill in the body of the given function *)
and build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.A.fname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  currentf := fdecl;

  (* Construct the function's "locals": formal arguments and locally
     declared variables.  Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  let add_formal m (t, n) p =
    L.set_value_name n p;
    let local = L.build_alloca (find_type (t, n)) n builder in
      ignore (L.build_store p local builder);
    StringMap.add n (t, local) m
  in

  let add_local m (t, n) =
    let local_var = L.build_alloca (find_type (t, n)) n builder in
    StringMap.add n (t, local_var) m
  in

  let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.formals
                (Array.to_list (L.params the_function)) in

  local_vars := List.fold_left add_local formals
                (List.map (fun (t, n, _)-> (t, n)) fdecl.A.fbody.A.f_vdecls)
                   ;

  let assign_variable (_, n, e) =
    let e' = expr builder e in ignore (L.build_store e' (name_to_llval n)
       builder); e'
  in
  (* Build the code for each statement in the function *)
  let builder = ignore (List.map assign_variable fdecl.A.fbody.A.f_vdecls);
    stmt builder (A.Block fdecl.A.fbody.A.f_stmts);
  in


  (* Add a return if the last block falls off the end *)
  add_terminal builder (match fdecl.A.typ with
      A.None -> L.build_ret_void
    | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))

in

let build_class_body _ =
```

```
    ()
  in

  (* Declare each global variable; remember its value in a map *)
  let globals = List.rev globals in
  List.iter (fun k -> global_vars := global_var !global_vars k) globals;
  List.iter build_class_body classes;
  List.iter build_function_body functions;
  the_module
```

### 8.0.7 testall.sh

```
#!/bin/sh

# Regression testing script for scolkam
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli-3.7"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the scolkam compiler.  Usually "./scolkam.native"
# Try "_build/scolkam" if ocamlbuild was unable to create a symbolic link.
SCOLKAM="./scolkam.native -c"
#SCOLKAM="_build/scolkam.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.sco files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
```

```
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalError "failed: $* did not report an error"
    return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.sco//'`
    reffile=`echo $1 | sed 's/.sco$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$SCOLKAM" $1 "|" "$LLI" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
```

```
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.sco//'`
    reffile=`echo $1 | sed 's/.sco$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$SCOLKAM" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`
```

```
LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall
     .sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.sco tests/fail-*.sco"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```

### 8.0.8 Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : scolkam.native

scolkam.native :
        ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
                scolkam.native

# "make clean" removes all generated files

.PHONY : clean
clean :
        ocamlbuild -clean
```

```
        rm -rf testall.log *.diff scolkam scanner.ml parser.ml parser.mli
        rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.ll
        rm -rf *.err *.out

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx scolkam.cmx

scolkam : $(OBJS)
        ocamlfind ocamlopt -linkpkg -package core -package llvm -package llvm.
           analysis $(OBJS) -o scolkam

scanner.ml : scanner.mll
        ocamllex scanner.mll

parser.ml parser.mli : parser.mly
        ocamlyacc parser.mly

%.cmo : %.ml
        ocamlc -c $<

%.cmi : %.mli
        ocamlc -c $<

%.cmx : %.ml
        ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
scolkam.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
scolkam.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball
TESTS = add arith arith_2 arith_3 arith_4 arith_5 augment augment2 augment3
   augment4 avl_tree bst calc fibonacci float_add float_div    \
    float_multi float_sub for funct funct1 funct2 global hello_world if
       linked_list     \
    misc_test modulus print print-string tuple_access tuple_slice unary \

FAILS = x_funct x_funct print \

TESTFILES = $(TESTS:%=test-%.sco) $(TESTS:%=test-%.out) \
        $(FAILS:%=fail-%.sco) $(FAILS:%=fail-%.err)
```

```
TARFILES = ast.ml codegen.ml Makefile scolkam.ml parser.mly README scanner.mll
    \
    semant.ml testall.sh $(TESTFILES:%=tests/%)
```

# References


[1] http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html *The GNU C Reference Manual.*. N.p., n.d. Web. 26 Oct. 2015.

[2] Edwards, Stephen. *"COMS W4115 Programming Language and Translators."* Lectures.

[3] https://docs.python.org/3/reference/index.html "The Python Language Reference" *The Python Language Reference.*

[4] https://courses.cs.washington.edu/courses/cse140/13wi/eval_rules.pdf "Python Evaluation Rules" *"Python Evaluation Rules"*