# “Ram” — A Beginner’s Introductory Programming Language Using Abstract Syntax Trees in Python

Will Assad, Ariel Chouminov, Ramya Chawla, Zain Lakhani

April 16, 2021

## Problem Description and Research Question

Modern 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 then parsed into ASTs in which the computer can understand and execute [1, 2]. As humans, we often take this necessary translation for granted, given that we can simply read programs as text [1]. The computer, however, interprets this text, translates it several times, and then produces an output with incredibly high speed.

A programming language serves as an instrument to bridge the gap between machines and humans. Although the possibilities are seemingly limitless, a program is only as effective as the developer designs it to be. That is, the necessary tools to develop even the most complex programs are made available to one’s disposal. However, with the vast number of nuances, it can be quite difficult for a beginner to indulge in the world of programming.

Programming languages present humans the opportunity to interact with computers in unique ways. One of these languages is Python, a relatively easy scripting language 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. This means it offers beginner programmers an introduction into the world of programming while making their transition into other languages quite difficult.

The other problem with introducing programming to new students of Computer Science is getting them to understand what a particular syntax represents in English. The intuition behind writing certain lines of code can hide from those who lack experience. 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, **the goal of this project is to create a programming language (Ram) that is easily translatable to English and helps better prepare beginner programmers to use other programming languages.** That is, we’re looking to provide our users with the ability to convert basic English sentences, following the syntax of our language, into lines of Ram code.

## Computational Plan

Before diving directly into the code and our plan of implementation, we will clearly define all the syntactical structures of Ram code. As previously mentioned, an important feature is the readability of this code and the fluency with which it can be translated to English. Table 1 contains various expressions in Python with a side-by-side comparison of how they can be expressed following the syntax of Ram code.

| Python Syntax | Ram Syntax |
|---|---|
| `for x in range(5):`<br>`    print(x)` | `loop with x from 0 to 4 {`<br>`    display x`<br>`}` |
| `x = 5` | `set integer x to 5` |
| `letters = ‘hello’` | `set text letters to ‘‘hello’’` |

| | |
|---|---|
| def f(x: int) -> None:<br>    x = x - 5<br>    print(x) | new function f takes (x) {<br>    reset x to x - 5<br>    display x<br>} |
| var1 = 10<br>var2 = f(8)<br><br>if var1 == var2:<br>    print('They are equal!')<br>elif (var1 + var2) == 5:<br>    print('Equals 5')<br>    var1 = 4<br>else:<br>    print('Nothing happened.') | set integer var1 to 10<br>set integer var2 to f[x=8]<br><br>if (var1) is (var2) {<br>    display ''They are equal!''<br>} else if (var1 + var2) is (5) {<br>    display ''Equals 5''<br>    reset var1 to 4<br>} else {<br>    display ''Nothing happened.''<br>} |
| def compute(x: int, y: int) -> int:<br>    x = x + y<br>    return x | new function f takes (x,y) {<br>    reset x to x + y<br>    send back x<br>} |

Table 1: Comparison of Python and Ram Syntax

Our program consists of three main parts. Initially, we begin by defining various statements and expressions in Python such as for-loops, functions, if-statements, assignment statements, and print statements. We define these statements through the use of Abstract Syntax Trees (Figure 1). We define an abstract class representing a statement, `Statement`, and an abstract subclass representing an expression, `Expr`[1].

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

class Expr:
    """An abstract class representing a Python expression.
    """
    def evaluate(self, env:  dict[str, Any]) -> Optional[Any]:
    """Evaluate this statement with the 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. All other ASTs have a similar structure, being composed of other statements and expressions.

[1]These two abstract classes and subclasses such as `If`, `BoolOp`, and `Loop` are credited to the *CSC111* notes [1].

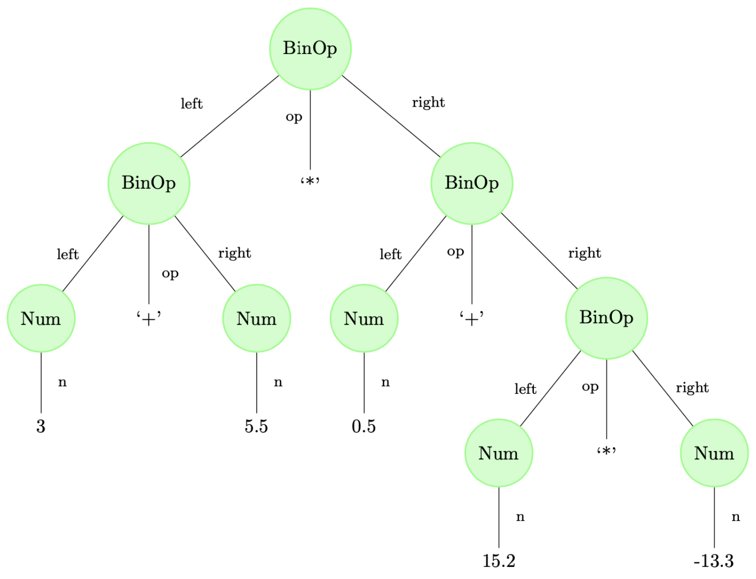


Figure 2: `BinOp` Tree Structure [1]

We as humans take for granted the complexity in interpreting the context of plain text. We read from left to right, and process text with ease. A computer, however, must convert the plain text into the deeply nested structures as seen in Figure 2. This is the process of parsing.

As previously mentioned, parsing, within the context of programming, is the process of analyzing a string of symbols and or words that are presented in either natural language, computer language, or data structures [5]. Similar to Python, the code written by our users will initially be processed as the basic data types — strings, booleans, and integers. However, from there, we begin to parse the data into the ASTs. Since parsing is the most difficult and technically demanding component of our project, we decided to divide and conquer. Parsing numbers, variables, and everything else in between was done separately in three main steps:

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 introduced 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 relevant 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 Loop Block), so 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 AST `If(Statement)`.

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

Our language 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*. This being said, 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)
```

This line is read in 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 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` returned by the `lexify` algorithm. 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 need to ensure that the nesting of the blocks occurs in the correct order. Which brings us to the `process_ram` function in `process.py`. With the assistance of a helper function, we're able to use `process_ram` to take in chunks of Ram code as a list of tuples in the form: `[(<line>, <line number>), ...]` and return a list which correctly nests the blocks provided. Therefore, this function essentially transform the data into a nested structure in an accurate manner. Further, the recursive nature of this function is directly associated with trees, as it can be viewed as a non-binary tree where each node has one child and each child represents a recursive call.

As a result of the combination of these AST's, we are able to successfully create an interactive component and allow the user to freely experiment with Ram code and even build simplistic programs. The overarching theme of simplistic, introductory programming is obtained by following a parsing similar to that of Python, but with a more surface level depth in terms of the features and capabilities of the program. But what the program lacks in sophistication, it makes up for with directness, as users are now able to follow clear syntax guidelines to convert near-standard English convention sentences into actual lines of code.

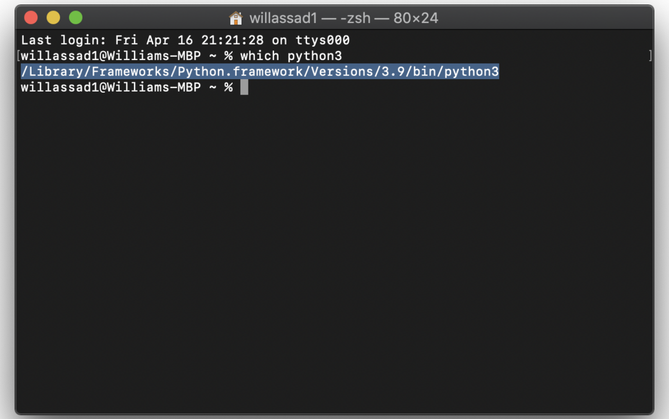
For the libraries, we used `sys`, `os` and `platform`, which all come packaged with python and do not require any additional installation. We used `sys` to read the command line argument when calling `ram <filepath>.ram` in order to get the file path in the `verify_file` function in `process.py`. In `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.

## Running the Program

### Using MacOS

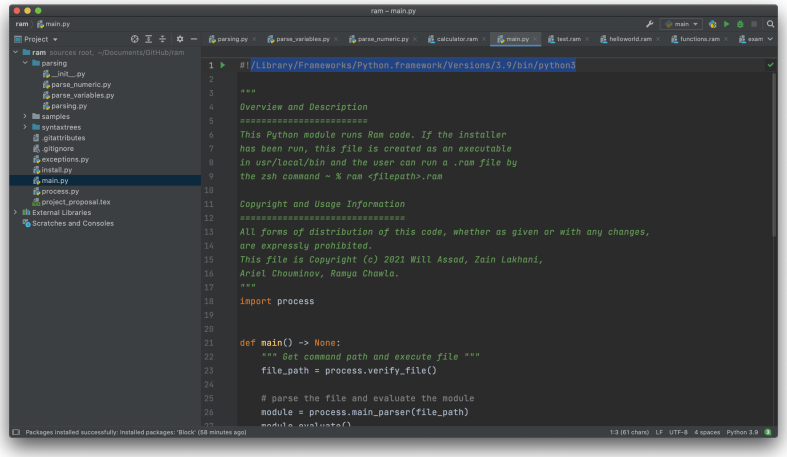
- Python Console
  - You can write a new `.ram` file in any text editor. We even defined custom syntax highlighting for `*.ram` files in PyCharm (see images below)!
  - Run main.py
  - You will be prompted to enter the file path of the .ram file to run.

- Terminal
  - Open Terminal and type `which python` or `which python3` (depending on the version installed).



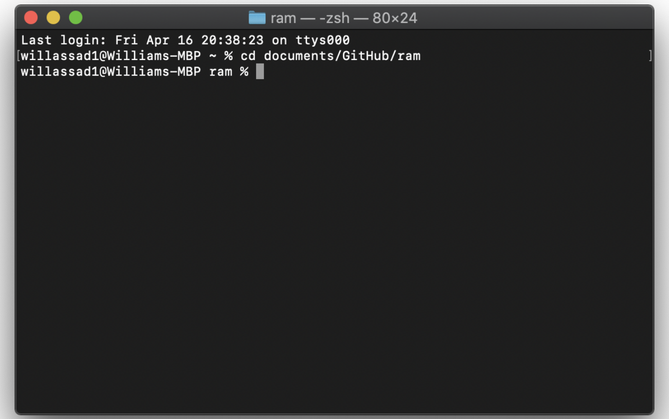
```
Last login: Fri Apr 16 21:21:28 on ttys000
willassad1@Williams-MBP ~ % which python3
/Library/Frameworks/Python.framework/Versions/3.9/bin/python3
willassad1@Williams-MBP ~ %
```

  - Copy the result and then open `main.py` and paste the result after the `!#`.



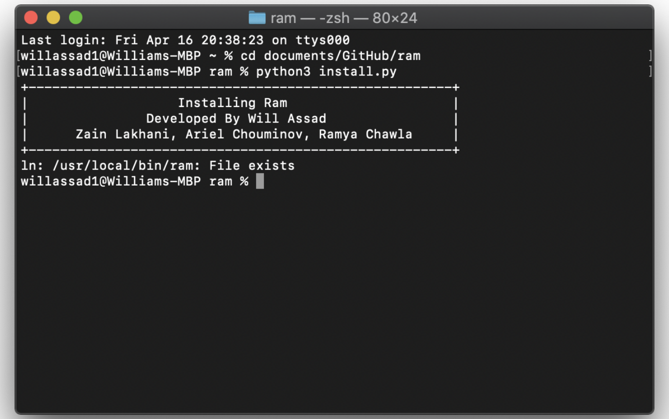
```
#!/Library/Frameworks/Python.framework/Versions/3.9/bin/python3

"""
Overview and Description
========================
This Python module runs Ram code. If the installer
has been run, this file is created as an executable
in usr/local/bin and the user can run a .ram file by
the zsh command ~ % ram <filepath>.ram

Copyright and Usage Information
===============================
All forms of distribution of this code, whether as given or with any changes,
are expressly prohibited.
This file is Copyright (c) 2021 Will Assad, Zain Lakhani,
Ariel Chouminov, Ramya Chawla.
"""
import process


def main() -> None:
    """ Get command path and execute file """
    file_path = process.verify_file()

    # parse the file and evaluate the module
    module = process.main_parser(file_path)
    module.evaluate()
```

  - Navigate to the main `ram` program directory (wherever it is stored).



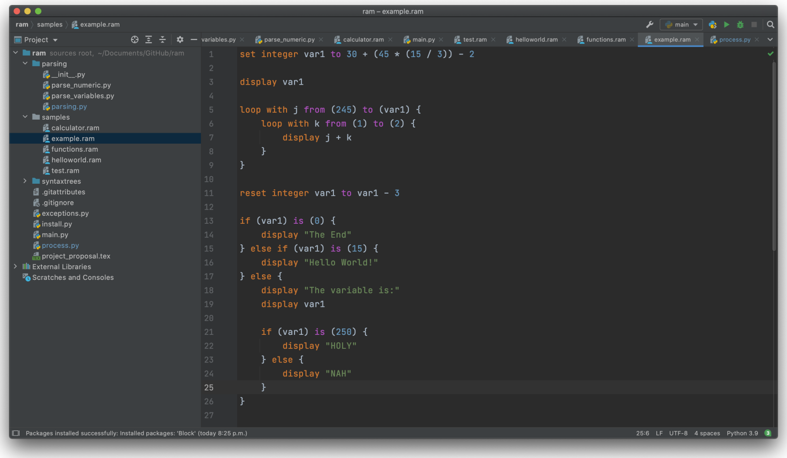
```
Last login: Fri Apr 16 20:38:23 on ttys000
willassad1@Williams-MBP ~ % cd documents/GitHub/ram
willassad1@Williams-MBP ram %
```

– Now run the command `python install.py` or `python3 install.py` (depending on version used).



```
Last login: Fri Apr 16 20:38:23 on ttys000
willassad1@Williams-MBP ~ % cd documents/GitHub/ram
willassad1@Williams-MBP ram % python3 install.py
+------------------------------------------------------+
|                    Installing Ram                    |
|                Developed By Will Assad               |
|       Zain Lakhani, Ariel Chouminov, Ramya Chawla    |
+------------------------------------------------------+
ln: /usr/local/bin/ram: File exists
willassad1@Williams-MBP ram % 
```

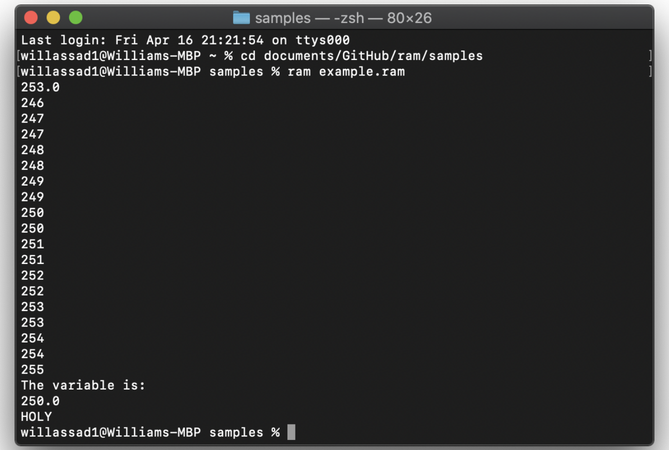
– You can now write a new `.ram` file in any text editor. We even defined custom syntax highlighting for `*.ram` files in PyCharm!



```
set integer var1 to 30 + (45 * (15 / 3)) - 2

display var1

loop with j from (245) to (var1) {
    loop with k from (1) to (2) {
        display j + k
    }
}

reset integer var1 to var1 - 3

if (var1) is (0) {
    display "The End"
} else if (var1) is (15) {
    display "Hello World!"
} else {
    display "The variable is:"
    display var1

    if (var1) is (250) {
        display "HOLY"
    } else {
        display "NAH"
    }
}
```

Packages installed successfully: Installed packages: 'Block' (today 8:25 p.m.)

– Then run the file using the command `ram <filepath>.ram` or just `ram`.

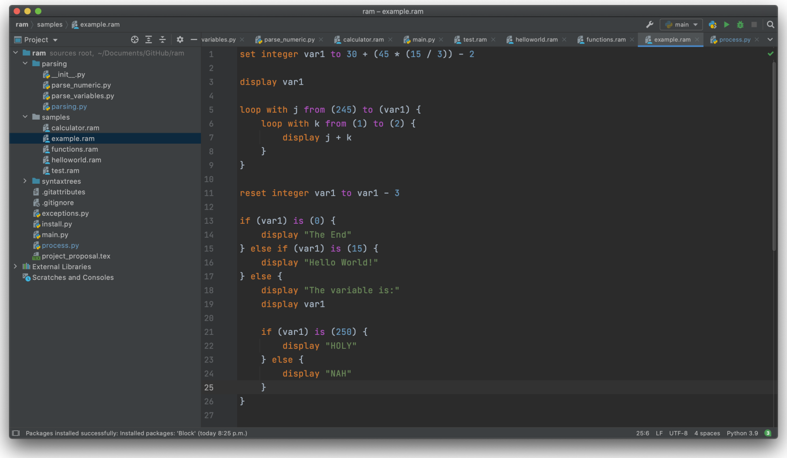


```
Last login: Fri Apr 16 21:21:54 on ttys000
willassad1@Williams-MBP ~ % cd documents/GitHub/ram/samples
willassad1@Williams-MBP samples % ram example.ram
253.0
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
The variable is:
250.0
HOLY
willassad1@Williams-MBP samples % 
```

### Using Windows

*Note:* only python console is supported in Windows.

- Using Python Console
  - You can write a new `.ram` file in any text editor. We even defined custom syntax highlighting for `*.ram` files in PyCharm!



```
set integer var1 to 30 + (45 * (15 / 3)) - 2

display var1

loop with j from (245) to (var1) {
    loop with k from (1) to (2) {
        display j + k
    }
}

reset integer var1 to var1 - 3

if (var1) is (0) {
    display "The End"
} else if (var1) is (15) {
    display "Hello World!"
} else {
    display "The variable is:"
    display var1

    if (var1) is (250) {
        display "HOLY"
    } else {
        display "NAH"
    }
}
```

  - Run main.py
  - You will be prompted to enter the file path of the `.ram` file to run.
  - The console runs the file!

### Technical Notes

- Strings must be defined with double quotes, not single quotes.
- Return statements, if any, must be the last line in a function.
- There cannot be spaces when calling functions (i.e. `f[x=40,y=some_variable]`).
- The ending curly brace of a loop, function, or if statement must be the only item in the line.

## Changes

The project proposal provided a clear outline of our final objective, and we had an explicit plan to arrive at the final result. Despite no changes to our final objectives, we had to alter our computational plan in certain areas. In particular, we found that the libraries we had initially planned on integrating, `ast` and `nltk`, did not serve the intended purpose. We actually ended up hard-coding all of the ASTs, using some of the basic expressions included in the *CSC111* Course Notes [1].

## Discussion

Upon testing numerous different sample files with many different variations of code — a combination of assignment statements, loops, if statements, and functions — we have discerned that our program is able to accurately take in the code as strings, and parse them to appropriate blocks.

Referring back to our initial project goal, we see that 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. That is, when users do not correctly follow syntax requirements, the appropriate errors are raised. In addition, we built an extremely interactive environment where the user can do an array of things all the way from simple arithmetic calculations to building entire algorithms. It is very interesting to see the intricate and extremely nuanced approach to building a language. The process of going from strings to blocks of code which then gets parsed and interpreted by the system was expectantly difficult as a large majority of our key functions had especially complicated and technically demanding recursive implementations.

The limitations we encountered largely had to do with the sheer complexity of many of the key function implementations, alongside the general project as a whole. In addition to that, we feel that we were very ambitious in the early stages of our exploration process which minimized the time we had to implement our actual plan. Given an increase in the time restriction, we would've liked to see what additional features we could have integrated into the program from things like comments to nested functions. Given the nature of this project, whenever we encountered an error, it was very difficult to trace the error and determine what the triggers were. As a result, our approach for error testing was much more time consuming and unreliable than we had hoped for.

To explore the content further, we may have tried to implement comments so that our users can potentially leave bits of information for their own reference. Alongside that, we could have tried to explore new libraries that could expand the range of possibilities. Lastly, we could have further investigated the possibilities of nested functions.

## 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.