Recursive Algorithms for Parsing the Ram Scripting Language into Abstract Syntax Trees in Python

William S. Assad*, Ariel Chouminov†,
Ramya Chawla†, Zain Lakhani†

April 16, 2021

Abstract

For the student of Computer Science, we introduce a systematic algorithm for parsing a new scripting language into Abstract Syntax Trees (ASTs) in Python. An induction-based structure for parsing is used where the base case represents an instance of `Line` and the recursive step represents an instance of a `Block` child (enumerated type). Once the nested structure is created, we recursively parse each object into a specific AST. The conventional approach is used for the evaluate method; the abstract classes `Statement` and `Expr` are defined. Instead of presenting a standard technical parsing algorithm, *Ram* is implemented with minimal emphasis on asymptotic running-time complexity. Like Python, many features of *Ram* are "syntactic sugar". We see that for an input family of n lines of *Ram* code and maximum nesting of depth k, the average running time complexity of the `.parse` and `.evaluate` algorithm is $\mathcal{O}(kn)$.

1 Introduction and Theory

Programming languages are built using Abstract Syntax Trees (ASTs), a recursive data structure that represents written code [1]. Programmers write code in the form of large strings, which are parsed into ASTs and recursively evaluated [1, 2]. As humans, we often take this necessary translation for granted, given that we can simply read programs as text [1]. Any computer, however, will interpret this text, translate it several times, and then produce an output with incredible efficiency.

*Department of Mathematics, University of Toronto
†Department of Computer Science, University of Toronto

Programming languages present humans the opportunity to interact with computers in unique ways. One of these languages is Python, a high-level programming language that is relatively easy to learn. While it has many advantages as a first programming language for new programmers, there are some key disadvantages. Mainly, it differs structurally from other programming languages in terms of indented blocks and variables that represent multiple data types[1]. This means it offers beginner programmers an introduction into the world of programming while making their transition into other languages quite difficult.

Translation to English is another key obstacle in introducing programming languages to new students of Computer Science. The intuition behind writing certain lines of code is often masked by the nuances of a particular language. For example, there is no obvious reason why the line of Python code `print([(x, x % 2 == 0) for x in range(10)])` should display a list of numbers from 0 to 9 and whether or not they are even. Hence, we introduce the programming language (Ram) to be easily translatable to English and help better prepare beginner programmers to use other languages. We discuss a method of implementing the *Ram* programming language in Python using a systematic algorithm.

2 Implementation

2.1 Ram Syntax

Before diving directly into the code and our plan of implementation, we will clearly define all the syntactical structure of *Ram* code. We emphasize the readability of this code and the fluency with which it can be translated to English. Table 1 and Table 2 contain various statements in Python with a side-by-side comparison of their equivalent statements in *Ram*.

Python Syntax	**Ram Syntax**

`for x in range(5):`
`  print(x)`
	`loop with x from 0 to 4 {`
	`  display x`
	`}`

`x = 5`	`set integer x to 5`
`letters = 'hello'`	`set text letters to ''hello''`

Table 1: Comparison of Python and Ram Syntax

[1] Indentation in Python represents nesting instead of standard curly braces.

Python Syntax	Ram Syntax

`var1 = 10`
`if var1 == f(8):`
`  print(‘Equal!’)`
`elif (var1 + var2) == 5:`
`  print(‘Equals 5’)`
`else:`
`  print(‘:(’)`
	`set integer var1 to 10`
	`if var1 is f[x=8] {`
	`  display ‘‘Equal!’’`
	`} else if (var1 + var2) is (5) {`
	`  display ‘‘Equals 5’’`
	`} else {`
	`  display ‘‘:(’’`
	`}`

`def f(x, y) -> int:`
`  x = x + y`
`  return x`
	`new function f takes (x,y) {`
	`  reset integer x to x + y`
	`  send back x`
	`}`

Table 2: Comparison of Python and Ram Syntax

2.2 AST Implementation

Processing *Ram* code ultimately results in the creation of ASTs, which we define in Python. We begin by defining various statements and expressions including (but not limited to) for-loops, functions, if-statements, assignment statements, and print statements. Class inheritance is structured using an abstract class representing a statement, `Statement`, and an abstract subclass representing an expression, `Expr` (Figure 1).

```
class Statement:
    """An abstract class representing a Python statement.
    """
    def evaluate(self, env: dict[str, Any]) -> Optional[Any]:
    """Evaluate the statement in a given environment."""
        raise NotImplementedError

class Expr:
    """An abstract class representing a Python expression.
    """
    def evaluate(self, env: dict[str, Any]) -> Optional[Any]:
    """Evaluate the statement in a given environment."""
        raise NotImplementedError
```

Figure 1: `Statement` and `Expr` Abstract Classes

ASTs are a form of trees since a `Statement` can be composed of many `Expr`'s which can also be composed of many different other `Expr`'s, which represent subtrees. For example, a binary operation expression `BinOp(Expr)` is defined in terms of a left `Expr` and a right `Expr` as seen in Figure 2. This allows for any form of nesting, provided that the left and right subtrees evaluate to floating-point values (leafs are instances of `Num` or `Name`). All other ASTs have a similar structure, being composed of other statements and expressions.

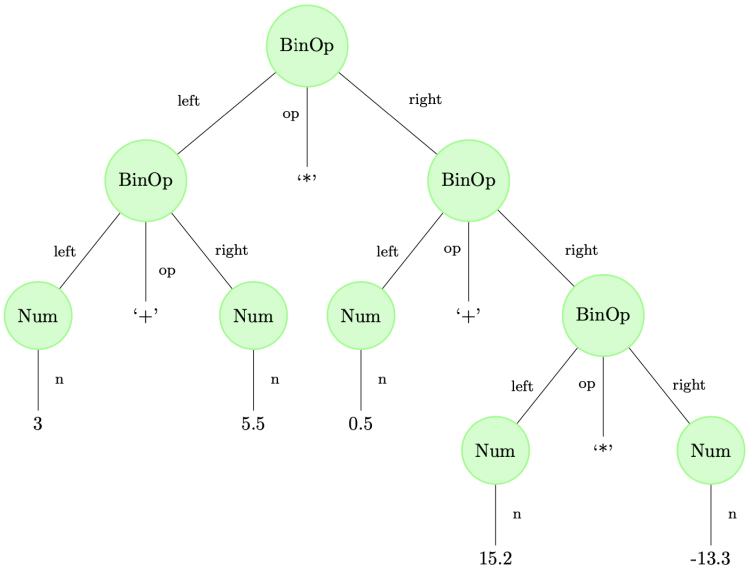

Figure 2: `BinOp` Tree Structure [1]

2.3 Parsing Algorithms

Instead of reading from left to right, the interpreter must convert linear plain text into the deeply nested AST structures seen in Figure 2. This is the process of parsing. More formally, parsing is the process of analyzing a string of symbols and or words that are presented in either natural language, computer language, or data structure and converting them into ASTs [5]. Like Python, the code written by our users will initially be processed as the basic data types — strings, booleans, and integers. For example, the ram data type `text` is equivalent to the AST `String` whose `evaluate` method returns a Python `str`. Here we describe in detail the four step parse and evaluate algorithm.

1. Read in the `.ram` file as a list of tuples.

   The `.ram` file is read in as a list of tuples where each tuple contains the line and the line number.

2. Recursively create the nested block structure.

   In `parsing.py`, we introduce an abstract class `Block` (Figure 3) which has three sub-classes — `Loop`, `Function`, and `If`. A complicated recursive algorithm `make_child` that runs in `Block.__init__` creates the proper nested structure. The function determines the block type of the caller and creates a new specific block instance. The newly created block recursively parses through all of its children, which runs through the same process as the parent. The parent class assigns its caller a new sub-class type (for example a `LoopBlock`), such that when the block is parsed, the parsing specific to that block type is executed. Similar to `Statement` and `Expr`, these classes and sub-classes are used to create a nested structure. The use of these classes allows us to recursively parse each block of code.

3. Recursively parse the nested block structure.

   Each type of `Line` and child of `Block` has its own `parse` method which recursively creates an AST. For example, `IfBlock.parse` would return the `If(Statement)` AST.

4. Evaluate the resulting ASTs.

   As described in Figure 1, each AST has an `evaluate` method. For example, `BinOp.evaluate` would return a floating point value, as represented in Python.

```
class Block:
    """A block of Ram code to parse.  """
    def parse(self) -> Statement:
    """Parse a block of Ram code."""
        raise NotImplementedError
```

Figure 3: `Block` Abstract Class

Ram is intended to support simple arithmetic expressions, including the use of brackets. Regardless of the number of smaller computations that are nested within any binary operation, we need to evaluate the expression in terms of the order of operations. We were able to successfully implement very complex recursive algorithms such as `lexify` and `pedmas` in *parsing/parse_numeric.py*. Recursion was an essential tool in many algorithms; without it, we would have great difficulty locating and keeping track of various blocks of code. To give an example of this process, let's run through the specific algorithm for parsing this line of Ram code:

```
set integer var1 to 4 * 3 + (8 - 4 * 2)
```

We first read the line as a tuple consisting of the line itself as a string and the corresponding line number in the enclosing file. The keyword `set` is detected, meaning the line is to be parsed as a variable assignment statement. The keyword `integer` is then detected, meaning the line is to be parsed more specifically as a numerical value. The variable name, `var1`, is stored and the numeric expression `4 * 3 + (8 - 4 * 2)` is run through the `lexify` algorithm. This algorithm is fairly complex, using recursion to create the relevant nesting structure around the binary operations, following the mathematical order of operations. Once this process is finished, `var1` is assigned to the `BinOp` AST returned by the `Line.parse` method. Finally, the statement is evaluated and the environment is updated to contain the new variable.

The intuition is to take blocks of code and separate them into loop blocks, if blocks, and function blocks. Then the body portion of those blocks can contain nested blocks within, and this forms the nested structure we addressed earlier. Once these blocks are parsed, the program is able to treat them as functions and can accurately identify and interpret each statement of code.

We also ensure that the nesting of the blocks occurs in the correct order through the `process_ram` function in `process.py`. This function takes in *Ram* code as a list of tuples in the form: `[(<line>, <line_number>), ...]` and returns a list which correctly nests the blocks provided. This transform the data into the proper nested structure that is assumed in step (2) of the parse and evaluate algorithm. This nested structure is directly associated with trees— a non-binary tree where each node has one child and each child represents another layer of nesting.

As a result of the combination of these ASTs, we are able to successfully create an interactive component and allow the user to freely experiment with Ram code and even build simple programs.

Referencing external libraries, we used `sys`, `os` and `platform`, which all come packaged with python and do not require any additional installation. We used `sys` to read the command line argument when calling `ram <filepath>.ram` in order to get the file path in the `verify_file` function in `process.py`. In `install.py`, `platform` and `os` are used in the `InstallRam` class. An alias that links to an executable `main.py` is created at `usr/local/bin/ram` using `os`. This is only done (using `platform`) provided the platform is Darwin.

3 Discussion

Upon testing numerous different sample files with many different variations of code — a combination of assignment statements, loops, if statements, and functions all nested within one another — the four step parse and evaluate algorithm accurately processes *Ram* code.

We were able to successfully create a new programming language that is easily translatable to English and helps better prepare beginner programmers to use another programming language. The syntax that we had in mind was properly executed and properly reinforced; when users do not correctly follow the *Ram* syntax requirements, the appropriate `RamException` is raised.

4 Notes and Acknowledgement

See the GitHub repository for instructions on how to install *Ram* and write simple code in `.ram` files.

The development of this project (motivation and theory) would not be possible without *CSC111* and David Liu at the University of Toronto.

References

[1] David Liu. *CSC111 Lectures*. University of Toronto. 2021.

[2] Dennis Howe. *Free On-line Dictionary of Computing*. Internet Encyclopedia Project. 2008.

[3] *Abstract Syntax Trees*. Python Documentation. Available from: https://docs.python.org/3/library/ast.html

[4] *Natural Language Toolkit*. Python Documentation. Available from: https://www.nltk.org/

[5] Chapman, Nigel P. *LR Parsing: Theory and Practice, Cambridge University Press*. 1987.