Interoperability between Python and Scheme Syntax using AST Manipulation

Principles of Programming Languages

Mohammad Zaid
Roll Number : 2021101004

1 Introduction

In this project, we aim to enhance interoperability between Python and a Scheme-like language by dynamically converting those expressions into Python Abstract Syntax Trees (ASTs). The motivation behind this project stems from the need for seamless integration of different programming paradigms and languages. By improving interoperability, we can leverage the unique strengths of functional programming in Scheme-like syntax and Python's versatility.

The core of this work involves reading code in this custom syntax, converting it into Python ASTs, and allowing these converted expressions to be executed within Python. This approach makes it possible to mix functional programming constructs with Python's procedural and object-oriented paradigms.

2 Project Objective

The primary objective of this project is to enable the dynamic linking of files or modules written in a Scheme-like syntax within Python, allowing such code to be executed seamlessly in a Python environment. By creating a mechanism to convert Scheme-Like syntax into Python ASTs, we aim to achieve true interoperability between the two languages. The key goals of this project include:

- **Dynamic Conversion and Execution:** Dynamically convert Scheme-like syntax into Python ASTs so that it can be executed within Python scripts.
- **Interoperability:** Allow functions and modules written in this custom syntax to be used as easily as native Python functions, enabling multi-language integration in real-time.
- **Flexibility in Programming:** Combine the functional programming capabilities of Scheme-like syntax with Python's procedural and object-oriented features.

To demonstrate interoperability, examples are included in subsequent sections. The syntax supported in this project is formally defined below.

3 Supported Grammar for Scheme-like Syntax

The Scheme-like syntax supported in this project is defined formally as follows:

- **Program:**

$$\text{Program} :: -\text{Expression}$$

- **Expressions:**

$$\begin{aligned}\text{Expression} &:: -\text{Number}\\ &:: -\text{Identifier}\\ &:: -(\text{operator Expression Expression})\\ &:: -(\text{assume ((Identifier Expression) ...) Expression})\\ &:: -(\text{proc Identifier Identifier Expression})\\ &:: -(\text{if Expression Expression Expression})\end{aligned}$$

- **Numbers:**

$$\text{Number} :: -\text{a numeric literal (e.g., 5, 3.14)}$$

- **Operators:**

$$\text{operator} :: + \mid - \mid * \mid / \mid < \mid > \mid =$$

- **Bindings:** (assume ((Identifier Expression) ...) Expression) ::- Binds variables to values and evaluates the Expression in the modified scope.
- **Procedures:** ::- (proc Identifier Identifier Expression) Defines a procedure with a name, parameter, and body.
- **Conditionals:** (if Expression Expression Expression) ::- Evaluates the second Expression if the first is true; otherwise, evaluates the third Expression.

4 Examples of Tokenization

Below are examples illustrating the tokenization of Scheme-like syntax into AST representations as list.

Example 1: Simple Addition

Input Code:

```
(+ 5 3)
```

Tokenized Output:

```
['+', 5, 3]
```

Example 2: Nested Expressions

Input Code:

```
(+ (* 2 3) (/ 10 5))
```

Tokenized Output:

```
['+', ['*', 2, 3], ['/', 10, 5]]
```

Example 3: Variable Binding

Input Code:

```
(assume ((x 5) (y 10)) (+ x y))
```

Tokenized Output:

```
['assume', [['x', 5], ['y', 10]], ['+', 'x', 'y']]
```

Example 4: Procedure Definition

Input Code:

```
(proc my_proc x (* x 2))
```

Tokenized Output:

```
['proc', 'my_proc', 'x', ['*', 'x', 2]]
```

Example 5: Conditional Expressions

Input Code:

```
(if (> 5 3) 10 20)
```

Tokenized Output:

```
['if', ['>', 5, 3], 10, 20]
```

These examples demonstrate the grammar rules and their corresponding tokenized representations, which serve as input to the Python AST generation process.

We aim to achieve interoperability as shown through examples in the next section.

5 Examples of Dynamically Linking Scheme-Like Code in Python

Example 1: Including a Scheme-Like Function in Python

Scheme-Like Code (in a file `math_operations.rkt`):

```
(proc (x) (* x x))
```

Python Code:

```
x = 10
y = racket_insert("math_operations.rkt")
result = y(x)
print(result)  # Output: 100
```

In this example, the Scheme-Like function `square` is dynamically converted into a Python function. The function is used within Python to calculate the square of `x`.

Example 2: Using Environment Bindings

Scheme-Like Code (in a file `binding_example.rkt`):

```
(assume ((a 5) (b 10)) (+ a b))
```

Python Code:

```
result = racket_insert("binding_example.rkt")
print(result)  # Output: 15
```

In this example, Variables `a` and `b` are dynamically bound within the environment.Their sum is calculated and returned as the result.

Example 3: Nested Functions with Conditional Logic

Scheme-Like Code (in a file `nested_logic.rkt`):

```
(proc (n m)
  (if (< n m)
      (+ n m)
      (- n m))
)
```

Python Code:

```
calculate = racket_insert("nested_logic.rkt")
result_1 = calculate(3, 5)  # Output: 8, because 3 < 5
result_2 = calculate(10, 4)  # Output: 6, because 10 > 4
print(result_1, result_2)
```

In this example calculate function binds to the lambda function defined in the file nested_logic.rkt .

Example 4: Environment with Multiple Bindings and Arithmetic

Scheme-Like Code (in a file `arithmetic_env.rkt`):

```
(assume ((x 3) (z 4)) (+ x y))
```

Python Code:

```python
x = 10
y = 15
result = racket_insert("arithmetic_env.rkt")
z = x * result
print(result)  # Output: 180
```

In this example,the `assume` construct binds `x = 3` and `z = 4`. These bindings are local to the Scheme-like code. The variable `y` is not defined in the Scheme-like code, so it refers to Python's globally defined `y = 15`. The Scheme-like code computes `(+ x y)` using its local `x = 3` and Python's global `y = 15`, resulting in `3 + 15 = 18`. This result is returned to Python as `result`.

Example 5

Scheme-Like Code (in a file `polynomial_example.rkt`):

```
(proc (a b c x) (+ (* a (* x x)) ( + (* b x) c)))
```

Python Code:

```python

funcc = racket_insert("polynomial_example.rkt")

# Manipulate the polynomial mathematically
scale_factor = 2  # Scale the polynomial's coefficients
shift_value = 3  # Shift the polynomial vertically

# Redefine polynomial with scaled coefficients
def transformed_quadratic(a, b, c, x):
    global funcc
    global shift_value
    return scale_factor * funcc(a, b, c, x) + shift_value

x_val = 4
result = transformed_quadratic(1, 2, 3, x_val)
print(result)  # Output : 2703
```

6 Implementation Details

The project focuses on key aspects of AST manipulation, including AST generation, AST manipulation, function binding, and environment evaluation. Below, we discuss these components in detail.

1. AST Generation

Abstract Syntax Trees (ASTs) are fundamental in representing code structures in a language-agnostic manner. We started by building a Python-based AST representation for Scheme-Like code. This involved creating Python classes to represent different types of AST nodes such as:

- **UnaryOp:** Represents unary operations (e.g., negation).
- **BinOp:** Represents binary operations (e.g., addition, multiplication).
- **Ifte:** Represents conditional expressions (if-then-else).
- **Num, Bool, IdRef:** Represent numerical values, boolean values, and identifier references respectively.
- **Proc, Assume:** Represent functions (proc) and environment bindings (assume).

The classes we created mirror Scheme-Like expressions, providing a unified representation of Scheme-Like code within Python. This was crucial to enabling conversion from Scheme-Like code into Python executable code.

2. AST Manipulation

Once ASTs were generated, the next step was to manipulate these ASTs to perform operations or transform the tree structure. The key part of this project was implementing the `racket_ast_to_python_ast()` function, which takes a Scheme-Like AST and converts it to an equivalent Python AST representation.

- **Binary and Unary Operations:** Converting Scheme-style operations (e.g., +, -, *, /, <,>,==) into Python AST representations using Python's `ast` module.
- **Conditional Expressions:** Transforming Scheme's `if` expressions into Python's equivalent `IfExp` AST representation.
- **Procedure Definitions:** Translating Scheme-Like procedures (`proc`) into Python function definitions (`FunctionDef`) or lambda expressions (`Lambda`).

These transformations enable Python to understand and execute Scheme-Like code.

3. Function Binding

One of the most significant features implemented was the ability to define and bind functions in Scheme-Like syntax within Python code. The `proc` keyword in our Scheme-Like code allows us to define a function, and we translated this into Python using `FunctionDef` or `Lambda` nodes.

The use of lambda expressions is particularly useful for representing anonymous functions, which are common in functional programming languages like Racket. The conversion ensures that these functions can be dynamically bound and called in Python, maintaining the original semantics of Scheme-Like code.

4. Environment Evaluation and Dynamic Code Insertion

We developed the `RacketTransformer` class to handle dynamic code insertion. This class replaces `racket_insert` calls with corresponding Scheme-Like ASTs that were tokenized and transformed into Python ASTs. The transformation allowed Scheme-Like code to be read, tokenized, converted to AST, and then dynamically linked within Python, making it possible to write hybrid Python and Scheme-Like programs.

5. Tokenization and AST Parsing

In order to parse Scheme-Like code effectively, we created a tokenizer function called `tokenize_racket()`. This function reads Scheme-Style syntax, tokenizes it, and converts it into a Scheme AST (nested list representation) that can be passed to our AST generation functions. The tokenizer supports the following constructs:

- Expressions with operators (e.g., `(+ 5 3)`).
- Conditional expressions (`if`, `>`, `<` etc.).
- Procedures and variable bindings (`proc`, `assume`).

A Scheme-Like expression like :

```
(assume ((x 5) (y 10)) (proc (x) (* x y)))
```

would be tokenized into :

```
['assume', [['x', 5], ['y', 10]], ['proc', ['x'], ['*', 'x', 'y']]]
```

This representation is then passed through the `racket_ast_to_python_ast()` function to convert it into a Python AST.

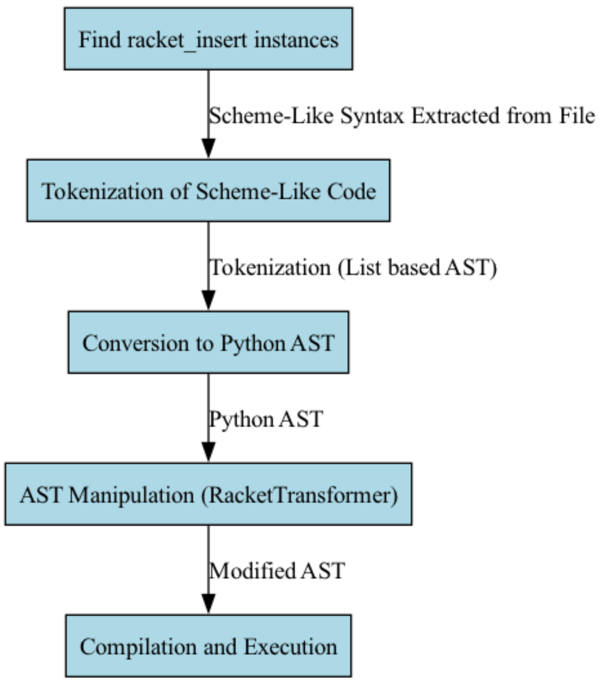
Find racket_insert instances
Scheme-Like Syntax Extracted from File
Tokenization of Scheme-Like Code
Tokenization (List based AST)
Conversion to Python AST
Python AST
AST Manipulation (RacketTransformer)
Modified AST
Compilation and Execution

Figure 1: Pipeline

7 Design of the Pipeline

Our pipeline for achieving interoperability between Python and a Scheme-like language involves several important steps.

Step 1: Finding `racket_insert` Instances

The pipeline begins by identifying instances of `racket_insert` in the Python code. This function call acts as a placeholder for dynamically inserting and executing Scheme-Like code within Python.

Step 2: Tokenization of Scheme-Like Code

Once a `racket_insert` instance is found, the corresponding Scheme-Like code is read from a file and tokenized using the `tokenize_racket()` function. This step breaks down the Scheme-Like code into tokens, representing components like operators, variables, numbers, and nested expressions. The result is a nested list structure that serves as an intermediate representation.

Step 3: Conversion to Python AST

The tokenized Scheme-Like code is passed to the `racket_ast_to_python_ast()` function, which converts the nested list representation into an equivalent Python Abstract Syntax Tree (AST). This step ensures that the Scheme-Like constructs are translated into Python-compatible AST nodes, such as binary operations, conditional expressions, and function definitions.

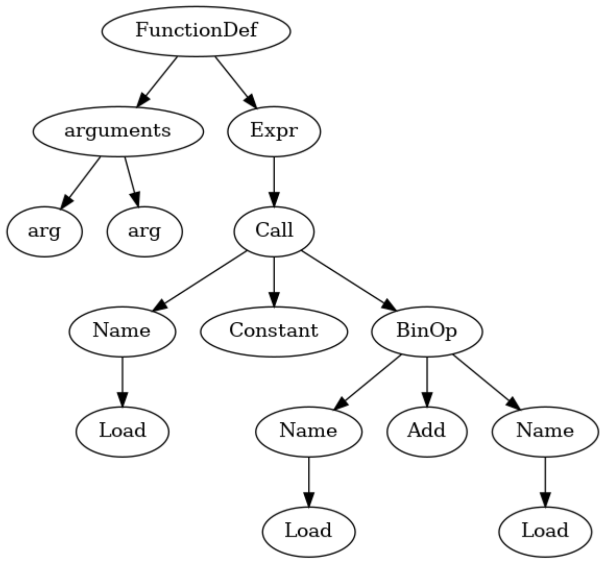

Figure 2: Python AST (ast module)

Step 4: AST Manipulation (RacketTransformer)

The Python AST is further manipulated using the `RacketTransformer` class. This step dynamically replaces the `racket_insert` function call with the corresponding Python AST representation of the Scheme-Like code. The transformed AST integrates the Scheme-Like logic into the Python code seamlessly.

Step 5: Compilation and Execution

The modified Python AST, now containing the translated Scheme-Like code, is compiled using Python's `compile()` function. This generates bytecode that can be executed directly by the Python interpreter. The pipeline ensures that the semantics of the original Scheme-Like code are preserved during execution.

8 Similar Tools

Cython

Cython is a superset of the Python programming language that enables developers to write Python code with optional C-inspired syntax extensions. It bridges the gap between Python's simplicity and C's high performance, making it an ideal choice for optimizing Python applications.

Cython works as a compiled language, converting annotated Python-like code into C code. This C code is then automatically wrapped in interface code to create CPython extension modules. These modules can be imported and used in Python applications just like regular Python modules, but with significantly reduced computational overhead during runtime.

Cython's ability to generate efficient and reusable CPython extensions makes it a powerful tool in scenarios where Python's native performance might be a bottleneck.

Pybind11

Pybind11 is a lightweight, header-only library specifically designed for exposing C++ types to Python and vice versa. It is widely used for creating Python bindings of existing C++ code. Pybind11's design minimizes boilerplate code and leverages modern C++11 features, such as tuples, lambda functions, and variadic templates, to provide a clean and efficient interface for bridging the gap between the two languages.

Pybind11's approach to bridging C++ and Python serves as an inspiration for our methodology. While Pybind11 focuses on static bindings at compile time, our project emphasizes dynamic translation and execution at runtime, enabling flexible and interactive integration of Scheme-like syntax with Python. Both approaches, however, share the overarching goal of enhancing language interoperability by leveraging the strengths of each language.

9 Possible Enhancements

- **Error Handling and Type Checking:** Add more sophisticated error handling to catch type mismatches between Scheme-Like Code and Python during AST transformation.
- **Optimization:** Optimize the generated code for performance, especially when dealing with recursive Scheme-Like functions.
- **Support for More Racket Features:** Extend support for other Racket features, such as more complex data structures, macros, and continuations.