ICSI 680 Finale Report - Salamander

William Dahl
Dept. of Computer Science
University at Albany - SUNY
Albany, NY, USA
wdahl@albany.edu

***Abstract*—Salamander is a bytecode compiled, high-level general purpose programming language. Salamander draws inspiration from python, aiming to replicate its code readability and sue of logical line and indention to divide up program blocks and statements. Salamander is statically typed and garbage collected, with built in support for dynamic typing. It supports multiple programming paradigms, including procedural and object oriented programming. Salamander comes with a standard library that enables the use of standard input and output, file handling, socket programming, and enables the use of threading.**

I. Introduction

Running Salamander source code consists of the code going through two major phases, Compilation and Code execution. The salamander compiler taxes in the source code and produces bytecode that is then executed with the Salamander Virtual Machine. The Compilation process of salamander source code consist of the following steps: 1. Lexical Analysis 2. Syntax Analysis 3. Semantic Analysis 4. Code Generation 5. Code optimization 6. Source file importing The Salamanders compiler performs the first 4 steps in one pass, acting as a traditional one-pass compiler. However the compiler makes multiple other passes over the generated code for the sake of performing code optimizations.

The Virtual Machine interprets the byte code generated by the salamander compiler and executes the operations one at a time. The Salamander Virtual Machine is a stack based machine, utilizing a program stack to perform the operations during code execution. The virtual machine also proved automatic memory management via garbage collection, performed using the mark-and-sweep garbage collection algorithm. The virtual machine also enables the use of threading through built-in thread data types and threading utility functions. Threads can also access global data safely thought he use of locks which is also offered by salamander through built in lock data types and lock utility functions.

Salamanders standard built-in library allows users to perform I/O operations on either standard in and out as well as on files by offering a built in file data-type and file handling utility functions. It also enables users to perform socket programming by offer a built-in socket data type along with utility functions that would allow for a user to construct a basic server client app through socket programming. The standard library also include built-in functions for type conversions and more.

II. Compiler

A. Lexical Analysis

The first phase of compilation is Lexical Analysis. Lexical Analysis is the process of braking down an inputted source code into individual tokens or lexems, which are then fed as input to the parser to handle syntax analysis[**1**][**2**][**3**][**4**][**5**]. The tokens are created by identifying the pattern that a specific lexem matches. After the pattern in identified the respective token then then generated by the lexical analyzer and given to the parser. the lexical analyzer reads in a character stream from the source file given and begins matching patterns that the lexem may match until finally generating the resulting token. The generated tokens are categorized as keywords, identifiers, constants, and symbols.

Salamanders Lexer takes in the user provided source code in the form of a string as input. It then reads through each character int he provided source code keeping track of the current location in the file using a counter. The Lexer also keeps tracks of physical lines in the file for error reporting purposes. The Lexer maintains a indention stack to keep track of changing indention levels through out the source code. It contains a flag to indicate the start of a new logical line as well as a count for the number of groupings the pointer to the current character is within, as well as a counter for when a formatted string began to remember at which grouping level that a formatted string will begin again, along with a flag that indicates weather the Lexer is handling a formatted string. The Lexer produce Lexems that consumed as input by the parser. A Lexem contains the Token type, the value of the Lexem stored as a string and the physical line that the lexem can be found on in the source code.

A Salamander program is first read by a lexical analyzer, or Lexer. The Lexer generates a stream of tokens that is then passed as input to the syntax analyzer, or Parser. Salamanders syntax is inspired by pythons syntax and as such, a salamander program is divided up into a sequence of ”logical lines”. The end of a logical line is represented by a NEWLINE token, which marks the end of a statement. A logical line can be represented by one or more ”physical lines” as both implicate and explicit ”line joining” is done. A Physical line is an actual line in the source file, the end of which is represented by the ’\n’ character. A comment is denoted by a ’#’ and ends at the end of the physical it is on. Comments are ignored by the Lexer and treated like white space and are therefore not tokenized and passed to the parser. Explicate line joining is done by using a (\) character at the end of a physical line. This tells the Lexer to skip the generation of a NEWLINE token thus making the next line part of the same logical line. For example:

```
if 1900 < year and year < 2100 \
    and 1 <= month and month <= 12 \
    and 1 <= day and day <= 31 \
    and 0 <= hour and hour < 24 \
    and 0 <= minute and minute < 60:
        print("is valid data")
```

Implicate line joining is done on expression within parenthesis, brackets, or braces without the need for a backslash. For example:

```
list months = ['January', February',
               'March','April',
               'May', 'June',
               'July', 'August',
               'September','October',
               'November', 'December']
```

No NEWLINE token is generated by the Lexer when the end of a physical line is detected while in between either parenthesis, brackets, or braces and so all expression within them are implicitly joined together into one logical line. Blank lines, i.e. a line that only contains white space characters are ignored by the lexer. No NEWLINE tokens are generated to represent the blank line.

Indention is used in salamander to determine statement grouping and program scoping. The indention level is determined by the number of leading white spaces, either tabs or spaces, at the beginning a new logical line. The Lexer will generate either an INDENT or a DEDENT token to represent changes in the indention level and therefore program scoping and grouping. When the number of leading white spaces increases as compared to the last logical line a INDENT token is generated, and when the number of white spaces goes down, a DEDENT token is generated. This is handled by keeping a stack of the indention levels used in the program. When ever the indention level goes up, the new level is pushed onto the stack and an INDENT token is generated. When the indention level is lower than the level currently on top of the stack the value is popped off and a DEDENT token is generated. This is repeated until the current indention level matches the level on top of the stack. The indention stack is initialized with a 0 which will never be popped off as the indention level cannot be lower than 0. This results in the numbers being pushed onto the stack being strictly increasing from bottom to top. At the end of the file, A DEDENT token is also generated for each number left on the stack that is greater than 0.

Except for that the beginning of logical lines and in strings, the Lexer will not generate nay tokens for white spaces. Instead white space is used by the Lexer to delimit lexems. The Lexer will take the longest possible lexem that matches a valid token for token generation. The lexer first starts by looking for possible keywords and identifiers, or names. It does this by taking in a character from the character stream of the source code file and checking if it is either letter or an underscore. From there the lexer begins to build a lexem for every new character read in from the character stream that is either a letter, number, or underscore. Once a character is read that does not match this conditions the currently constructed lexem is first compared to Salamanders set of keywords. If the lexem does not match any of Salamanders keywords then an IDENTIFIER token is generated. Salamanders keywords include:

and	class	else	if	None
or	return	super	while	False
for	def	self	True	del
elif	global	import	in	is
not	pass			

The Lexer generates a respective unique token type for each of these key words when the current lexem matches. For example, if the current lexem is ”and” a AND token is generated, if it is ”class” then a CLASS token is generated, etc. However, in the cases of the keywords ”is” and ”not” some special handling is required. If a lexem matching ”is” is identified then the next possible lexem must be constructed to see of it matches ”not” in order to see if a”is not” token should be generated. If the next lexem in the source file matches ”not” then a ”is not” token is generated. The same applies to the keyword ”not” to see if the ”not in” token should be generated instead. Salamander supports static typing and as such, salamanders builtin types also constitute keywords. Salamanders builtin data-types include:

int	float	str	bool	list
tuple	set	dict	file	thread
lock	socket	Object		

When the first character of the lexem is a digit, the Lexer reads in characters from the character stream and continues to build the current lexem until it reads in a non digit character. From there the Lexer checks if the character is a decimal point ('.'). if it is then the lexer adds it to the current lexem and continues reading characters until another non digit character is read in. Once finished, the Lexer generates a NUMBER token for the parser containing the constructed lexem. When the character being read by the Lexer is not alpha-numeric, it checks to see if the lexem matches any special characters, including delimiters, operators, and strings. Salamanders delimiter characters include:

(	)	{	}	[	]
;	,	.	:		

When ever a '(', '', or '[' is found by the Lexer, a group counter is incremented. This counter keeps track of how many groupings there currently are for implicate line joining. Like wise, when ever a ')', '', or ']' is found the group counter is decremented. Salamanders operation character's include:

-	+	/	//	*	**
!	!=	=	==	<	<=
>	>=	<<	>>	%	&
|	ˆ	∼			

Each of salamanders special characters generate s their own respective Token when found by the Lexer.

Strings are identified by lexems starting with the a double quote (”) or a single quote (’). When a quote is encountered by the Lexer its type, either double or single, is saved in the Lexer and the Lexer reads characters in from the source constructing a string lexem until the respective quote is encountered again by the Lexer. When the string closes, the Lexer generates a STRING token. String in Salamander can denote new lines and table by using ’\n’ and ’\t’ respectively within their strings. if the foll owning character int he string is not a ’n’ or a ’t’ then the backslash is treated as a normal backslash within the string. Implicate line joining is also used within strings, so until the closing quote is found by the lexer, no NEWLINE tokens will be generated, allowing string to span multiple lines in salamander. Salamander also supports inline joined string formatting, similar to python. This is identified by the lexer by finding a lexem that matches the pattern of an ’f’ followed by a double quote (”). This tells that the following string is a formatted string and to generate a ”formatted string” token. When a formatted string is found a flag in the Lexer is set which tells the lexer that a formatted string is still being tokens. within a formatted string, the lexer will break up the string if curly braces are found. From there until the closing curly brace is found, the lexer tokenizes as usual. However once the closing curly brace is found by the lexer the lexer will treat the next lexem as a formatted string until either a closing ”” is found, or another curly brace is encountered, generating another ”formatted string” token. If the constructed Lexem does not match any patters, the Lexer returns an ERROR token containing a message describing the error, which is then sent to and handled by the parser.

B. Syntax Analysis

The second step in the compilation process for salamander is Syntax Analysis[**7**][**8**][**9**][**10**][**11**]. In this step the Parser makes sure that the matched lexems follow the syntax rules for salamander. The parser will step through each match lexem checking to make sure what was given was a valid salamander program. The Parser gets a lexem from the Lexer and then begins constructing a parse tree for the current statement or expression. If the syntax rules for salamander are not followed then an error will be generated. If the rules are followed the result is a abstract syntax tree that can then be passed onto the Semantic Analysis phase.

The salamander compiler handles compiling source code to byte code in a single pass, performing syntax analysis, Semantic analysis, and code generation in a single pass, side-by-side. The Syntax of Salamander is broadly divided up into two different groups, statements and expressions. Statements include defining variables, classes, and functions, as well as flow control such as if statements and loops. Statements are ended by a NEWLINE token. The following table displays the statement type and an example of what it would like in salamander code.

Statement Type	Example
import statement	import ’./importFile’
class definition	class Foo(Bar)
function definition	def add(int x, int y)
variable definition	int x = 2
global variable definition	global int x = 2
for loop	for x in range(10):
if statement	if x ¡ 10:
return statement	return x
while loop	while x ¡ 10:
del element	del list1[4]
pass statement	pass

Expressions in salamander include function calls, unary operations, binary operations, logical connective like ’and’ and ’or’, as well as subscripts of a lists, etc. The following table displays an exhaustive description of what is considered an expression in Salamander:

Expression Type	Example
parentheses	(1+2*3)
tuple	(1,2,3)
function call	func()
set	{1,2,3}
dict	{1:”one”, 2:”two”}
class attribute	Foo.bar()
Unary operation	-1
Binary Operation	1+2
identifier reference	x = 1
string	”hello”
number	2
and	x and y
literal	False
if expression	a if y else c
or	x or y
super call	super()
self reference	self.i = 0
list	[1,2,3]
subscript	list1[4]
formatted string	f”1+1={1+1}”

Each expression in Salamander has an order of precedence. The following table shows the order of precedence for operations in Salamander from the highest precedence to the lowest precedence[**12**]:

Precedence of operations is modeled using an enum type with the lowest precedence having an enum value of 0 and each enum having a higher level of precedence than the last.

Parsing expressions in salamander is done using Pratt parsing [**6**][**13**][**14**][**15**][**16**][**17**]. Pratt parsing is a type of parsing that was created by Vaughan Pratt in a 1973 paper. It is also called “top-down operator-precedence parsing” because it is a top-down, recursive algorithm which can easily handle operator precedence’s. To take advantage of Pratt parsing

Operator	Description
(), [], , {key:value}	parenthesized expression, list display, dictionary display, set display
x[], x(), x.a	Subscription, slicing, call, attribute reference
*	Exponentiation
-x, ∼x	negative, bitwise NOT
, /, //, %	Multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	shifts
&	bit wise and
ˆ	bit wise xor
|	bit wise or
in, not in, is, is not, <, <=, >, >=, ! =, ==	Comparisons, including membership tests and identity tests
not x	logical not
and	logical and
or	logical or
if-else	conditional expression

Salamander has a table of Parsers, each parser being dedicated to a specific lexem type that is generated by the lexer. Each parser has a function that will handle the infix case and the prefix case for the respective lexem. For example, When a 'minus' lexem appears in the prefix position we will parse a unary operation, and when it is infix position we will parse a binary operation. Each Parser also contains the precedence of the infix operation, so the precedence of the parser is the addition and subtraction precedence. For some tokens there is no case where it would be used in a prefix expression, or vise versa in an infix expression. In those cases the respective function is set to NULL and if a NULL function is retrieved from the parser that indicates a syntax error and an error is thrown. Parsing an expression starts with getting the next Lexem from the Lexer. The parser for the lexem is then retrieved from the table. The prefix parsing function for the parser is then executed. After the prefix function is executed the next parser for the next operator is retrieved from he table. If the precedence of the next parser is less than or equal too the current parser then the infix function is executed for the next parser.

The prefix parsing rule for a left-paren ('(') parses expressions until a right-paren (')') is returned by the Lexer. If a right-paren is not the next Lexem returned by the Lexer an error is thrown. When a left-paren occurs in the infix position then that is a function call. Function calls parse the expressions in between the left and right parens assuming them to be separated by commas. If a right-paren is not returned by the Lexer after parsing the arguments then an error is thrown.

The prefix parsing rule for a left-brace ('{') handles parsing a set by parsing the expressions within the braces separated by commas. If the next Lexem after parsing the internal expressions is not a right-brace ('}') then an error is thrown. The left-brace lexem does not have a infix parse function.

The dot ('.') lexem does not have a prefix parsing function. Its infix parsing function handles parsing class attributes. If the next lexem is an equals token ('=') then the next expression is parsed for staring the expressions value int he attribute. If the next Lexem is a left-paren then it is a method call and the args to the method are parsed using the same method in function calls.

Unary operations are prefix functions. Unary operations are parsed by first getting the operator type and then parsing the expression that the operator is being applied to. The unary parsing function is used on "not", "minus" ('-'), and "bit not" ('∼').

Binary operations are infix functions. A binary operation is parsed by first getting the operator type and the parser for the operator type. The right side of the binary operation is then parsed. The binary parsing function is the infix parsing function for the following lexems:

!=	==	>	>=	<	<=
is	is not	in	not in	+	-
	/	//	%	**	<<
>>	|	ˆ	&		

Identifiers have an prefix parsing rule. Identifiers are parsed by first funding the identifier (covered more in Semantic Analysis section). Then, if the next lexem is an equals then the next expression is parsed. Identifiers do not have an infix parsing function. Both strings and numbers also only have a prefix parsing function that simply generates the code for the constants (covered more in code generation section). The "and" and "or" lexems only have an infix parsing function and simply move on to parsing the right side of their expression and generating the respective jumps. (Cover more in code generation). The "False", "True", and "None" lexems only have prefix parsing functions. They are parsed as literals and as a result their parsing functions only generate their byte code.

The left-bracket ('[') lexem has both a prefix and infix parsing functions. When the left-bracket is in the prefix position we are parsing a list. Lists are parsed by parsing each expression within the brackets, separated by commas. If the the next lexem after the expressions are parsed is not a right-bracket (']') then an error is thrown. If the left-bracket is in the infix position then we have a subscript. Subscripts are parsed by parsing the first expression between the brackets. After, if the next lexem is a colon (':') then the next expression is parsed to get the index ranges for a slice. After parsing either one or two indexes a right-bracket is checked for by the Lexer. If the next lexem is not a right-bracket, then an error is thrown.

The "if" lexem has a infix parsing rule that parses a if-else conditional expression. The if-else expression is parsed by first parsing the condition and then checking for the next lexem to be ”else”. If it is not an else then an error is thrown. If it is an else then else expression is parsed. The ”if” lexem does not have a prefix parsing function.

The ”super” lexem has a prefix parsing function that parses a reference to the classes super class. It starts getting parsed by first reading in a left-paren (’(’) and right-paren (’)’) and then a dot followed by the attributes identifier. If these lexems are not read in this order an error is thrown. A ”self” lexem is then created and parsed using the identifier parsing function. If the next lexem is a left-paren then it is a method call and the arguments are parsed using the same parsing method for function calls. A new ”super” lexem is then created and parsed as an identifier. the ”super” lexem does not have a infix parsing function

Formatted strings have a prefix parsing function. Formatted strings are parsed by parsing each consecutive expression while the next lesxem is either a left-brace ’{’ or is another formatted string. If the next lexem if a left-brace then the expression within it is parsed and a right-brace (’}’) is expected after. If a right-brace is not the next lexem then an error is thrown. Formatted strings do not have an infix parsing function.

Statements are parsed in a linear manner as opposed to a recursive manner like expressions. This is because a statement doe not have to worry about order of operations and will be clearly terminated by a NEWLINE lexem. An ”import” statement is parsed by first reading in a string that contains the salamander source code file to import. It then loads the source code to be compiled and parses it (more on this in the linking section).

variable definitions are parsed by first reading in a data type for the variable and then reading int the variables identifier. If the next lexem is an equals then the next expression is parsed. If a ”global” lexem is encountered then it denotes the variable will be a global one. The data type of the variable is then read and then the variables identifier. like non global variables, if the next lexem is an equals then the next expression is parsed for assignment. A del statement is parsed when a ”del” lexem is encountered. A del statement deletes the specified entry in a list or dictionary A del statement is parsed by first reading in an identifier and parsing it. Next, a left-bracket (’[’) is expected. The expression within the the bracket is then parsed. A colon lexem is then checked for and if found the next expression is parsed. After either one or two expressions are parsed a right-bracket is expected followed by a NEWLINE to mark the end of the statement.

Function definitions are identified by starting with a ”def” lexem. Then a return type is checked for followed by the function identifier. Next, the function args and body is parsed. This starts with reading in a left-paren and then parsing the arguments, separated by a comma. The arguments are expected to consist of a data-type and an identifier. If a right-paren is not followed after the last argument then an error is thrown. Next a colon followed by a NEWLINE and INDENT lexem is read in. From here the following statements and expression are parsed as part of the function body until the DEDENT lexem is encountered.

A class definition is parsed by first reading in an identifier which will be the class name. If the next Lexem is a left-paren then the class has a base class. The identifier for the base class is then read in and parsed. After, a right-paren is expected. after the class identifier (and possible base class identifier) a colon (’:’) is expected followed by a NEWLINE and a INDENT lexem. It then parses the body until either DEDENT lexem is encountered or the end of the file is reached. In the class body, if a data-type lexem is found then a filed is to be parsed. A class field is parsed by first reading in the fields identifier. If the next lexem is a equals then the next expression is parsed for storing. If the ”def” lexem is found int he class body then that is a method. A method is parsed by first checking for a return type and then reading in an identifier which will be its name. The method is then parsed in the same fashion as a function.

A for loop is parsed when a ”for” lexem is encountered. It starts getting parsed by first reading in the identifier for the loop larget, next it expects an ”in” lexem followed by an expression. After the loop header a colon followed by a NEWLINE and an INDENT Lexem is expected. The following statements and expressions are then parsed as part of the for loops body until a DEDENT Lexem is encountered. An if statement is parsed when a ”if” lexem is encountered. The if statement is parsed by first parsing the test condition expression. The parser then expects to read in a colon followed by a NEWLINE and an INDENT lexem. The indented block is then parsed as part of the if statements body. The parser then checks for a ”elif” lexem. If one is found then it is replaces with a ”if” lexem and the resulting if statement is then parsed recursively. If the next lexem is an ”else” lexem then a colon followed by a NEWLINE and an INDENT lexem is expected. The resulting block is then parsed as the else statements body until a respective DEDENT lexem is found. While statements a re parsed similarly. First parsing the conditional expression and then reading in a colon, NEWLINE, and DEDENT and parsing the resulting block as part of the loops body.

a return statement is parsed when a ”return” lexem is found. If the the next lexem after the return lexem is a NEWLINE then a simple return is generated. Other wise the expression to be returned is parsed, followed by reading in a NEWLINE to mark the end of the statement. A pass statement is parsed whenever a ”pass” lexem is found. A pass statement is simply a no-op instruction and thus the next Lexem is simply read in by the parser.

C. Semantic Analysis

The next step for the compilation of salamander code is Semantic analysis. The semantic analysis phase is where the salamander compiler will be handle task such as variable scoping, and type checking [**18**][**19**][**20**][**21**][**22**]. When performing semantic analysis the salamander compiler will be able to catch errors is the provided source code such as type mismatch, undeclared variables, already declared variables, function argument type mismatch, and handling accessing variables outside of the scope. After the source code passes semantic analysis it is ready to have byte code generated.

To perform semantic analysis information about variables needed to be stored, such as data type, scope, and argument types in the case of a function. That information is stored in a Name node for each variable. To organize the semantic information within a function or a class separate passers are created. A Function Parser contains semantic information relevant to the current function that it is parsing, including the variables declared within the function, the functions return type, and any non-local variables referenced within the function and the scope of the function. The return type of the function will be a string representing the type being returned by the function after execution. The non-local variables will be pointers to variables defined in a parent function that are referenced in the function body. Because Salamander is a multi-paradigm programming language that supports scripting, the script its self is treated as a function and thus a Function Parser is used to parse the outside script as well. When a new function is defined a new Function parser is created and used to parse and house the semantic information for that function. Function Parsers are added to a stack when they are created, and when the function is done being parsed, the function parser is popped off the stack. So, the first Function Parser to be pushed onto the stack is the Function Parser for any scripting code and will consequently enclose all of the other Function Parsers and will be the last function parser to be popped off of the stack at the end of the parsing. Classes also have their own type of Class parser which keeps track of the classes name, whether the class has a base class or not, and any fields declared within the class as variables.

A Function Parser can have different types, including a script (The first Function Parser and only first function parser will be of type script), a function, a method, or an initializer. If the function type is a method or an intializer then this means the Function Parser is enclosed within a class and ”self” is defined as referring to the enclosing class and added to the variables in the Function Parser. After the Function Parser is created it is pushed onto the stack of Function Parsers. When a variable is being defined it is added to the variables in its enclosing Function Parser. If the variable already exists in the Function Parser at the current scope, then an error is thrown. Otherwise the variable is stored into the list of variables of the enclosing Function Parser. When being stored a new Variable Node is created which contains the variables identifier and its scope. The scope of the variable is initially set to -1 in order to mark it as uninitialized. When the variable does get initialized, the scope is set to the current scope of the Function Parser. Nonlocal variables are retrieved by the function parser by recursively moving though the enclosing function parsers below it in the Function Parser stack. Once the variable is found it is stored as a non-local variable in each Function Parser above it containing an index to the variable reference in the above Function Parsers. A variable is retrieved by looping backwards over the list of variables in the Function Parser and comparing each variables identifier with the one being searched for. If the variable is found but its scope is still -1 then the variable has not been initialized yet and cannot be used thus resulting in an error being thrown.

A Class parser handles its loading and storing of fields the same way as a function parser handles variables. However classes have some additional semantic analysis tasks. First, when a new class is created its name is added to a set of class types. This allows for semantic analysis to be done in regards to type checking for class instances. Since a class Parser is itself enclosed in a function Parser, the class identifier is added to the list of variables in the enclosing function parser. When a new Class Parser is created it is also added pushed on to a stack of other Class Parsers and then popped off after parsing is completed. A mapping is already created that map a class name to its Class parser, this is used to check for fields and methods defined on the class. If a class has a base class then an identifier for the ”super” keyword is created ad added to the variables of the enclosing function with a data-type of the base class. A base class mapping is also created which maps each class to its base, which is used to handle polymorphism and inheritance in Semantic analysis. Any fields and methods defined in the class body are added to the list of variables in the class parser. Methods are also functions themselves, so when a method is defined in a class and new function parser is created to handle the semantic analysis of the method. When a function or a method is parsed the argument types are stored in a list int he order they appear. This is used to check the argument types for when the method or function is called.

global variables are not added to the list of variables maintained by the enclosing function parser. Instead, global variables are added to a global variable map that can be accessed at any point in the program. This map maps the global variable identifiers to their data type. When ever a global variable is referenced, this mapping is used to look up the type. When an identifier is being parsed we fist check if it is a variable defined within the current enclosing function parser. If it is not then we check if it is among other enclosing function parsers further down the stack. If we still cannot find the identifier, we then check the global map. If the identifier is not contained within the global map, the variable is undefined and an error is thrown.

Data types are checked by first seeing if the current lexem is one of the built in data-types, if it is not then the identifier is looked for in the list of classes that have been defined in the program. One of the built in data types is called ”Object” this is a generic data type that allows for dynamic typing to be used and results in type checking being done at run time. Floats and integers can have operations performed on each other and the resulting data type will be a float. Floats can be set to an integer type, but integer types cannot be set to a float.

Each expression parsing function as described in the Syntax Analysis section returns a string that represents the data type for the expression. This returned value is used to check if expressions, statements, and variable assignments are type correct. For example, the parsing function for the ”and” lexem returns a ”bool” data type because the result from an and expression will be a boolean. In binary operations type checking is done to insure that the operands are compatible types for the operator. For example, when the operation being done is ”in” or ”not in” the 2nd operand should be either a list, set, tuple, or dictionary. Other wise the types of the operands should either be compatible (i.e. float and int or string and string). Further type checking is done depending on the operation being performed, for example, ”!=” and ”==” can be applied to types float, int, str, list, tuple, set and dict but ” $>$ ”, ” $>=$ ”, ” $<$ ”, and ” $<=$ ” can only be applied to types str, int, and float. Operators ”is” and ”is not” can be applied to any operators as they compare the references themselves. Comparison operators will result in a ”bool” data type being returned for the expression. The following table shows what data types can be used for each binary arithmetic operations:

Operator	Compatible Data Types
+	str, int, float, list, tuple, set
-	int, float
	int, float
/	int, float
//	int, float
%	int
*	int, float
<<	int
>>	int
|	int
^	int
&	int

Note, that if either operand type was an object then type checking would have pushed off until runtime. If one of the operands is a ”Object” type then the return type if the function is an ”Object” type. If one of the operands is of type ”float” then the return type of the expression with be a float. Other wise the operands must have been the same data type and the return type will be whatever the operands type was. This type checking logic is applied across all instance of type comparisons throughout the compiler.

When a call is being parsed, the Semantic analysis done on the call is the check if the function being called is an initializer or not. If the identifier can be found among the variables of the enclosing function then it is either a function or a method. If it cannot be found then it checked to see if the it is a class initializer. If it the function identifier is a class that has been defined then the function identifier is changed to ”__init__”, other wise the function being called is a globally defined, built-in function. The argument data types are checked by retrieving the argument type list for the function and comparing them to the types of the arguments given in the order they appear. If the types do not match and error is thrown. Note, Object argument types can take in any argument type, but a Object argument cannot go in to any argument type unless it is a Object type as well. If the function is an initalizer then the ”__init__” function is retrieved from the class that is being intialized using the class map that maps a class name to its class parser. The arguments are then type checked in the same way as a method or function is. If the function is globally defined function then the argument types are dynamic and checked at runtime.

Semantic Analysis on a attribute is done by first getting the instance of the class from the function parser and then getting the class type. From there the class parser is retrieved using the class map and then the attribute is retrieved from the variables in the class parser. If the attribute is not in the class compiler and the class has a base class then the base class is searched. This is repeated until the last base class is found. If the attribute cannot be found then the type checking a pushed off to be done at runtime and the returned expression type is ”Object”. if it can be found then the data type of the attribute is what is returned by the expression.

In the prefix parsing function for the left-paren, semantic analysis is applied by deducing if the expression data type is a ”tuple” or the type of the expression within the parenthesis. If a comma is found between the parenthesis, then a ”tuple” data type is returned, otherwise the type of the expression within the parenthesis is returned. parsing a list expression with return a ”list” data type, and parsing a set expression with return a ”dict” data type if key value pairs are found in between the braces, and a ”set” data-type if not. When parsing a subscript, if the subscript is getting a slice, then the returned data type from the expression with be a ”list”, other wise it will be an ”Object”. This is because of the dynamic nature of the built in data structures in salamander, that each element of the data structure is treated as an Object and will be dynamically typed.

Parsing a number will return a datatype of either ”int” if the number is an integer, and return ”float” otherwise. ”Or” and ”and” expression will both return ”bool” types and both string and formatted string expressions will return a ”str” type. parsing an identifier will return the data type of the identifier. Built in global functions are defined at the beginning of the compilation process. They are loaded into the global map with their return values being what they map to. allowing for semantic analysis and type checking to be done on the built in function.

D. Code Generation

The next phase in the compilation process is code generation. The salamander compiler compiles a given source code file into byte code that is then executed by a Virtual Machine byte code interpreter. Byte code is an instruction set that is generated by a compiler and then executed by another software program instead of by the computer its self [23][24][25][26][27]. the generated byte code usually consists of numeric codes, constants, and references to values which represents the results of lexical, syntax, and semantic analysis done prior by the compiler.

Salamanders byte code consists of a list of integers representing the operations to perform and their arguments. It also contains a list of references to parsed expression results generated by the compiler. Salamander byte code also keeps track of lines for error reporting. And Expression result has a value type which can either be a boolean, a None type, a number, a string, an object pointer, or an error. The Expression type denotes the type of data stored in the Expression node. An expression can reference an object such as a function definition, a class definition, a file pointer, a thread, etc. (will be explained more in depth in code execution section). Each operation code is represented as a number using a enum, thus giving each op code a unique value.

Each Function parser has a reference to a function object. The function object contains information relevant to the function that is being parsed by the Function parser. This includes the number of arguments for the function, the number of non-local variable references, the generated byte code for the function, the name of the function, and the return type of the function. As such, the byte code that is generated while parsing a function is then stored within the function object, representing what operations need to be executed when the function is called. So as byte code is generated it is written specifically to the function pointed to by the function parser at the top of the function parser stack. Therefore any scripting code that is generated will be written to the function pointed to by the initial scripting function parser and each new function will have the code generated by the respective function parser written directly the functions own byte code, allowing the code to be loaded and executed where ever the function is called.

Most of the byte code generated will either take no arguments or only 1 argument, However there are a few byte code operations that are more complex. Loops and jumps have to handle calculating the offsets in the code so they know how far to jump ahead or back in the code. To generate a loop operation the offset is calculated by subtracting the generated byte code size after parsing the loop body by the size before parsing the loop body. Jumps are generated by first generating a jump operation followed by a free space which will contain the offset. The offset is then calculated after the code between the jump branches has been parsed and generated. The offset is calculated by subtracting the size of the generated byte code after the Branch from the size of the byte code before the branch. The calculated offset is then stored in the free space that was saved right after the jump op code. Generating a return has a special case when the return is coming from a class initalizer. When a class intializer is returning the return value should be the name of the class its self. A constant is generated by writing the constant op code to the byte code, storing a reference to the constant in the byte codes list of variables and storing the index of it as a parameter to the constant op code.

Each expression parsing function will generate byte code to carry out the operation of the expression it parsed. The and expression parsing function generates a ”jump if false” operation followed by a ”pop” operation. After parsing the right side of the ”and” expression the jump offset is written to the byte code. Parsing an ”or” expression will generate both a ”jump if false op” and a standard ”jump op”. The ”jump if false” offset is ten generated, followed by a ”pop op” and then the jump offset is generated.

The following table shows the generated byte code for the following binary operations:

Operator	Generated byte code
!=	equal op and not op
==	equal op
>	greater than op
>=	less than op and not op
<	less than op
<=	greater then op and not op
is	is op
is not	is op and not op
in	in op
not in	in op and not op
+	add op
-	sub op
	mult op
/	division op
//	floor divined
%	mod op
*	pow op
<<	left shit op
>>	Right shift
|	bit or op
ˆ	bit xor op
&	bit and op

Parsing a call generates a ”call op” and the number of arguments passed to the call as its parameter. Parsing an attribute starts with writing the attribute reference to the list of expression value, getting the index that it store into in return. If the attribute is followed by an equals then a ”store attribute op” is generated followed by the index of the attribute in the list of expressions as its one parameter. If the next lexem is a left paren then a ”method call op” is generated followed by two parameters, the attribute index in the list of constants, and the argument count. Or it generates the loading ”load attribute op” followed by the attributes identifier index int the byte code references.

Parsing a literal will generate the ”false op” if the lexem is ”False”, the ”none op” if the lexem is ”None” and the ”true op” if the lexem is ”True”, none of which take in any arguments. The if expression parsing function will generate the ”if expression op” which takes in no arguments. Parsing a paren will either generate a ”tuple op” followed byt he size of the tuple if the paren contains expression separated by a comma or wont generate any code besides what is generated by parsing the expression in between the paren. Parsing a list will generate the ”list op” followed by the list size as its perimeter. Parsing a left-brace with either generate a ”dict op” if key:value pairs are found in-between the braces and will generate a ”set op” if not. Both operators are followed by their size as their parameter. In the even of a dictionary, the size parameter will be multiplied by 2 to account for having both a key and value for one entry. Parsing a subscript will generate either a ”storing op” or a ”loading op”. The ”storing op” will be generated when the next lexem after the subscript is an ”equals”, a ”loading op” will be generated otherwise. If a colon is found in the subscript then a ”slice op” will be generated, other wise a ”subscript op” will be generated. The possible generate code out comes when parsing a subscript are ”loading subscript op”, ”storing subscript op”, ”loading slice op”, and ”storing slice op”. None of these operations take in any parameters.

parsing a number or a string will generate a new constant which has its reference stored in the byte code. Parsing a formatted string will generate a ”add op” to concatenate the string and a formatted expression, otherwise a constant is generated and then a ”add op” is generated in order to concatenate the two formatted string together. When parsing an identifier, if the identifier is local to the current function parser, then either a ”load name op” or ”store name op” will be generated. If the identifier is a non local then either a ”load nonlocal op” or a ”store nonlocal op” will be generated. If the variable is not within a enclosing function parser then either a ”load global op” or ”store global op” will be generated. The respective store ops will be generated if the the next lexem is an ”equals”, other wise the respective load op will be generated, followed by the index of the location of the variable respective to where it was found as its parameter.

Parsing a super will call will generate a ”super call op” if the next lexem is a left-paren along with two parameters, the super classes index in the byte code constant and the number of arguments passed to the function. Otherwise a ”load super op” will be generated with the super classes index reference as its only parameter. Parsing the Unary operations will generate a ”not op” if the operation is ”not”, ”invert op” if the operation is a minus (’-’), and a ”bit not op” if the operation is the ”bit wise not” ($' \sim'$).

Parsing a function will generate a ”function def op”, write the function to the expression value list of the byte code and store the index as the parameter to the operation. It will also take in additional parameters for each nonlocal variable reference made within the function. Parsing a class field will write the fields identifier to the byte code constants and then generate a ”field def op” with the index to the Fields identifier as its parameter. Similarly parsing a method will write the method identifier to the byte code constants and then generate a ”method def op” with the index of the method identifier as its parameter. parsing the actual class definition will start with writing the class name to the list of byte code contestant reference and the generating a ”class def op” with the index to its identifier as its parameter. If the class has a base class then a ”base op” is generated. After the class body is done being parsed a ”pop op” is generated to remove the extra class definition.

Defining a global variable will write the global identifier to the byte code list of constants and then generate a ”assign global op” with the index of the identifier as its parameter. Generating for loop code starts by generating a ”store name op” with the index of the iterable being iterated over as its parameter. Next it generates a ”load iter start op” which takes in a index of a iterable as a parameter and gets the beginning of the iterator. This is followed by storing the beginning of the iterator into using the ”store name op” and taking in the index of the variable that will hold the current position of the iterator as a parameter. Next we load the current position of the iterator and then the end position of the iterator. We then check if the are not equal by generating the ”equal op” and ”not op” code. We then generate a jump branch to take in the event the the condition is false, followed by a ”pop op” to get rid of the results from the condition. Next the jump for if the condition is true is generated. In this branch we generate byte code to increment the iterator by generating the ”increment iterator op”. after the the loop is generated and the jump offset is stored into the byte code. We are not entering the body of the for loop but before we can start parsing the body we need to generate code to set the target to be the current value pointed to by the iterator. We do this by generating a ”load iterator op” code with the iterator index as its parameter, and generating a ”store name op” with the targets index in order to store the iterators current value into the target of the for loop.

Generating if statement byte code is done by generating a ”jump if false op” followed by a ”pop op” to clear the results from the comparison. after the body is parsed a jump is generated for the else branch and the if branches jump offset is generated and stored as the parameter for the first jump code that was generated. after parsing the else body, the jump offset for the else branch is generated and stored into the byte code. Generating code for a while statement is done similarly, by generating a jump to take if the condition is false, then parsing the while body and then generating the loop to go back to the beginning of the loop and then generating the jump offset and storing it into the byte code for when the condition is false. Generating code for a del statement will result in either a ”del slice op” or a ”del subscript op” being generated depending on if the expression being deleted is a slice of a list or a subscript of a list/dictionary. Neither operation takes in a parameter.

Once the end of a Function is reached then return byte code is generated and the function parser is popped off of the function parser stack.

E. Optimization

Compiler optimization is done in order to optimize the execution speed of the generated code [**28**][**29**][**30**][**31**][**32**]. Optimizations are typically implemented through several optimization transformations done in sequence. The goal of a compiler optimizer is to take in a code as input and output new code that does the same thing as the input code but with fewer instructions. Compiler optimization types include peephole optimizations which are optimizations done through a ”peephole”, i.e. they consider only a small, local portion or block of the code and are usually applied directly to the generated code.

One optimization done by the salamander compiler is constant folding [**33**]. Constant folding is when the compiler recognizes a that an expression is being performed on only constants and evaluates the expression at compile time instead of run time, thus reducing the total number of operations needed to be done. Salamander does this when parsing binary operations. The salamander compiler looks back to the last two op codes generated by the compiler. If both the op codes are for handling constants, then the constants are retrieved from the constant reference list in the byte code using the indexes that are parameters into the ”constant op”. The generated code instructions for creating the constants operands are then removed from the generated code and the binary operation is then applied to the constants directly. the binary operation code from three instructions to just one, which is to retrieve the constant value after computing the expression.

Another optimization done by the salamander compiler is strength reduction [**34**]. Strength reductions when the compiler replaces expensive operation with equivalent but less expensive ones. For example, replacing the addition of the save value with each other with a single left-shit operation, this also applies to multiplication of a value by 2. The floor division of a value by 2 can also be done with a single right-shift. Salamander performs this optimization on binary operations that satisfy the above conditions. First the salamander compiler checks to make sure that the operand being shifted is an integer to insure the accuracy of the computation is accurate. In the case of a variable being added to its self, the salamander compiler checks to make sure that the last two op codes are the same and are referencing the same variable. From there the second operand is dropped and replaced code to generate a constant 1. Then instead of a ”add op” being generated a ”left-shift op” is generated. In the case of multiplication, the salamander compiler checks if one of the operands is a constant with a value of 2. If this is the case then the operation to get a constant 2 is replaced with an operation with a constant 1 and instead of generating a ”mult op” a ”left-shit op” is generated instead. In the case of floor division, the salamander compiler checks to see if the denominator is a constant with the value 2. If it is then it is replaced with a constant of a value 1 and a ”right-shift op” is generated in-place of a ”floor division op”.

The third peephole optimization that is performed by the salamander compiler is copy propagation [**35**]. copy propagation is done to reduce any redundant storing and loading of variables. For example in the following code:

```
int y = x
int z = 3 * y
```

The storing of the value x into y is unnecessary and the above code can be reduced to just:

```
int z = 3 * x
```

The salamander compiler achieves this by making two passes over the completely generated code of a function parser. When a function is done being parsed its generated code is passed through the copy propagation optimizer. The copy propagation optimizer loops through generated code and checks for cases where a variable is just set to another variable and creates a mapping from the new variable to the one it was set to. When ever a variable is loaded, the compiler checks to see if the variable is in the propagation map. if it is, then the variable to be loaded is replaced with the variable being mapped to in the propagation map. If either a variable being mapped or one being mapped two appears in another store operation then they are removed from the propagation map, as the values for them have changed and they may not necessarily have the same value any more. Copy propagation is able to be done in one pass over the generated byte code.

The next function wide optimization is the deletion of dead code. This optimization is ran after the copy propagation optimization because, after propagating variables there may be unused variables that can be deleted. This is ac hived by making two separate passes over the functions generated byte code. The compiler starts by creating a mapping from each variable to the last operation done on it and the index of that operation in the byte code. When a store operation is found, the optimizer checks if the last operation on the variable was also a store operation. If it was then that last instruction is marked for deletion. When a load operation is encountered then the last operation fort he variable is updated to reflect that the variable was in fact loaded and used since the last time it was set. After the byte code is iterated through the optimizer has a list of instructions that can be deleted. This deletion is done by looping back over the byte code and creating a new sequence of instructions without the dead code and saving that as the byte code for the function.

Salamanders compiler also preforms a function optimization called ”tail call recursion” [**36**]. A tail call optimization is a optimization done by a compiler that optimizes the code in functions in cases where the last operation of a function is another function call. The optimization works by jumping right into the function code and not generating a new function call. A specific type if tail call optimization is tail call recursion, which is a special case of a tail call, when the final operation of the function is a recursive call to itself. Instead of generating a new function call the salamander compiler turns the recursive call into a loop. and simply jumps back to the top of the function. The salamander compiler handles this optimization by checking if the last operation in the in a function is a call to itself. If it is then the call operation is removed from the byte code along with all of the arguments passed into the function. This is done by maintaining a pointer to the last function call in the byte code. All of the generated byte code until pointer is removed and stored into a separate list to be added back into the byte code later. After the loading op to get the function identifier is removed from the byte code, all of the byte code for the arguments are then added back to the byte code, followed by the generation of storing operations to update the values of the arguments in the function. Finally a loop is generated that will push move the instruction pointer back to the top of the function for execution again, thus converting a tail recursive call into a loop.

The final optimization that salamander uses is its hash maps. The salamander compiler uses flat hash maps to perform all of its mapping and variable loop ups in order to get an as optimal compilation time as possible and not be slowed down [**37**]. The hash map that is used by the salamander compiler is a open source c++ library that was created by Malte Skarupke and can be found at [**38**]. This hash map uses robin hood hashing with a upper limit on the number of probes that will be done before the hash maps size is grown. The hash map uses Open addressing, Linear probing, Robing hood hashing, Prime number amount of slots, and has an upper limit on the probe count. Open addressing means that the underlying storage of the hash table is a continuous array. Linear probing means that if the location the new item wants to be inserted at is already full then its simply goes to the next location over, this continues until a free space is found. Robin hood hashing means that when a item is inserted into the hash table, it tries to keep each element as close to its desired location as possible. This involves moving elements around after insertions to insure each element is as close to their desired location as possible. If the a new element that is inserted will end up farther away from its ideal location than the other element is, they are swapped. The hash map also maintains a prime number size so that to find the insertion point you simply use the modulo operator to assign the hash value of an element to a location. This Hash map get consistent O(1) look up time and much better overall performance than c++'s standard unordered map library.

F. Linking

Linking in salamander is done by using the import statement at the beginning of a a source file to be executed. When an import statement is encountered the Salamander compiler, the compiler expects the next lexem to be a string containing the path to the source code file to be imported. Salamander achieves this by keeping a stack of lexers that will each contain the lexem for a respective source code file. When a import statement is found, then file being imported is open and its contents are read into a string. This string contains the imported files source code is then given as input to a new Lexer which is then pushed onto the Lexer stack. When a new Lexem is requested by the salamander compiler, the Lexem is retrieved from the Lexer that is on top of the Lexer stack. Each statement in the imported source file is then parsed until the end of the file is reached. After the import file is done being parsed, the Lexer is popped of the Lexer stack. Within imported files, only identifier definitions are parsed at the script level. This is so that any expressions in the imported file at the script level are not executed. Only import statements, class definitions, function definitions, and variable definitions are parsed at the script level in an imported source file. If the Lexer stack size is greater than 1 and the current function parser is at the script level all other statements and expression are skipped by the compiler and code is not generated for them. However any statements and expressions within function definitions are parsed. The references between the imported files and the main file are maintained because the script level function parser for the main, file and the imported file are the same, so any definitions at the script level in an imported file are stored in the main files function parser. The generated code from the imported source file is essentially pre-pended to the generated code of the main file.

III. Virtual Machine

A. Byte Code Execution

The byte code generated by the salamander compiler is executed within a Virtual Machine. The Virtual Machine understands the byte code instruction set and executes the appropriate operations. Salamander Virtual machine is specifically a virtual stack machine, maintaining a program stack with all the variables, identifiers, and pointers pushed onto it as execution is done, instruction by instruction [**25**][**6**][**39**][**40**][**41**]. The byte code instructions are passed to the virtual machine by the salamander compiler and then executed one at a time.

The Salamander Virtual Machine is a stack based machine. It has a stack that contains all of the program references. It also maintains a stack of function calls to keep track of the current function that is being executed by virtual machine. A function call contains a reference to the function definition a counter to keep track of the instruction that is currently be executed within the function definition, and a start variable which holds the index of the start of the function call on the virtual machines stack. The virtual Machine also keeps a stack of non-local variables referenced by the byte code in the function. This acts as a way to find the location of the non-local variable on the stack so that it can be used in the function asking for it. A non-local variable pointer in the Virtual machine contains the variables scope and whether the reference is still valid. The virtual machine also maintains a set of objects that have been pushed onto the stack as well as a garbage collector stack to be used for garbage collection.

The Virtual machine starts by getting the byte code generated by the salamander compiler as a reference to the script function. if the function returned is NULL then a compilation error must have happened an a compiler error is returned. Otherwise the the script function is wrapped in a function definition, the stack is initialized with the script function definition as its first element and ten the function definition is called by the compiler. A function definition contains a reference to the function actual function and a list of non-local variables that are referenced by the function. When a function is called a new function call object is created and added to the list of function calls in the Virtual Machine. It start index is set to the current size of the stack minus its number of arguments, minus another 1 to cover the function name itself. After the function call is generated for the script function the Virtual Machine starts executing the byte code. When executing byte code the virtual machine starts by getting the function call on top of the function call stack because this is the most recent function call and thus its byte code should be what is getting executed.

A constant operation has one parameter which is the index of the constant expression in the byte code expression list. This expression is retrieved from the byte code and pushed into the stack. None operations pushes a None expression onto the

the stack. Likewise the True and False operations push their respective expressions onto the stack. A Pop operation simply pops the expression on top of the stack off.

The load and store name operations both take one parameter which is the index of the variable on the stack. The load name operation retrieves this value and pushes it onto the top of the stack, and the store operation stores whatever is at the top of the stack into the index specified in the stack. The operator to get the start of an iterator has one parameter which is the location of the iterable object on the stack. A respective iterator pointing to the beginning of the iterable object is then created and pushed onto the stack. Likewise, loading the end of the iterator is done in the same fashion but instead a iterator pointing to the end of the iterable is pushed into the stack. Incrementing an iterator operation is done by getting the iterator to be increment using the index parameter and then simply incrementing the iterator pointer, thus moving the pointer to the next object in the iterable. Loading an iterator is done similarly but instead of being incremented, the iterator reference is pushed onto the stack. Global identifiers are stored into a global map that maps the identifier to the value stored in it. Assigning a global operation is carried out by finding the constant referenced by the index parameter and then in the byte code constant list and adding it to the global identifier map with its value being whatever is on top of the stack. Storing a Global identifier is done similarly but instead the variable is checked to already exist in the global map, if it does not then a variable undefined error is thrown. A global identifier is loaded by finding the identifier int he map and then pushing its value onto the stack. Non local variables are loaded by pushing the expression stored in the current function definitions non local list at the given index to the stack. Similarly storing a nonlocal variable is done by storing the expression at the top of the stack into the current function definition nonlocal list at the index given to the operation. Attributes are loaded onto the stack by getting the class instance on top of the stack and finding the attribute using the operators index parameter in the byte code constants. An instance object contains a reference to the class definition it is an instance of and maintains its own set of dynamically allocated fields. The attribute is then looked for in the instances set of dynamic fields. If the attribute cannot be found then it looks for the static fields in the class type. the value stored in the field is then pushed onto the stack. If the attribute still cannot be found then the Virtual Machine looks for through the classes methods and pushing the method onto the stack if found. If the method cannot be found then a run-time error is thrown. A method object contains a reference to the class definition it is in and a reference to is function definition. Storing an attribute operation is done by updating the instances fields with the attribute and setting its value to the expression on top of the stack. Loading a super call is done by getting the base class which is on top of the stack and then funding the method being called within the base class. If the method cannot be found then a run-time error is thrown.

Binary operations are executed by pooping the last two values off of the stack and executing the respective operation on them and pushing the result back onto the stack. Equality, greater than, and less than operations are executed by popping the top two expression from the stack off and comparing them with respective operation and then pushing the result onto the stack. An ”is” operation is done by directly comparing the references of the top two expression on the stack and pushing the result onto the stack. The ”in” operation expects the first element on top of the stack to be a iterable abject, and the second to be an object to search for in the iterable object. These values are popped off of the stack and the respective iterator is searched through. for the value being looked for. An add operation can be applied to numbers, string, lists, tuples, and sets. When applied to numbers the numbers are simply added together and the result is pushed onto the stack. When applied to any other data type the operands are concatenated together and the result is pushed onto the stack. The remaining binary operations can only be doe on numbers are executed by popping the last two expression off of the stack and pushing the result after the operation onto the stack.

The unary operations ”not”, ”bit wise not” and ”invert” are executed by popping the top expression off of the stack, applying the respective operation to it and then pushing the result onto the stack. The not operation expects a boolean value and will negate the boolean value it received and push the result onto the stack. bit wise and invert both expect numbers and will push the result back onto the stack.

A Jump operation is executed by adding the jump offset to the current function calls counter, skipping over the byte code between the counter and the offset. Likewise a jump if false operation is done in the same way but the function call counter is only changed if the value on top of the stack is false. Otherwise the virtual machine juts moves onto the next instruction. A loop operation functions juts like a jump operation but in the opposite direction, subtracting the loop offset from the current function call counter. An if-else expression is executed by popping the top three expression off of the stack. The 2nd expression if the if condition so if it is true, the third expression from the stack is pushed back onto the stack, otherwise the first expression popped off of the stack is pushed back on.

For a call operation the number represent the number of arguments expected. A Call operation could be to a method, a class initalizer, a user defined function, or a built-in function. If the call is for a method then the class the method is for is pushed onto the stack and a new function call is generated for the method. If the it is a class initalizer, then a new instance of the class in created and pushed onto the stack and new function call is generated using the classes initialize function if it is defined, otherwise no function call is generated. If the call is for a user defined function then a new function call is generated and pushed onto the stack. If the call is for a built-in function then the arguments are stored into a vector of expressions which is then sent to the builtin function to be executed on. A built-in function in salamander takes in a vector of expression representing the arguments to the function and returns an expression. The result from the function is then pushed onto the stack and no new function call is needed. If the expression returned by the built-in function is an error type then a error is thrown.

List, tuples, sets, and dictionary operations all have one parameter which refers to the number of elements in them. Each operation is executed by popping the number of expressions equal to the size of the structure off of the stack and then storing them into the respective data structure object. Lists and tuples contain a vector holding the expressions, the difference being that tuples are immutable and cannot be changed after being declared. A set store the expression in a set of expressions, and a dictionary stores the expressions into a map of expressions to expressions. A load slice operation is done by popping the top three elements off of the stack. The first element is the ending index, the second is the starting index, and the third is the list or tuple being sliced. The slice is then computed and pushed onto the stack. Storing a slice is done by popping the top four expression off of the stack. The first expression is the expression being stored into the slice. The reaming three expression are the end, start, and list being sliced. Storing a slice can only be done on a list and expects the expression being stored to be a list its self. A slice can also be deleted from a list. The slice of the list is computed using the top three expression popped off of the stack as done before, but the elements in the list between the starting and end index's are removed from the list. The first slice index is expected to be smaller than the ending slice index. If the ending index is larger than the size of the object then the up until the end of the object is returned. If the starting index is larger than the size of the object being sliced then a index out of bounds error is thrown. A load subscript operation is executed similarly to loading a slice. The top two expression are popped of the stack with the first being the subscript and the second being the item to subscript. A subscript can be done on lists, tuples, and dictionaries. If the item being subscripted is a list or a tuple then it is expected that the subscript is an integer and is smaller than the size of the list or tuple. If the subscript is a negative number then the subscript is gotten starting from the back of the the list, however if the negative number is larger than size of the object an index out of bounds error is thrown. If the subscript is for a dictionary then the subscript can be any expression. If the expression cannot be found in the set of keys for the dictionary then a key not found error is thrown. Storing a subscript is done by popping the top three expression off of the stack, the first being the expression to store, the second being the subscript and the third being the object being subscripted. Storing a subscript can only be done on dictionaries and lists because tuples are immutable. If the subscript of the list is equal to the size of the list then the expression is appended to the end of the list. If the subscript is larger than the size of the list then a index out of bounds error is thrown. Otherwise, the value in the list at the index is overwritten with the new expression. If the a dictionary is being subscripted, the value for the key is overwritten is the key exists, if the key does not exists in the dictionary then it is added and the expression is stored as its value. A subscript can also be deleted from a list and a dictionary. Deleting a subscript is executed by popping the subscript and the object off of the stack and removing the expression stored at the index or key. This will reduce the size of the list or dictionary by one.

Method call operations have two parameters, the argument count, and the methods identifier. These values are retrieved and the class instance is gotten from the stack. The method identifier is then looked for in the instances fields. If the method is within the instances field then a new function call is generated, if it did not find the method then it looks for the method within the base classes fields if the the class has a base class. If the attribute still cannot be found then a error is thrown. IF the the method is found then a new function call is generated, pushed onto the stack, and the current function call is updated to be the one on top of the call stack. A super call operation has two parameters like a method call. The base class referenced by the super class is popped off of the stack and the method being called from is looked for in the base classes fields. If the method is found a new function call is generated pushed on to the stack and the current function call is updated to be the new function call on top of the call stack.

A class definition operation is executed by getting the name of the class and crating a new class definition object that is then pushed on to the virtual machines stack. A class definition has a name, a mapping containing the methods of the class mapped to by the method identifier stored as a string and a mapping containing the fields of the class that maps the fields identifier to its value. A base class operation stars by getting the base class expression off of the stack. The sub class is then popped off of the top of the stock and all of the methods and fields from the base class are copied over to the method and field maps in the subclass. A field definition operation has one parameter which is the index of the fields identifier in the byte code constants list. The fields value is then retrieved from the top of the stack and the class that the field is apart of is also retrieved from the stack. The class definition then has the filed identifier added to its fields map and the value is set to the expression on top of the stack. A method definition operation is executed similarly by getting the methods identifier from the byte code constants and then adding the method identifier to the classes methods map with the value being the expression that is on top of the stack.

A function definition operation is executed by getting the function stored at the index referenced by the byte code instruction. The function is then wrapped in a function definition and is pushed onto the stack. A return operation is executed by popping the top expression on the stack off, this is the returned result from the function, and popping the top function call on top of the call stack off. If after popping the function call off, the call stack size is 0 then we've reached the end of the byte code and execution is done. Otherwise, everything on stack left from the previous function call is popped off the stack and then the first expression popped off at the beginning is pushed back into the stack and the current function call is set to the new top of the function call stack.

B. Garbage Collection

Garbage collection is a type of automatic memory management performed by the virtual machine. The garbage collector frees up memory that contains a reference to an object but is no longer begin used. This allows the virtual machine to reuse memory locations though out the its execution of the byte code. The idea of garbage collection was created by John McCarthy in 1959 to simply memory management in Lisp [**42**][**43**][**6**].

A specific type of garbage collection algorithm is the Mark-and-Sweep garbage collection algorithm [**44**][**45**][**46**]. This algorithm has two phases, the mark phase and the sweep phase, which will detect all of the no longer reachable objects in memory and then making the free the memory space for the program to use. Salamanders Virtual Machine uses this garbage collection algorithm in order to perform memory management. Each object that is created in salamander has a marked flag that is initially set to false and is added to a set containing all the of objects created during execution of the program. The first step of the algorithm is to mark each object that is directly accessible by the user by marking each object that is currently on the execution stack, each function call in the call stack and each globally defined object in the global variable pool, this is referred to as marking the roots of the program. When a object is marked it is added to the garbage collection stack. If the object has already been marked then it is not pushed onto the garbage collector stack to stop any reference cycles.

After all of the roots are marked the next step is to begin tracing through each object that is reachable through one of the roots. This is done by popping the object at the top of the garbage collector off the garbage collection stack and then marking all object references the object has. If the object is a method object, then the class definition and method function references are marked and added to the garbage collection stack to have their own object references marked. If the object is a class definition then the classes methods and fields are marked by iterating over the method and fields map and marking each object reference. For a function definition, the function reference is marked. A function object will iterate over each of the expressions in the functions byte code constants list. A class instance will mark its referenced class definition that it is an instance of and the dynamic fields of the instance. For lists, tuples, dictionaries, and sets, the Objects stored with in them are iterated over and marked. If the object is a thread then the referenced function and the tuple contain the arguments to the function are marked. Files, locks, sockets, iterators, and built in functions do not need to be traced through because they do not contain any references to objects.

The last step in garbage collection is the sweep phase. The sweep phase involves clearing the memory of any unreachable objects. This is done by iterating over the set of all the created objects and seeing if the object was marked. If it was marked then it is simply unmarked for the next run through of the garbage collector. If the object is not marked, that means that the object is unreachable from the current objects directly accessible by the user and can be freed from memory. The object is then removed from the set of objects and the object is cleared. Clearing an object involves clearing any vectors, maps and other data structures being used by the object and then deleting the reference to it. When the virtual machine finishes execution all of the object remaining within the set of objects are cleared.

C. Threading

Salamander supports threading built-in to the virtual machine. A thread can be created by using the built-in function ”Thread(function, (1,))”. The function takes in a function to be executed by the thread and a tuple containing the arguments to be used for the function to be threaded and returns a Thread object. A thread object contains a thread, the function definition of the function be threaded and a tuple holding the arguments to passed into the function. To start the thread, the built-in function ”start(thread)” is used. This function starts the thread passed to it. Salamanders Virtual Machine handles starting threads by starting a c++ thread that will create a new instance of a Virtual Machine. The Virtual Machines are added to a Virtual Machine stack which holds all of the created Virtual Machine threads. The newly created Virtual Machine has its stack initialized with the function definition pointed to by the Salamander Thread object along with each of the arguments contained within the argument tuple. A new function call is then generated for the new Virtual Machine to initialize its function call stack. After the new Virtual Machine has had its program stack and function call stack initialize it can start executing byte code instruction. After the Virtual Machine is done executing, it is popped off of the stack of virtual machines. and the thread finishes its execution. Threads can also be joined by using the ”join(thread)” function. This function takes in a Salamander Thread object and halts code execution until the provided thread is finished executing. This is done by joining the c++ thread contained within the Salamander Thread Object.

Salamander also provides built-in support for creating locks to allow threads to operate on global variables. A Salamander lock can be created by use the built-in function ”Lock()” which returns a salamander Lock object. A Salamander Lock object contains a mutex lock. A lock can be acquired by using the ”getLock(lock)” function. This function accepts a salamander lock object and halts thread execution until the mutex lock is acquired. The lock can then be released by using the built-in function ”releaseLock(lock)”, which takes in a Salamander lock object and releases the mutex lock for other threads to acquire. Variables defined as global in salamander are stored into a global variable pool that is shared by all the concurrently executing threads. So, for threads to be able to access a variable defined outside of their Virtual Machine, the variable must have been declared as a global variable using the global keyword. Therefore a lock shared by multiple threads will need to be declared as a global variable. If the user wants a variables change to be seen across all of the running threads, then that variable will have be defined as a global variable, otherwise each thread will have their own copy of the variable and changes to it will not be reflected across threads.

IV. Standard Library

Salamanders standard library provides access to user console input and output functionality, reading from and writing to files, and a socket programming API.

A. I/O

A user can print output to standard output by using the ”print(string)” function. The print function takes in a single argument and prints its string representation to standard output. A user can also get input from standard input by using the ”input(string)” function. The input function takes in a string argument to print out to standard output to prompt a user for input. Program execution then halts while it waits for the user to give an input through the console. The function returns what the user entered into standard input as a string. A file can be opened in salamander by using the ”open(string, string)” function. The function takes in a string for its first argument which should be a path to a file to open. The second argument is also a string which should denote if the file is being opened for reading (”r”), writing (”w”), or appending (”a”). The function returns a Salamander File object which contains the path of the file and the file stream to read from or write to the file. If the file could not be opened then an error is returned. Opened files can be read in by using the function ”read(file)”. The function takes in a salamander file object to read from and returns the contents of the file as a string. A file and be written to by using the ”write(file, string)” function. The function accepts two argument, the first being a salamander File object to write too and the second being the string to write to the file. A Salamander file object can be closed by using the ”close(file)” function, which accepts one argument, a salamander file object to be closed. If a file is already opened then it cannot be opened again until it is closed first.

B. Socket Programming

A socket can be created by using the built-in function ”Socket()”. This function takes in nor arguments and returns a salamander Socket object. A Salamander socket object has a file descriptor for the socket and the address information for the socket. Salamander sockets use the AF_INET family and SOCK_STREAM data streams. A socket can be binded to a port by using the ”bind(socket, port)” function. The function takes in a salamander socket object to bind and the port number to bind too. Sockets can listen for for a connection by using the ”listen(socket)” function. The function takes in a salamander socket object and halts program execution until a connection is made. Sockets can accept connections by using the ”accept(socket)” function. This function takes in a Salamander socket object returns a new socket containing the newly accepted connection. sockets can receive data by using the ”recv(socket, int)” function. This function takes in two arguments, the first being the socket to receive data and the second an integer specifying the amount of data to accept. The data received is read and returned by the function as a string. Sockets can send data by using the ”send(socket, string)” function. This function accepts 2 arguments, the first being a salamander socket object to send the data through, and the second being a string containing the data to be sent. Sockets can initiate connections with other sockets by using the ”connect(socket, string, int)” function. The function accepts three argument, the firs is a salamander socket object to hold the connection, the second is a string containing the address to connect to, and the 3rd is a int containing the port number to connect to.

C. Type conversion functions and others

A datatype can be converted to its string representation by using the ”toStr(Object)” function. This function accepts any value and returns the string representation of it. The function ”toInt(Object)” will attempt to convert a given value to an integer. If it cannot be done an error is thrown. The function ”toFloat(Object)” functions in the same manner, attempting to convert the given argument to a float. The ”range(int, [int])” takes in at least one argument and an optional second argument. Both arguments should be integers, and if two arguments are given, the first argument should be less than the second argument. If a single argument is given the function will return a list with elements starting from 0 to the number given. If two arguments are given then a list with values ranging from the first argument to the second argument is returned. In order to clear up ambiguity between declaring a empty set and an empty dictionary, An empty dictionary should be created with the ”Dict()” function which takes in no arguments and returns an empty salamander dictionary. Attempting to create a empty dictionary by doing ”{}” will result in creating an empty set, not a dictionary. The size of a list, tuple, set, or dictionary can be retrieved through the use of the ”len(iterable)” function. The Function takes in one argument and expects it to be either a list, tuple, set, or dictionary and returns an integer value that is the size of the structure provided to the function.

V. Conclusions and Challenges

Salamander is a bytecode compiled, high-level general purpose programming language. Salamander draws inspiration from python, aiming to replicate its code readability and sue of logical line and indention to divide up program blocks and statements. Salamander is statically typed and garbage collected, with built in support for dynamic typing. It supports multiple programming paradigms, including procedural and object oriented programming. Salamander comes with a standard library that enables the use of standard input and output, file handling, socket programming, and enables the use of threading.

One challenge that was prominent through out the development of salamander was implementing many of the user friendly features found in python into salamander with static typing. On example would be python enabling the user to dynamically ad fields to class instances. This of course can not be done while maintaining static typing on the fields of the class. So the solution I implemented was to allow for a class to have both statically typed fields and dynamically typed ones. This allows statically typed fields to be defined within the class definition its self and allows the user to add any fields they want on the instance dynamically through out their program. In order to implement support for dynamic typing side by side static typing, Salamander has a "Object" data-type that can have any data type stored into it and will have its typing be done at run time instead of at compile time. Any expression that uses a "Object" data type will be dynamically typed at run time. This challenge also carried over to implementing easy to use, python style, list, sets, tuples, and dictionaries. These Python structures can hold any data type with in them and mix and match data types through out. IN order to maintain this functionality, each element within one of these structures is treated as a "Object" data type and will be typed dynamically at run time when they are accessed. This means that the target in "for" loops must be an object type because the values being iterated over are typed as objects. Other challenges included challenges due to external forces, such as the project taking almost a month to actually get approved resulting in a late start, and then again with the syllabus taking almost another month to be approved resulting in some initial work that was done being misguided and needing to be reworked.

References

[1] "Compiler Design - Lexical Analysis." Tutorialspoint, www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm#:~:text=Lexical%20analysis%20is%20the%20first, comments%20in%20the%20source%20code.

[2] "Lexical Analysis." Wikipedia, Wikimedia Foundation, 29 Apr. 2021, en.wikipedia.org/wiki/Lexical_analysis.

[3] "Introduction of Lexical Analysis." GeeksforGeeks, 9 June 2020, www.geeksforgeeks.org/introduction-of-lexical-analysis/.

[4] "Lexical Analysis in Compiler Design with Example." Guru99, www.guru99.com/compiler-design-lexical-analysis.html.

[5] "2. Lexical Analysis¶." 2. Lexical Analysis - Python 3.9.5 Documentation, docs.python.org/3/reference/lexical_analysis.html.

[6] Nystrom, Robert. Crafting Interpreters, 2015, craftinginterpreters.com/.

[7] "Syntax Analysis: Compiler Top Down; Bottom Up Parsing Types." Guru99, www.guru99.com/syntax-analysis-parsing-types.html#:~:text=Syntax%20Analysis%20is%20a%20second,the%20programming%20language%20or%20not.

[8] "Compiler Design - Syntax Analysis." Tutorialspoint, www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm.

[9] "Introduction to Syntax Analysis in Compiler Design." GeeksforGeeks, 21 Nov. 2019, www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/.

[10] https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/02/Slides02.pdf

[11] "Parsing." Wikipedia, Wikimedia Foundation, 30 Apr. 2021, en.wikipedia.org/wiki/Parsing.

[12] "6. Expressions¶." 6. Expressions - Python 3.9.5 Documentation, docs.python.org/3/reference/expressions.html.

[13] "2. Introduction to Pratt Parsing and Its Terminology¶." 2. Introduction to Pratt Parsing and Its Terminology - Typped Documentation, abarker.github.io/typped/pratt_parsing_intro.html#:~:text=Pratt%20parsing%20is%20a%20type,can%20easily%20handle%20operator%20precedences.

[14] "Top-Down Operator Precedence Parsing." Eli Benderskys Website ATOM, eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing.

[15] Dmy. "Dmy/Elm-Pratt-Parser." GitHub, github.com/dmy/elm-pratt-parser.

[16] Pratt Parsing and Precedence Climbing Are the Same Algorithm, www.oilshell.org/blog/2016/11/01.html.

[17] "Top down Operator Precedence Parsing in Go." Cristian Dima - Top down Operator Precedence Parsing in Go, www.cristiandima.com/top-down-operator-precedence-parsing-in-go/.

[18] https://home.adelphi.edu/~siegfried/cs372/372l8.pdf

[19] "Compiler." Wikipedia, Wikimedia Foundation, 15 May 2021, en.wikipedia.org/wiki/Compiler#Front_end.

[20] "Compiler Design - Semantic Analysis." Tutorialspoint, www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm.

[21] "Semantic Analysis in Compiler Design." GeeksforGeeks, 22 Apr. 2020, www.geeksforgeeks.org/semantic-analysis-in-compiler-design/.

[22] "Phases of Compiler with Example." Guru99, www.guru99.com/compiler-design-phases-of-compiler.html.

[23] "Generating Bytecode." Strumenta, 12 June 2020, tomassetti.me/generating-bytecode/.

[24] Federico Tomassetti, et al. "Building a Language: Generating Bytecode - DZone Java." Dzone.com, 18 Sept. 2016, dzone.com/articles/generating-bytecode.

[25] "Bytecode." Wikipedia, Wikimedia Foundation, 16 May 2021, en.wikipedia.org/wiki/Bytecode.

[26] Chapter 22 Byte Code Generation, esper.espertech.com/release-7.1.0/esper-reference/html/bytecodegen.html.

[27] https://www.cs.fsu.edu/~engelen/courses/COP562107/Ch2.pdf

[28] "Optimizing Compiler." Wikipedia, Wikimedia Foundation, 20 Apr. 2021, en.wikipedia.org/wiki/Optimizing_compiler.

[29] Kexugit. "Compilers - What Every Programmer Should Know About Compiler Optimizations." Microsoft Docs, docs.microsoft.com/en-us/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations.

[30] "Haoyi's Programming Blog." How an Optimizing Compiler Works, www.lihaoyi.com/post/HowanOptimizingCompilerWorks.html.

[31] "Compiler Optimization." Compiler Optimization - an Overview — ScienceDirect Topics, www.sciencedirect.com/topics/computer-science/compiler-optimization.

[32] "Code Optimization in Compiler Design." GeeksforGeeks, 3 July 2020, www.geeksforgeeks.org/code-optimization-in-compiler-design/.

[33] "Constant Folding." Wikipedia, Wikimedia Foundation, 8 Apr. 2021, en.wikipedia.org/wiki/Constant_folding.

[34] "Strength Reduction." Wikipedia, Wikimedia Foundation, 27 Nov. 2020, en.wikipedia.org/wiki/Strength_reduction.

[35] "Code Optimization in Compiler Design." GeeksforGeeks, 3 July 2020, www.geeksforgeeks.org/code-optimization-in-compiler-design/.

[36] "Tail Call." Wikipedia, Wikimedia Foundation, 28 Mar. 2021, en.wikipedia.org/wiki/Tail_call.

[37] Skarupke, Malte. "I Wrote The Fastest Hashtable." Probably Dance, 26 Feb. 2017, probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/.

[38] Skarupke. "Skarupke/flat_hash_map." GitHub, 15 July 2018, github.com/skarupke/flat_hash_map/blob/master/flat_hash_map.hpp.

[39] Kazanov, Vladimir. "Home-Grown Bytecode Interpreters." Medium, Bumble Tech, 31 Dec. 2020, medium.com/bumble-tech/home-grown-bytecode-interpreters-51e12d59b25c.

[40] Paul, et al. "How To Create An Interpreter Of Bytecodes In Fifteen Minutes." Information Technology Blog, 22 Oct. 2018, www.smartspate.com/how-to-create-an-interpreter-of-bytecodes/.

[41] "Create Your Own Programming Language with Rust." Introduction - Create Your Own Programming Language with Rust, createlang.rs/intro.html.

[42] "Garbage Collection (Computer Science)." Wikipedia, Wikimedia Foundation, 8 May 2021, en.wikipedia.org/wiki/Garbage_collection_(computer_science).

[43] https://www.seas.harvard.edu/courses/cs153/2019fa/lectures/Lec25-Garbage-collection.pdf

[44] "Mark-and-Sweep: Garbage Collection Algorithm." GeeksforGeeks, 4 June 2018, www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/.

[45] "Tracing Garbage Collection." Wikipedia, Wikimedia Foundation, 28 Dec. 2020, en.wikipedia.org/wiki/Tracing_garbage_collection.

[46] 13.2.1 Mark-and-Sweep Garbage Collection, lambda.uta.edu/cse5317/notes/node47.html.

Appendix A
Comments

```
# this is the first comment
int spam = 1  # and this is the second comment
          # ... and now a third!
str text = "#␣This␣is␣not␣a␣comment␣because␣it's␣inside␣quotes."
print(spam)
print(text)
```

Appendix B
Numbers

```
print( 2+2 ) # 4
print( 50 - 5*6 ) # 20
print( (50-5*6) / 4 ) # 5
print( 8/5 )  # 1.6
print( 17/3 ) # 5.66667

# floor division discards the fractional part always retunring an integer
print( 17 // 3) # 5

# the % operator returns the remainder of the division
print( 17 % 3 ) # 2
print( 5 * 3 + 2 ) # 17

# 5 squared
print( 5 ** 2 ) # 25

# 2 to the power of 7
print( 2 ** 7 ) # 128

int width = 20
int height = 5 * 9
print( width * height ) # 900

# n # will throw a variable undefined error

print( 4 * 3.7 - 1 ) # 13.8

float tax = 12.5 / 100 # division always returns a float
float price = 100.50
print( price * tax ) # 12.5625
```

Appendix C
Strings

```
print('spam␣eggs') # single quotes
print('doesn\'t') # use \' to escape the single quote...
print("doesn't") # ...or use double quotes instead
print('"Yes,"␣they␣said.')
print("\"Yes,\"␣they␣said.")
print('"Isn\'t,"␣they␣said.')

str s = 'First␣line.\nSecond␣line.'  # \n means newline
print(s) # with print(), \n produces a new line
print('C:\some\name')  # here \n means newline!
print(r'C:\some\name')  # note the r before the quote
print("sal" + "aman" + "der") # strings can be concatinated with plus operator
str text = 'strings␣can␣be␣concatinated' + '␣together␣with␣the␣"+"␣operator'
print(text)
print( text + "␣in␣salamander")
```

Appendix D
Lists

```
list squares = [1, 4, 9, 16, 25]
print(squares)
print(squares[0]) # indexing returns the item
print(squares[-1])
print(squares[3:5]) # slicing returns a new list

print(squares + [36, 49, 64, 81, 100]) # lists can be concatinated

list cubes = [1, 8, 27, 65, 125]  # something's wrong here
print( 4**3 ) # 64
cubes[3] = 64  # replace the wrong value
print(cubes) # [1, 8, 27, 64, 125]

list letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print(letters)
letters[2:5] = ['C', 'D', 'E']
print(letters)

# now remove them
letters[2:5] = []
print(letters)

letters = ['a', 'b', 'c', 'd']
print(len(letters))

list a = ['a', 'b', 'c']
list n = [1, 2, 3]
list x = [a, n]
print(x) # [[a, b, c], [1, 2, 3]]
print(x[0]) # [a, b, c]
print(x[0][1]) # b
```

Appendix E
Loops

```
int a = 0
int b = 1
while a < 10:
    print(a)
    a = b
    b = a+b

int i = 256*256
print('The␣value␣of␣i␣is␣' + toStr(i))
```

Appendix F
If statments

```
int x = toInt( input("Pleae enter an integer: ") )

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

Appendix G
For statements

```
list words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w + " " + len(w))

for i in range(5):
    print(i)

list a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i + " " + a[i])

print(range(10))
```

Appendix H
Pass statement

```
class MyEmptyClass:
    pass

def log():
    pass

while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

Appendix I
Defining Functions

```
def fib(int n):
    int a = 0
    int b = 1
    while a < n:
        print(a)
        a = b
        b = a+b

    print(a)

fib(2000)
```

```
Object f = fib
f(100)

fib(0)
print(fib(0))

def list fib2(int n):
    list result = []
    int i = 0
    int a = 0
    int b = 1
    while a < n:
        result[i] = a
        i = i+1
        a = b
        b = a+b

    return result

list f100 = fib2(100)
print(f100)
```

APPENDIX J
THE DEL STATEMENT

```
list a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a)

del a[2:4]
print(a)
```

APPENDIX K
TUPLES

```
tuple t = (12345, 54321, 'hello!')
print(t[0])
print(t)

tuple u = (t, (1,2,3,4,5))
print(u)

#t[0] = 88888

tuple singleton = ('hello', )
print(len(singleton))
print(singleton)
```

APPENDIX L
SETS

```
set basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)

print('orange' in basket)
print('crabgrass' in basket)
```

APPENDIX M
DICTIONARIES

```
dict tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print(tel)

print(tel['jack'])

del tel['sape']
tel['irv'] = 4127
print(tel)

print('guido' in tel)
print('jack' not in tel)

for key in tel:
    print(key + ":" + tel[key])
```

APPENDIX N
IMPORTING

```
import './functions.sal'

fib(1000)
print(fib2(100))
```

APPENDIX O
FORMATTED STRINGS

```
int year = 2016
str event = 'Referendum'
print(f"Results␣of␣the␣{year}␣{event}")

int yes_votes = 42572654
int no_votes = 43132495
float percentage = yes_votes / (yes_votes + no_votes)
print(f"{yes_votes}␣YES␣votes␣{percentage}")

str s = 'Hello,␣world'
print(toStr(s))
print(toStr(1/7))

float x = 10 * 3.25
int y = 200 * 200
s = 'The␣value␣of␣x␣is␣' + toStr(x) + ',␣and␣y␣is␣' + toStr(y) + '...'
print(s)

dict table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
for name in table:
    print(f"{name}␣==>␣{table[name]}")

str animals = 'eels'
print(f"My␣hovercraft␣is␣full␣of␣{animals}.")
```

Appendix P
Files

```
file writer = open("./file.txt", "w")
write(writer, "This␣will␣be␣written␣to␣the␣file\n")
close(writer)

file reader = open("./file.txt", "r")
print(read(reader))
close(reader)

file appender = open("./file.txt", "a")
write(appender, "This␣will␣be␣appended␣to␣the␣file\n")
close(appender)

reader = open('./file.txt', "r")
print(read(reader))
```

Appendix Q
Scope

```
def scope_test():
    str spam = "test␣spam"

    def do_local():
        str spam = "local␣spam" # creates new spam variable

    def do_nonlocal():
        spam = "nonlocal␣spam" # gets own COPY of spam variable
        print("Inside␣nonlocal:␣" + spam)

    def do_global():
        global str spam = "global␣spam" # creates a spam varibale in global scope.


    do_local()
    print("After␣local␣assignment:␣" + spam)
    do_nonlocal()
    print("After␣nonlocal␣assignment:␣" + spam)
    do_global()
    print("After␣global␣assignment:␣" + spam)

scope_test()
print("In␣global␣scope:␣" + spam)
```

Appendix R
Classes

```
class MyClass:
    int i = 12345
    def __init__():
        self.data = []

    def str f():
        return "hello␣world"

MyClass x = MyClass()
```

```
print(x.i)
print(x.f())
print(x.data)

class Complex:
    def __init__(float real, float imag):
        self.r = real
        self.i = imag

Complex c = Complex(3.0, -4.5)
print((c.r, c.i))

x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)

Object xf = x.f
while True:
    print(xf())
```

Appendix S
Instances

```
class Dog:
    str kind = 'canine' # class variable shared by all instances
    list tricks = []

    def __init__(str name):
        self.name = name # instance variable unique to each instance

    def add_trick(str trick):
        self.tricks[len(self.tricks)] = trick

Dog d = Dog('Fido')
Dog e = Dog('Buddy')
print(d.kind) # shared by all dogs
print(e.kind) # shared by all dogs
print(d.name) # unique to d
print(e.name) # unique to e

d.add_trick('roll␣over')
e.add_trick('play␣dead')
print(d.tricks) # shared by all dogs,
                # an instance varibale should be used instead for desiered functionality.
```

Appendix T
More classes examples

```
class Warehouse:
    str purpose = 'storage'
    str region = 'west'

Warehouse w1 = Warehouse()
print(w1.purpose + "␣" + w1.region)
Warehouse w2 = Warehouse()
w2.region = 'east'
```

```
print(w2.purpose + "␣" + w2.region)

def float f1(float x, float y):
    return x if x < y else y

class C:
    Object f = f1

    def str g():
        return "hello␣world"

C c = C()
print(c.f)
Object f2 = c.f
print(f2(1,2))

class Bag:
    def __init__():
        self.data = []

    def add(Object x):
        self.data[len(self.data)] = x

    def addTwice(Object x):
        self.add(x)
        self.add(x)

Bag bag = Bag()
bag.add(1)
bag.addTwice("two")

print(bag.data)
```

Appendix U
Inheritance

```
class Person:
    def __init__(str fname, str lname):
        self.firstname = fname
        self.lastname = lname

    def printname():
        print(self.firstname + "␣" + self.lastname)

Person x = Person("John", "Doe")
x.printname()

class Student(Person):
    def __init__(str fname, str lname, int year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome():
        print("Welcome␣" + self.firstname + "␣" + self.lastname + "␣to␣the␣class␣of␣" +
        self.graduationyear)
```

```
Student s = Student("Will", "Dahl", 2021)
s.welcome()
```

Appendix V
Iterators

```
for e in [1,2,3]:
    print(e)

for e in (1,2,3):
    print(e)

for key in {"one": 1, "two": 2}:
    print(key)
```

Appendix W
Threads

```
global int i = 0
global lock l = Lock()

def thread_fuction(int n):
    while i < n:
        getLock(l)
        print(i)
        i=i+1
        releaseLock(l)

thread t1 = Thread(thread_fuction, (100,))
thread t2 = Thread(thread_fuction, (100,))

start(t1)
start(t2)

join(t1)
join(t2)
```

Appendix X
Socket Programming

A. Server

```
socket s = Socket()
bind(s, 65432)
listen(s)
socket conn = accept(s)
str data = recv(conn, 1024)
send(conn, data)
```

B. Client

```
socket s = Socket()
connect(s, '127.0.0.1', 65432)
send(s, "Hello, world")
str data = recv(s, 1024)

print(data)
```