LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF THEORETICAL COMPUTER SCIENCE AND THEOREM PROVING

An Interpreter for a Subset of Python in Rust

Jakob Franziskus Selmeier

Bachelor's Thesis
in Computer Science plus Computerlinguistics

Supervisor: Prof. Dr. Jasmin Blanchette

Advisor: Prof. Dr. Jasmin Blanchette

Submission Date: August 24, 2024

Disclaimer

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, August 24, 2024 Author

Abstract

This thesis presents the development of a tree-walk interpreter for a subset of Python implemented in Rust, using only the Rust standard library. A tree-walk interpreter is characterized by its direct interpretation of tree structures generated by the parser. This simplifies implementation by providing an easy execution model at the cost of performance when compared to compiled solutions. The primary objective is to create an interpreter that parses and executes a basic subset of Python while relying on Rust's inherent memory safety guarantees, exploring how Rust concepts and features can be helpful for language implementation.

Keywords: Python, Rust, Interpreter, Parser, Recursive Descent, Abstract Syntax Tree.

Contents

1 Introduction

The following thesis presents the implementation of a tree-walk interpreter for a subset of Python, programmed in Rust using only the standard library. The interpreter consists of three main parts: The scanner first reads the source code written by the user and turns it into tokens, which are then organized by a recursive descent parser into tree structures. The interpreter then traverses over these structures, executing the code in the process.

The key motivation for this project was to see how Rust concepts can be applied to programming other programming languages. Particularly Rust's inherent memory safety guarantees through its ownership model make it an interesting choice for implementing interpreters. Even without interacting with advanced ownership concepts, Rust helps in avoiding many errors that are common in other low-level programming languages, while at the same time not having to sacrifice performance for garbage collection.

This thesis has the following structure:

Firstly, Chapter 2 shortly introduces Python and its core features. It also introduces Rust, explaining every programming concept that will be used in the thesis.

Chapter 3 then focuses on the implemented subset of Python, showing the grammars used for the scanner and parser and every feature of the subset.

The main part of the thesis showcases how the interpreter is implemented. Chapter 4, 5 and 6 each focus on the individual parts of the entire interpreter: Chapter 4 explains how the scanner works. Chapter 5 demonstrates how the output of the scanner is converted into input for the interpreter through the parser. Finally, chapter 6 then shows how the structures generated by the parser are executed by the interpreter. Each of these chapters also present the data structures used and discuss which errors are handled. The input and output of every part is shown using an example Python program.

Afterwards chapter 7 shows related work, shortly covering three related projects and their differences to this thesis.

Finally, in the conclusion in chapter 8 features that have not been implemented in this project are explored, explaining shortly how one could go about adding them.

2 Background

2.1 Python

Python is a high-level programming language that is easy to learn, while remaining powerful (van Rossum 2024). In this section three core features of Python are presented, which distinguish it from other languages.

Firstly, Python's syntax is designed to be easy to read, almost resembling pseudo-code. As an example consider the following function definition, first in C then in Python:

```c
int triple(int x) {
    return x * 3;
}
```

```python
def triple(x):
    return x * 3
```

As you can see, Python does not need an explicit return type and parameter type, deferring the handling of possible type errors to runtime. Semicolons to mark end of lines and curly brackets to mark blocks are also omitted, instead relying on correct indentation and line breaks.

Another core feature of Python is dynamic typing, meaning a variable can contain values of different types during the execution of a program. This stands in contrast to static typing, where a variable can only contain values of the type it was declared as. This also means functions can return different types, depending on the type of the parameters they are called with. For example, the variable `a` here is first initialized as an integer, then a string is assigned to it, then a floating point number:

```python
a = triple(2)    # 6
a = triple("a")  # "aaa"
a = triple(-3.3) # -9.9
```

In a statically typed language like C, this would not be possible, since the function `triple` is defined to only take integers, and the variable `a` would need to be declared with a type.

Lastly, Python is generally considered an interpreted language. This means that a Python source code file, e.g. `triple.py` is not, like for example in C, compiled to an executable, which then gets executed. Instead the Python interpreter is called on the source file itself, processing and running the code without producing any intermediate files.

2.2 Rust

Rust is a systems programming language focusing on safety and performance. It does so through its ownership system, providing memory safety guarantees without the need for a garbage collector. Because of this, together with its high-level programming concepts and extensive error messages and tooling, Rust is designed to be a good choice for programming reliable and fast systems (The Rust Project Developers 2024a).

In this section, ownership is explained shortly and every Rust concept that is relevant to this project is presented.

2.2.1 Ownership

Ownership in Rust is a memory management system that prevents common memory issues like double frees, data races and dangling pointers by enforcing strict rules at compile time. This ensures programs are memory safe without the need for a garbage collector or extremely careful programming.

The basics of ownership are as follows: Each value in a Rust program must have an owner at any given time. There can only be a single owner of a value at a time. Whenever the owner goes out of scope, the value is deallocated, eliminating the need for manual memory management. Values can temporarily be borrowed using references/pointers, either immutably or mutably. For a more in-depth explanation of ownership and borrowing, see (Klabnik & Nichols 2019:4.1, 4.2).

In this project, ownership only played a background part. This is because the project didn't involve complex memory management or mutably shared states. Since the ownership system is designed to be intuitive, most rules are automatic and stay invisible unless necessary. If this project involved lower-level management of memory or aimed to be as fast as possible, ownership concepts would become more critical.

2.2.2 Functions and Data Structures

Consider the following example for a function definition from Klabnik & Nichols (2019):3.3:

```
fn plus_one(x: i32) → i32 {
    x + 1
}
```

This function has a parameter `x` of type `i32` (in the parentheses) and a return value of the same type (after the arrow). The expression `x + 1` is the value that is returned from the function. Notice that there is no semicolon after this line, making it an expression instead of a statement. You could also write this as a return statement, as follows: `return x + 1;`, in which case a semicolon is needed to mark it as a statement.

Structs are types that are composed of other types. The most commonly used type of struct has named fields (The Rust Project Developers 2024b:struct). They are defined as follows:

```
struct Player {
    name: String,
    x: u32,
    y: u32,
}
```

Enums are similar to enums in other languages, with the key difference that each variant of the enum can have data to go along with it (The Rust Project Developers 2024b:enum). As an example consider this enum:

```
enum Command {
    Quit,
    Move(i32, i32),
    NameChange(String),
}
```

Structs and enums are instantiated like this:

```
let player1 = Player {
    name: "Jakob".to_string(),
    x: 3,
    y: 1,
};

let cmd = Command::Move(1, -1);
```

Methods can be defined on structs and enums using an `impl` block (The Rust Project Developers 2024b:impl). For example, `exec_cmd` is defined as a method of `Player`:

```
impl Player {
    fn exec_cmd(&mut self, cmd: Command) {
        match cmd {
            ..., // other two enum variants
            Command::NameChange(n) => self.name = n,
        }
    }
}
```

This method takes a mutable reference to the struct it is defined on as a parameter, in order to be able to mutate the fields of the struct. It also uses a `match` statement, which is a pattern matching construct similar to a `switch` in other languages. This has to handle every pattern exhaustively (The Rust Project Developers 2024b:match).

An `impl` block can also be used to implement a trait on a data type (Klabnik & Nichols 2019:10.2). Traits are similar to interfaces in other languages, defining shared behaviour.

2.2.3 Option, Result and the Question Mark Operator

There are two special types in the standard library that are used a lot in Rust code:

Firstly, the `Option` type is defined as follows:

```
enum Option<T> {
    None,
    Some(T),
}
```

It is an enum defined on a generic type `T`, and can either be `None` or `Some(T)`, containing a value of type `T`.

`Option` is used to represent optional values and has a variety of use cases (see The Rust Project Developers (2024b):std::Option for most of them). Most of the time, they are used as return values for operations which are only partially defined or to report simple errors, where no error message is needed.

The `Result` type is defined similarly to `Option`:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

This enum takes two generic types, one for the value inside of the success case `Ok(T)` and one for the error case `Err(E)`.

`Result` is always used for error handling: A function, that could produce an error should always return a `Result`, which is then handled later, either by unpacking the value inside of `Ok` or reporting the error inside of `Err`.

Both `Option` and `Result` can both be handled in the same way with a match statement. Take the following code snippet from the parser of this project:

```
fn while_statement(...) →  Result<Stmt, Error> {
    // expression() returns a Result<Expr, Error>
    let cond = match expression() {
        Ok(expr) => expr,
        Err(err) => return Err(err),
    };
    ...
}
```

This function parses a while statement and returns a `Result` which contains either a statement or an error. In order to do so, it needs to parse the condition expression first by calling `expression()` and then handling the returned `Result` by either putting the expression inside of it into the `cond` variable or returning the error.

This approach works, but the concept of unpacking values from `Options` and `Results` is used very often and this is rather lengthy. The question mark operator is syntactic sugar, doing the same thing in a single line, as follows:

```
fn while_statement(...) →  Result<Stmt, Error> {
    // expression() returns a Result<Expr, Error>
    let cond = expression()?;
    ...
}
```

3 Python Subset

In this section, the grammars of the implemented subset of Python are presented. Additionally, all key features are shown together with example programs that the interpreter can execute.

3.1 Grammars

The lexical grammar represents all tokens accepted by the scanner (3.2.3) through the following regular expressions:

```
Keyword     → True | False
              | not | and | or
              | if | else | while
              | def | return
              | print
              | None

Identifier  → [a-zA-Z_][a-zA-Z0-9_]*

Integer     → [0-9]+
Float       → [0-9]+\.[0-9]+

String      → "([^"\\]|\\.)*"

Operator    → \+|\-|\*|\/
              |==|!=|<=|>=|<|>

Delimiter   → \(|\)|\[|\]
              |\,|\:|\=

Whitespace  → [ \t]+

EndOfLine   → \n

Comment     → \#.*
```

The tokens are then arranged by the parser (4.3) into abstract syntax trees according to the following grammar:

```
file           → stmt* EOF

stmt           → exprStmt
                | printStmt
                | assignStmt
                | ifStmt
                | whileStmt
                | funDecl
                | returnStmt

exprStmt       → expr "\n"
printStmt      → "print" "(" expr ")" "\n"
assignStmt     → IDENTIFIER "=" expr "\n"
ifStmt         → "if" expr ":" block ("else" ":" block)?
whileStmt      → "while" expr ":" block
funDecl        → "def" IDENTIFIER "(" parameters? ")" ":" block
returnStmt     → "return" expr?

block          → "\n" INDENT stmt* DEDENT
parameters     → IDENTIFIER ("," IDENTIFIER)*

expr           → disjunction
disjunction    → conjunction ("or" conjuction)*
conjunction    → equality ("and" equality)*
equality       → comparison (("==" | "!=") comparison)*
comparison     → term ((">" | ">=" | "<" | "<=") term)*
term           → factor (("+" | "-") factor)*
factor         → unary (("*" | "/") unary)*
unary          → ("-" | "not") unary
                | primary
primary        → NUMBER | STRING
                | "True" | "False" | "None"
                | "[" arguments? "]"
                | "(" expr ")"
                | IDENTIFIER ("(" arguments? ")" | "[" expr "]")?

arguments      → expr ("," expr)*
```

3.2 Features

3.2.1 Data Types and Dynamic Typing

The subset supports the following data types:

- Integer, e.g. `5`, `0`, `-10000`
- Floating Point, e.g. `2.9`, `-18.0508`
- String, e.g. `"abc"`, `""`, `"Hello World"`
- Boolean, e.g. `True`, `False`, `not (1 == 1 and 2 != 3)`
- List, e.g. `[1,2,3]`, `[]`, `[-1, 1.1, "a", True, [1]]`
- `None`

Variables can be assigned like in Python and dynamic typing is supported, meaning the same variable can contain different data types:

```
a = 3
a = "hello"
a = [1, "a"]
```

All the operators (see lexical grammar) are implemented for every type as you would expect. The addition operator ”+” can be used to concatenate strings and lists.

3.2.2 Control Flow

If clauses are written like this and can have an optional else clause:

```
x = True
if x:
    print("hello")
else:
    print("bye")
```

Basic while loops are also supported:

```
i = 5
while i >= 0:
    print(i)
    i = i - 1
```

3.2.3 Functions

Functions can have a return value and are defined as follows:

```
def power(x, n):
    res = 1
    while n > 0:
        res = res * x
        n = n - 1
    return res
```

After a function is defined, it can be called from anywhere:

```
def powers(x, n):
    i = 0
    while i <= n:
        print(power(x, i))
        i = i + 1

powers(2, 10)
```

Functions can also be recursive:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

4 Scanner

The first part of an interpreter needs to step through the source code provided by the user and convert it into meaningful chunks, called tokens. This happens in the scanner or lexer. During this process, it discards meaningless parts of the source code like whitespace and generates errors upon encountering unexpected characters.

4.1 Token Representation

Tokens are constructed by the scanner and then returned to the main function inside of a list. They are represented by the following data structure:

```
struct Token {
    token_type: TokenType,
    value: String,
    line: u64,
    column: u64,
}
```

The `line` and `column` fields are used to carry information about the location of the token through to later parts of the program, where they are then used in case of error messages.

`value` contains the raw text which represents the token, e.g. a `True` token has the string ”True” as its value. The scanner uses this field to update its index.

The type of the token is stored inside of `token_type` and can be one of the following members of this enum, which is derived from the lexical grammar:

```
enum TokenType {
    True,
    ...
    None,
    Identifier(String),
    Int(u64),
    Float(f64),
    String(String),
    Plus,
    ...
    LessEqual,
    LeftParen,
    ...
    Equal,
    EndOfLine,
    Indent,
    Dedent,
    EndOfFile,
}
```

4.2 Implementation

The scanner receives the source code as a `String` from the main function. It then iterates over the code using an index counter, repeatedly calling a `match` statement which matches on known characters and builds them into tokens according to the lexical grammar. After a token is built, it is added to the list returned by the scanner. Once the scanner finishes iterating over the entire `String`, it generates an `EndOfFile` token and returns a value of type `Option<Vec<Token>>`.

Before a newly built token is added to the returned list, the index is updated using the amount of characters of the `value` field of `Token`. Every ignored character moves the index by one.

The column counter is updated in the same way as the index counter with one difference: since Python does not use semicolons to mark the end of statements, the scanner generates an `EndOfLine` token whenever it encounters a newline character. This then causes the column counter to be reset and the line counter to go up by one.

4.2.1 Indentation

In Python, instead of curly brackets, the `Indent` and `Dedent` tokens are used to mark the beginning and end of code blocks. These tokens are generated in the scanner using the method described in the *Python Language Reference* (van Rossum & Drake Jr 2024:2.1.8):

At the start of scanning a stack containing the indentation levels is created and the value zero is pushed onto the stack. Then, every time a new line begins, its indentation is calculated by counting spaces and tabs, with tabs counting as however many spaces would be needed until the next multiple of 8. During this process the index counter is moved accordingly, until it points to a non-whitespace character. In case it is a newline character or a comment symbol the indentation is ignored and the scanner moves on to the next line. Otherwise, the calculated indentation is compared to the indentation level currently on top of the stack. If it is larger, the calculated indentation is pushed onto the stack and one `Indent` token is generated. If it is equal, nothing happens. If it is smaller, the scanner continues popping numbers off the stack and generating a `Dedent` token for every number popped off, until the current line's indentation matches one on the stack. Should there be no more numbers left, the scanner throws an error. At the end of the file, for every number left on the stack that is greater than zero, a `Dedent` token is generated before the `EndOfFile` token.

To illustrate this method consider the following incorrectly indented Python code:

```
1 def double(x):
2 _[tab]res = x * 2
3 __return res
```

Before beginning the scanning process, the indentation stack is initialized as [0]. After scanning the first line, the scanner calculates the indentation of the second line. In this case it sees a space followed by a tab character, which together is evaluated to an indentation level of 8. This is now put onto the stack and an `Indent` is generated, because 8 is larger than the current indentation of 0. The stack now looks as follows: [0,8].

After scanning the rest of the second line, the scanner starts the same process for line 3, for which it calculates an indentation level of 2. Since 2 is smaller than the 8 on top of the stack, the scanner generates a `Dedent` and pops the 8 off the stack. Now the scanner continues popping numbers off the stack until one of them is equal to 2, generating a `Dedent` for each one. After the 0 is removed, there are no more numbers on the stack, causing an "inconsistent dedent" error.

4.2.2 Error Handling

The project uses the following simple `struct` to report errors:

```
struct PyError {
    msg: String,
    line: u64,
    column: u64,
}
```

`PyError` implements the `Display` trait (The Rust Project Developers 2024b:std::fmt::Display) in order to be printable to the user. It formats an error message as follows:

```
[msg]
    Line [line], Column [column]
```

For example:

```
SyntaxError: Unknown character
    Line 3, Column 14
```

The scanner handles several syntax errors that can occur, e.g. unkown characters, unterminated strings or invalid decimal literals. When it encounters an error while scanning characters, instead of generating a token, it generates an error that is then printed out to the user as shown above. It then moves the index and column by one, continuing execution at the next character, and sets an error flag, signalling an error somewhere during the scanning process. This makes it possible to report multiple of such errors in a single run.

However, if there was an error while checking for correct indentation, instead of continuing at the next character, the scanner will stop execution. This is done because the indentation stack is no longer correct after unsuccessfully checking for a matching indentation. Because of this, following lines which are technically correctly indented can potentially cause an error.

After finishing, if the error flag is set, instead of the generated list of tokens it returns `None`. This then causes the interpreter to stop running, preventing errors from cascading into later stages of the execution.

4.3 Example Program

To illustrate the functionality of the scanner, consider the following example of a program defining a `fib` function, which calculates the nth Fibonacci number, and printing the result of the function with the argument 10.

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

print(fib(10))
```

As a general example for a line of code, these are the tokens that are generated by the scanner while reading the first line and put into the token list:

Type	Line	Column
Def	1	1
Identifier(”fib”)	1	5
LeftParen	1	8
Identifier(”n”)	1	9
RightParen	1	10
Colon	1	11
EndOfLine	1	12

After the `EndOfLine` token is generated, the scanner calculates the next line’s indentation, which in this case is larger than the previous indentation. Because of this, the indentation level is pushed onto the stack and an `Indent` token is constructed before the rest of the line is scanned. When the third line begins, the scanner repeats this same process. The indentation on the fourth line is smaller than previously, generating one `Dedent` token. For the second `if` clause, the scanner again generates one `Indent` and one `Dedent`. Finally, before scanning line eight, another `Dedent` is created. If the program ended after the sixth line, this `Dedent` would still be generated.

For the entire program, the following indentation tokens are generated:

Type	Line	Column
Indent	2	5
Indent	3	9
Dedent	4	5
Indent	5	9
Dedent	6	5
Dedent	8	1

5 Parser

The tokens generated by the scanner need to be organized into an abstract syntax tree (AST), which can then be executed by the interpreter. This is done via a recursive descent parser which traverses the token list and builds the AST according to the rules defined in the grammar.

5.1 AST Data Structures

The parser returns a list of statements, which can then be executed by the interpreter. A statement can be any one of the members of the following enum, mirroring the stmt rule in the grammar:

```
enum Stmt {
    Expr(Expr),
    Print(Expr),
    Assign(Name, Expr),
    If(Expr, Vec<Stmt>, Option<Vec<Stmt>>),
    While(Expr, Vec<Stmt>),
    FunDecl(Name, Vec<Name>, Vec<Stmt>),
    Return(Location, Option<Expr>),
}
```

All statements contain all the information necessary for the interpreter to execute the code and return errors with the correct location:

- Expression and Print statements only contain an expression. Expressions evaluate to a value in the interpreter and are represented by the enum below.
- Assignment statements also contain a `Name` struct in addition to the expression which is being assigned. This is a simple struct which has a `String` field for the name and a `Location` field for a `Location` struct, containing line and column information.
- `if` statements have a condition expression, a list of statements for the `if` case and an optional list of statements, if there is an `else` case. `while` statements are the same as `if` statements, only without the optional `else` list.
- Function declaration statements contain a Name, a list of names for the parameters and a list of statements for the body of the function.
- Return statements have a `Location` struct and an optional expression, if a specific value is returned.

This is the enum defining an expression:

```
enum Expr {
    Unary(UnOp, Box<Expr>),
    Binary(Box<Expr>, BiOp, Box<Expr>),
    Grouping(Box<Expr>),
    Literal(Lit),
    Variable(Name),
    Call(Name, Vec<Expr>),
    ListAccess(Name, Box<Expr>),
}
```

- Unary and Binary expressions contain their respective operators and the expressions these operators are applied to. `UnOp` and `BiOp` are structs containing their location and the type of operator, e.g. for unary operators `Not` and `Minus` and for binary operators `Plus`, `And`, `NotEqual` and so on.
- Grouping expressions are used for expressions inside of parantheses.
- Literals contain a `Lit` enum member, which can be any of the supported literal types.
- Function calls and list accesses have the `Name` of the function or list. Calls also contain a list of expressions, representing the values passed to the parameters of the function. List accesses only have a single expression, the index.

5.2 Implementation

Since this is a recursive descent parser, every rule in the grammar is translated into a function in the code (Aho et al. 2006:4). This method is top-down, thus it starts parsing at the outermost rule:

$$file \rightarrow stmt* \text{ EOF}$$

Until the parser reaches the `EndOfFile` token, it calls the function corresponding to the statement grammar rule. This function looks at the next token and, depending on the type, then calls the function for the specific type of statement. For example, if the next token has the type `While`, the function for the "whileStmt" rule is called.

$$whileStmt \rightarrow \text{"while"} \; expr \; \text{":"} \; block$$

The function in Rust looks like this:

```
fn while_statement(&mut self) -> Result<Stmt, PyError> {
    let cond = self.expression()?;
    self.check_or_error(vec![TokenType::Colon],
            "SyntaxError: missing colon after while statement")?;
    let block = self.block()?;
    Ok(Stmt::While(cond, block))
}
```

In line 2, the function matching the expression rule is called to parse the condition of the `while` loop. The question mark operator at the end of the line either returns an error that occurred while parsing the expression from this function or, if no error occurred, simply returns the parsed expression to the `cond` variable.

Line 3 calls the `check_or_error` function of the parser, which checks if the next token in the list which it received from the scanner is of the given type, in this case a colon. If it is not, the function returns with the given error message, else it continues parsing the statement.

Then the body of the `while` loop is parsed using the "block" rule:

```
block → "\n" INDENT stmt* DEDENT
```

After checking for an `EndOfLine` and an `Indent` token, the corresponding function parses statements until it reaches a `Dedent` token. All parsed statements are then returned in a list.

At the end of the function, a `While` statement of type `Stmt::While` is returned, with the parsed expression as the condition and the parsed block as the body. This is put into the list of statements that will be returned from the parser and, if there is no `EndOfFile` token afterwards, the next statement is parsed in the same way.

5.2.1 Parsing Expressions

While parsing expressions, it is important to consider the precedence of different operators and the associativity of operators with equal precedence.

In a top-down parser, the bottom-most rule is the rule with the highest precedence. Thus, elements in the "primary" rule have the highest precedence and will be evaluated first, while disjunctions have the lowest precedence in expressions and will be evaluated last.

```
expr        → disjunction
disjunction → conjunction ("or" conjuction)*
...
primary     → NUMBER | STRING
            | ...
```

Because of the way the rules are built, operators with the same precedence are always left-associative, e.g. `1 + 2 + 3` is evaluated as `(1 + 2) + 3`.

5.2.2 Error Handling

Errors in the parser are either syntax errors in case of missing or unexpected tokens, e.g. missing parentheses in a function call, or indentation errors if it runs into any unexpected `Indent` or `Dedent` tokens.

If an error occurred while parsing, the parser immediately stops execution in order to prevent cascading errors. In some cases, e.g. missing closing parentheses in a `print` statement, the parser could report the error and then move on to the next line, since a `print` statement is always contained to a single line. But consider this `if` statement with a missing colon:

```
1 if a == 1
2     print("a is equal to 1")
3 else:
4     print("a is not equal to 1")
```

While parsing the statement the parser expects a `Colon` token after parsing the condition expression, but instead finds an `EndOfLine` token. This causes an error and the parser stops parsing the `if` statement and discards it. If instead of stopping here it simply moved on to the next line, the following would happen: The first token in line 2 is of type `Indent`, which is now unexpected, causing an indentation error to be reported, even though the indentation here is technically correct. This then also happens in the next two lines, because the parser is not expecting an `else` block, since it never parsed an `if` block. Thus, the user sees several errors, all of which are only a result of the missing colon and do not indicate any real errors in the code.

5.3 Example Program

To show the AST structures the parser generates, consider the same Fibonacci program from before:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

print(fib(10))
```

The parser generates a list with the following two statements:

```
FunDecl(fib, [n],
        [
            If((== n 0), [Return({line: 3, column: 9}, 0)], None),
            If((== n 1), [Return({line: 5, column: 9}, 1)], None),
            Return({line: 6, column: 5},
                   (+ fib([(- n 1)]) fib([(- n 2)])))
        ]),
Print(Call(fib, [Literal(10)]))
```

All fields containing a name, e.g. `fib`, also contain a location, which is left out here.

The first statement is of type `Stmt::FunDecl` and contains the function's name `fib`, a list of parameters in this case only `n` and a list of statements, which comprise the body of the function. In this list there are two `Stmt::If` statements with their respective conditions and bodies. If one of the statements had an `else` case, there would be another list of statements at the end instead of `None`. The third statement in the function body is of type `Stmt::Return` and contains its location together with the expression being returned.

Expressions here are displayed in prefix notation for readability. The actual data structure for e.g. the returned expression at the end of the function is of type `Expr::Binary` and looks like this:

```
Binary(
    Call(fib, [Binary(Variable(n), Minus, Literal(1))]),
    Plus,
    Call(fib, [Binary(Variable(n), Minus, Literal(2))]),
)
```

6 Interpreter

In the final step, expressions are evaluated to values and statements are executed. During this, the interpreter also needs to handle runtime errors and variable scope.

6.1 Environments and Scope

To keep track of all functions and variables that are defined, the interpreter saves them in hash maps stored in fields of the following structure:

```
struct Environment {
    enclosed_by: Option<Box<Environment>>,
    funcs: HashMap<String, Function>,
    vars: HashMap<String, Value>,
}
```

There are two `assign` and two `get` methods implemented on this struct, one for each of the two hash maps.

To implement scope, the `enclosed_by` field optionally contains another environment. The highest level environment is not enclosed by any other. All functions have their own environments, which are enclosed by the ones in which they are called. When looking for a variable or function, the `get` methods first look for it inside of the current environment and, if the searched name could not be found, then call themselves on the one inside the `enclosed_by` field. If the field is `None`, an error is thrown.

6.2 Evaluating Expressions

Since Python is dynamically typed, meaning a variable can hold any possible data type at runtime, the interpreter needs a data structure that can be any of these types (Nystrom 2021:7). Because Rust does not have a similar construct to Java's `Object` type, the following enum is used:

```
enum Value {
    Int(i128),
    Float(f64),
    String(String),
    Bool(bool),
    List(Vec<Value>),
    None,
}
```

Every expression generated by the parser is evaluated to a `Value`. Every enum member contains its respective Rust version. Lists have a vector of `Values`, making it possible for them to be heterogeneous. A `to_bool` method is implemented on the enum, which returns the truth value of the value it is called on: `false` for zeros, empty strings and lists and `None` and `true` for everything else.

Expressions are evaluated using the `eval_expr` function, which then calls the corresponding function depending on the type of expression.

`Unary` and `Binary` expressions are evaluated by calling the respective functions on their parts and applying the operators on the results.

A `Grouping` simply evaluates the expression inside of it.

`Literal`s already are single values, thus they are converted to the respective `Value`.

As an example for basic arithmetic expressions consider the following expression returned by the parser:

```
// represents (6 + 3) * 4
Binary(
    Grouping(Binary(Literal(6), Plus, Literal(3))),
    Times,
    Literal(4)
)
```

The interpreter calls the function for evaluating binary expressions `eval_binary`. This then evaluates the left-hand side first, calling `eval_expr` on it, which in turn calls `eval_binary` again, this time on the binary expression inside of the grouping. Here, the left-hand side is evaluated to `Value::Int(6)` and the right-hand side to `Value::Int(3)` by `eval_lit`, before the addition operator is applied, returning `Value::Int(9)`. Then the right-hand side of the outer binary expression is evaluated to `Value::Int(4)`, before the two sides are multiplied and the resulting `Value::Int(36)` is returned from the function.

For `Variable`, the `get_var` method of `Environment` is called to search for the names in the `vars` hash map and then the associated value is returned.

`List Accesses` first evaluate the index expression to a value, then check if this value is an integer. Afterwards, if the found variable is a list, the value at the index is returned.

`Call` expressions are evaluated by first looking for the function in the `Environment` field and checking if the argument count is correct and then using the `call` method of Function. This method first creates a new `Environment`, which is enclosed by the one in which the function is called. Then it adds the parameter names together with the argument values as new variables to this environment before running the function by calling the statements in its body. The expression is then evaluated to the value returned by the method or to `None` if there was no return value.

6.3 Executing Statements

The main loop of the interpreter and the `call` method of functions repeatedly call the function `interpret_stmt`, which executes code depending on the type of statement passed to it.

For `Expr` statements it simply evaluates the expression. `Print` functions the same way, only it prints the expression afterwards.

`Assign` statements use the `assign_var` method of `Environment` to add a new variable to the `vars` hash map, containing the name and value calculated from the expression. If a variable already exists in the map, the value is simply updated. This only checks if the variable exists in the current scope and not if it exists in any outer one; thus you cannot change any variable from within a function that was first assigned from somewhere outside this function, instead a new variable in the current scope will be created.

Executing `If` statements is done by first calculating the value of the expression and then checking its truth value. Then, if this evaluates to `true`, the first list of statements is executed.

Otherwise, the interpreter checks if there is a list of statements for the else case and, if there is, executes those instead.

For `While` statements the interpreter continually evaluates the condition expression, executing the list of statements once if it is true and breaking from the loop otherwise.

When encountering a `FunDecl` statement, the `interpret_stmt` function calls the `assign_func` method of Environment. This method creates a new function in the hash map. Functions are represented through this structure:

```
struct Function {
    name: Name,
    parameters: Vec<Name>,
    body: Vec<Stmt>,
}
```

The fields of this `struct` are the same as the fields in the enum member for function declaration statements. Functions can be called via the `call` method which is implemented on this struct. This method builds a new environment and then interprets the statements inside the function body.

`Return` statements have two differences compared to other statements: they return a value from the `interpret_stmt` function and they are not allowed to occur outside of function bodies. To ensure this is the case, the interpreter only accepts a returned value from `interpret_stmt` during the `call` method of `Function`, causing the method to stop interpreting the function body and returning the value. If a value is returned from `interpret_stmt` during the main loop an error is thrown.

6.4 Example Program

To see how the interpreter executes statements and works with environments, consider the Fibonacci program again:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

print(fib(10))
```

The statements generated by the parser were the following:

```
FunDecl(fib, [n],
        [
            If((== n 0), [Return({line: 3, column: 9}, 0)], None),
            If((== n 1), [Return({line: 5, column: 9}, 1)], None),
            Return({line: 6, column: 5},
                   (+ fib([(- n 1)]) fib([(- n 2)])))
        ]),
Print(Call(fib, [10]))
```

Before the interpreter starts interpreting, the main or global environment is created. It is enclosed by no other environment and both the `funcs` and `vars` hash maps are empty.

Interpreting then starts at the first statement, calling `interpret_stmt` on it. Since it is a function declaration, the `assign_func` method of the environment is called, adding the ”fib” as the key and the following `Function` to the `funcs` map:

```
Function {
    name: fib,
    parameters: [n],
    body: [If, If, Return]
}
```

Statements in the body are shortened here for readability, but are the same three from the `FunDecl` body.

Afterwards, the print statement is interpreted by evaluating the expression inside of it. Since this is a function call, the `eval_expr` function calls the `call` method of the `Function` struct. This creates a new environment, which is enclosed by the global one. In its `vars` map, a new variable is created with the key ”n” (from the function declaration statement) and the value 10 (from the function call). Then the statements in the body of the ”fib” function are executed. Since `n` is 10, both if conditions are evaluated to false, and since there is no else branch in both cases, the statements do not do anything. When the return statement is executed, the interpreter evaluates the expression, first evaluating the call on the left-hand side then the one on the right-hand side. Once the expression was evaluated, in this case to `Value::Int(55)`, the value is returned from the function call to the print statement, which then prints it out to the user.

7 Related Work

The following section will present three projects that relate to this thesis. They will shortly be introduced and their main differences to this thesis will be highlighted.

7.1 RustPython

RustPython (RustPython Team 2024) is an interpreter for Python written in Rust. Its goal is to create a complete Python environment with a fast, clean and secure implementation, which can be embedded into Rust programs or compiled to WebAssembly.

The main differences between RustPython and the Python interpreter from this project are the following:

- RustPython aims to support the entire Python 3 specification, including libraries, instead of only a basic subset.
- Python source code is compiled to bytecode which is then interpreted by a virtual machine instead of using a tree-walk interpreter, which should make RustPython more performant.
- RustPython offers advanced features like Just-In-Time compilation.

7.2 Nederlang Tree-Walk Interpreter in Rust

To practice Rust and explore its memory safety capabilities compared to C, Danny van Kooten wrote a tree-walk interpreter for a toy language called Nederlang (van Kooten 2022).

The main difference when compared to the project in this thesis is that the Nederlang interpreter was optimized to be as fast as possible. Some optimization techniques from the blog post that could be leveraged for this project are:

- Rewriting `Environment` to use vectors instead of a hash map, making variable lookup faster in the process
- Using reference nodes in the AST instead of cloning statements and expressions when needed
- Using string references for variable names instead of owned Strings to avoid cloning

7.3 Nuitka

Nuitka (Hayen 2024) is a compiler written in Python that translates Python code to C code and then produces a standalone executable. In the process it optimizes the code to avoid overhead. This stands in contrast to a tree-walk interpreter which directly interprets the code instead of converting it to another language first.

8 Conclusion

This thesis demonstrated the viability of implementing languages in Rust. Rust makes for an excellent choice when building interpreters, because of its robust data structures, modern language features and inherent memory safety. Another feature is the expansive Rust ecosystem, which was not shown in this thesis, since the interpreter only used the Rust standard library.

For future work on this interpreter more features could be implemented:

- A standard library with built-in Python functions, e.g. len(), input(), etc.
- Object orientation with classes (Nystrom 2021:12)
- Performance improvements (see 7.1)
- More detailed error messages using a more complex error type

When implementing a new language feature, the usual way to do so is the following:

1. Add the necessary tokens to the lexical grammar, token data structures and the scanner
2. Add the rules to the grammar, expand the AST data structures accordingly and implement the rules as functions in the parser
3. Consider how the interpreter should execute the tree it receives, i.e. what does the environment need to contain, are new data structures needed or can it be represented by existing ones

References

AHO, ALFRED V.; MONICA S. LAM; RAVI SETHI; and JEFFREY D. ULLMAN. 2006. *Compilers: Principles, techniques, and tools*. 2nd Ed. Addison-Wesley.

HAYEN, KAY. 2024. Nuitka: Python compiler. Accessed: 2024-08-15. URL `https://nuitka.net/`.

KLABNIK, STEVE, and CAROL NICHOLS. 2019. *The rust programming language*. 2nd Ed. San Francisco, CA: No Starch Press. Available online at `https://doc.rust-lang.org/book/`.

VAN KOOTEN, DANNY. 2022. Rewriting a simple interpreter in rust. Accessed: 2024-08-10. URL `https://www.dannyvankooten.com/blog/2022/rewriting-interpreter-rust/`.

NYSTROM, ROBERT. 2021. *Crafting interpreters*. Genever Benning. Available online at `https://craftinginterpreters.com/`.

VAN ROSSUM, GUIDO. 2024. *Python tutorial, version 3.12.5*. Python Software Foundation. URL `https://docs.python.org/3/tutorial/`.

VAN ROSSUM, GUIDO, and FRED L. DRAKE JR. 2024. *Python language reference, version 3.12.5*. Python Software Foundation. URL `https://docs.python.org/3/reference/`.

RUSTPYTHON TEAM. 2024. RustPython: A Python Interpreter written in Rust. `https://rustpython.github.io`. Accessed: 2024-08-14.

THE RUST PROJECT DEVELOPERS. 2024a. Rust programming language official website. Accessed: 2024-08-10. URL `https://www.rust-lang.org/`.

THE RUST PROJECT DEVELOPERS. 2024b. *The rust standard library documentation*. The Rust Project. URL `https://doc.rust-lang.org/std`.