WhyPY

A Custom Interpreter

https://why-py.vercel.app/

Aneesh Sambu
sambu.aneesh@research.iiit.ac.in
2023121012

Santhosh Kotekal Methukula
santhosh.km@students.iiit.ac.in
2022101057

December 10, 2024

Abstract

WhyPY is yet an another interpreter, introducing an esoteric like semantic layer atop traditional programming constructs. This report presents the design and implementation of WhyPY, a programming language that transforms conventional programming paradigms into thematically consistent mystical abstractions. The language features a tree-walking interpreter implemented in Python, supporting dynamic typing, first-class functions, and lexical scoping. Notable features include Vaughan Pratt parsing for expression handling, comprehensive error management, and a robust type system encompassing integers (NUMBER), booleans (TRUTH), strings (SCROLL), and functions (RITUAL). The implementation includes an interactive REPL environment and extensive documentation, making it suitable for both educational purposes and experimental programming. This paper details the language's design principles, implementation architecture, and potential applications in computer science education and programming language research.

1 Introduction

The language implementation features a tree-walking interpreter written in Python, chosen for its accessibility and robust standard library. Key technical features include:

- Vaughan Pratt parsing for expression handling
- Dynamic typing with comprehensive type checking
- Lexical scoping with proper closure support
- First-class functions with full recursion support
- String manipulation with concatenation operations
- Interactive REPL with multi-line input support

This report is organized as follows: Section II discusses the rationale behind choosing Python as the implementation language. Section III details the language's syntax and semantics. Section IV explores the implementation architecture. Section V presents example programs demonstrating language features. Sections VI and VII cover testing and future work, respectively.

2 About Project

WhyPY's primary contribution lies in its innovative approach to semantic transformation in programming language design. The language demonstrates several novel features and design choices:

- **Systematic Semantic Transformation**:
  - Comprehensive mapping of traditional programming constructs to thematic equivalents
  - Consistent application of mystical terminology across all language aspects
  - Preservation of conventional programming paradigms under transformed semantics
- **Implementation Techniques**:
  - Efficient implementation of Pratt parsing for expression handling
  - Integration of dynamic typing with comprehensive type checking
  - Error handling system with thematic error messages
- **Documentation and Accessibility**:
  - Comprehensive web-based documentation system
  - Interactive code examples and tutorials
  - Live demonstration environment for experimentation

3 Why Python? Why Not Lisp?

The choice of Python as the implementation language for WhyPY was deliberate, despite Lisp's traditional popularity in interpreter development. Here's why:

- **Accessibility**: Python's readable syntax and widespread adoption allows us to focus on interpreter logic rather than language design.
- **Memory Management**: Python's automatic garbage collection allows us to focus on interpreter logic rather than memory management.
- **Avoiding the "Curse of Lisp"**: While Lisp offers powerful metaprogramming capabilities, its unique syntax and concepts can create barriers for collaboration and understanding.
- **Modern Tooling**: Python's development environment, debugging tools, and testing frameworks provide a more modern development experience.
- **Collaboration Support**: As we are two people, we need to collaborate and understand the codebase. Python's readability and accessibility allows us to focus on interpreter logic rather than language design.

This decision prioritizes maintainability and collaboration over the traditional choice of Lisp in language implementation, aligning with WhyPY's goal of being both educational and accessible.

4 Syntax and Semantics

This section provides a detailed overview of its lexical structure, data types, and semantics.

4.1 Esoteric like Semantics

WhyPY transforms traditional programming concepts into mystical rituals through its unique esoteric like semantics (This is not a complete esoteric language). This transformation of common terms into mystical equivalents adds depth and creativity to the coding experience. Below are the core transformations, operator mappings, type system, and error handling mechanisms in WhyPY.

4.1.1 Core-Transformations

Core Language Transformations

primary!20**Traditional**	**WhyPY Equivalent**	**Description**
`let`	`manifest`	Variable declaration
`=`	`with`	Assignment operator
`fn`	`rune`	Function declaration
`{`	`unfold`	Block start
`}`	`fold`	Block end
`;`	`seal`	Statement terminator
`return`	`yield`	Value return
`,`	`knot`	Parameter separator
`if`	`whence`	Conditional start
`else`	`elsewise`	Alternative branch

4.1.2 Operators

Operator Mappings

primary!20**Traditional**	**WhyPY Equivalent**	**Description**
`+`	`augments`	Addition
`-`	`diminishes`	Subtraction
`*`	`conjoins`	Multiplication
`/`	`divide`	Division
`<`	`descends`	Less than
`>`	`ascends`	Greater than
`==`	`mirrors`	Equality
`!=`	`diverges`	Inequality
`!`	`negate`	Logical negation

4.1.3 Values

This table lists the core values used in WhyPY, including truth values and null representations.

Traditional	**WhyPY Equivalent**	**Description**
`true`	`verity`	Truth value
`false`	`fallacy`	False value
`null`	`void`	Null value

4.1.4 Type System

The WhyPY type system classifies data into different mystical categories, as shown in the table.

Traditional	WhyPY Type	Description
Integer	`NUMBER`	Whole number values
Boolean	`TRUTH`	Truth values
String	`SCROLL`	Textual inscriptions
Function	`RITUAL`	Callable rituals
Error	`MISHAP`	Error conditions
Return Value	`YIELDED`	Returned values

4.1.5 Error Messages

This table describes the mishaps that occur in WhyPY, mapping traditional errors to their esoteric counterparts.

Traditional Error	WhyPY Mishap
Type mismatch	Incompatible mystical energies
Undefined variable	Unknown sigil invoked
Syntax error	Mystical incantation malformed
Division by zero	Attempted division by the void

For a detailed exploration of WhyPY's esoteric semantics, refer to the official documentation: Esoteric Semantics Guide.

4.2 Lexical Structure

- **Statements and Seals**: Each statement concludes with the keyword `seal`, signifying the completion of an action.
- **Sigils (Identifiers)**: Identifiers must begin with a letter or underscore and can include letters, numbers, and underscores.

4.3 Data Types

WhyPY supports several unique data types, each imbued with its own mystical characteristics.

4.3.1 Numbers (Integers)

: Represented as pure entities.
Operations:

- **Arithmetic**: augments (addition), diminishes (subtraction), conjoins (multiplication), divide (division).
- **Comparison**: ascends (greater), descends (less), mirrors (equal), diverges (not equal).

4.3.2 Truth Values (Booleans)

: Truth values ascend to 'verity' or descend to 'fallacy'.
Operations:

- **Logical**: negate (logical NOT).

- **Comparison**: mirrors (equal), diverges (not equal).

4.3.3 Scrolls (Strings)

: Encased in quotation marks to represent text.
Operations:

- **Concatenation**: augments (string concatenation)

For a complete overview of these data types and their behaviors, refer to the `https://why-py.vercel.app/language-guide/data-types/`.

4.4 Manifestations (Variable Declarations)

Variables are introduced using the keyword `manifest`, assigning values or rituals.

```
manifest x with 42 seal
manifest isActive with verity seal
```

4.5 Rituals (Functions)

- **Declaration**: Defined using `rune` for parameters and `unfold/fold` to mark the body.

```
manifest add with rune(x knot y) unfold
    yield x augments y seal
fold seal
```

- **Invocation**: Rituals are invoked using parentheses.

```
manifest result with add(5 knot 10) seal
```

4.6 Flow Control

- **Whence-Elsewise Expressions**: Used for conditionals.

```
whence (x ascends y) unfold
    yield x seal
fold elsewise unfold
    yield y seal
fold
```

- **Block Unfoldings**: Groups of statements yield the value of the last expression.

```
manifest result with unfold
    manifest x with 5 seal
    manifest y with 10 seal
    x augments y seal
fold seal
```

4.7 Error Handling

Errors in WhyPY are known as **mishaps**, maintaining the mystical theme of the language.
For more details, refer to the official documentation at `https://why-py.vercel.app/language-guide/syntax-overview/`.

5 Implementation Architecture

The WhyPY interpreter is implemented in Python, following a modular architecture that separates concerns across multiple components:

Lexical Analysis

- </> Token categorization for language primitives (NUMBER, TRUTH, SCROLL)
- </> Operator precedence handling for arithmetic and logical operations
- </> String literal processing with escape sequence support
- </> Comprehensive error detection for malformed inputs

Syntactic Analysis

- </> Top-down recursive descent parsing for statements
- </> Pratt parsing for expression handling with precedence rules
- </> Abstract Syntax Tree (AST) generation with node typing
- </> Error recovery mechanisms for syntax violations

Type System

- </> NUMBER: Integer arithmetic with overflow protection
- </> TRUTH: Boolean values with logical operations
- </> SCROLL: String type with concatenation support
- </> RITUAL: First-class functions with proper closure semantics

6 Example Programs and Applications

This section presents a series of example programs that demonstrate WhyPY's features and capabilities:

Recursive Functions

The following example implements the Fibonacci sequence, demonstrating recursion and function composition:

```
MANIFEST fibonacci AS RITUAL (n) UNFOLDS
    WHEN n < 2 THEN
        RETURN n
    END
    RETURN fibonacci(n - 1) + fibonacci(n - 2)
END

MANIFEST result AS fibonacci(10)
REVEAL result
```

6.1 Higher-Order Functions

This example demonstrates higher-order functions and closures:

```
MANIFEST makeMultiplier AS RITUAL (factor) UNFOLDS
    RETURN RITUAL (x) UNFOLDS
        RETURN x * factor
    END
END

MANIFEST double AS makeMultiplier(2)
MANIFEST triple AS makeMultiplier(3)
REVEAL double(5)    # Outputs: 10
REVEAL triple(5)    # Outputs: 15
```

6.2 String Manipulation

The following example showcases string concatenation and manipulation:

```
MANIFEST greet AS RITUAL (name) UNFOLDS
    MANIFEST prefix AS SCROLL("Greetings, ")
    MANIFEST suffix AS SCROLL("!")
    RETURN prefix + name + suffix
END

REVEAL greet(SCROLL("Mystic One"))
```

6.3 Control Flow and Conditionals

This example demonstrates complex control flow and conditional statements:

```
MANIFEST isPrime AS RITUAL (n) UNFOLDS
    WHEN n <= 1 THEN
        RETURN FALSE
    END

    MANIFEST i AS 2
    WHILE i * i <= n UNFOLDS
        WHEN n % i = 0 THEN
            RETURN FALSE
        END
        MANIFEST i AS i + 1
    END
    RETURN TRUE
END

REVEAL isPrime(17)  # Outputs: TRUE
```

These examples illustrate WhyPY's capability to implement both simple and complex algorithms while maintaining its unique semantic style. The language's support for recursion, higher-order functions, and comprehensive type system enables the implementation of sophisticated programming patterns.

7 Installation & Setup

To install and set up WhyPY, refer to the README file in the repository for detailed instructions. In summary, clone the repository, navigate to the directory, and run the interpreter using the provided commands.

8 Documentation

Comprehensive documentation for WhyPY, covering syntax, examples, and implementation, is available at `https://why-py.vercel.app/`. It serves as a complete guide for using and exploring the language.

9 Contributions

The development of WhyPY was a collaborative effort, with distinct contributions from the team members:

- **Santhosh**: Implemented the **lexer**, **parser**, and **abstract syntax tree (AST)**, forming the foundational components of the interpreter.
- **Aneesh**: Implemented the **evaluator**, **REPL**, and **testing suite**, forming the runtime components of the interpreter. Designed the **mystical semantics**, and additional data types like **string support**. Created the **documentation website** with interactive demonstrations.

This division of responsibilities ensured a seamless blend of creativity and technical implementation in WhyPY, with each team member focusing on their areas of expertise while maintaining consistent communication and integration of components.

10 Testing and Validation

The WhyPY implementation includes a comprehensive testing suite that validates various aspects of the language:

Unit Testing

- ✓ **Lexer Tests**: Validate token generation and error detection
- ✓ **Parser Tests**: Verify AST construction and syntax validation
- ✓ **Evaluator Tests**: Ensure correct execution of language constructs
- ✓ **Type System Tests**: Validate type checking and conversion

Integration Testing

- ✓ End-to-end program execution
- ✓ REPL functionality and error handling
- ✓ File I/O and program loading
- ✓ Environment management and scope handling

11 Future Work

Several areas have been identified for future development:

Language Extensions

- **Type System Enhancements**:
  - Optional static typing support
  - User-defined type declarations
  - Generic type support
- **Additional Features**:
  - Module system for code organization
  - Exception handling mechanisms
  - Standard library expansion
  - Pattern matching support

Documentation and Community

- **Documentation Enhancements**:
  - Interactive tutorial system
  - Advanced programming guides
  - API documentation improvements
- **Community Development**:
  - Open source contribution guidelines
  - Community package repository
  - Educational resources

12 References and Foundational Works

The development of WhyPY was informed by several seminal works in programming language design and implementation:

Primary References

- Ball, T. (2016). *Writing an Interpreter in Go*.
  - Foundational interpreter implementation concepts
  - Pratt parsing methodology and implementation
  - Evaluation strategy design patterns
- Friedman, D. P., & Wand, M. (2008). *Essentials of Programming Languages*.
  - Theoretical foundations of language design
  - Semantic framework development
  - Type system architecture
- Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs*.
  - Metacircular evaluator design principles
  - Environment model implementation
  - Closure semantics and implementation

13 Academic Context

This project was developed as part of the "Principles of Programming Languages" course at the International Institute of Information Technology, Hyderabad.

Course Information

- **Course**: Principles of Programming Languages
- **Institution**: International Institute of Information Technology, Hyderabad
- **Instructor**: Professor Venkatesh Choppella
- **Course Resources**: `https://faculty.iiit.ac.in/~venkatesh.choppella/popl/`

The course provided the theoretical foundation and practical framework necessary for implementing WhyPY, particularly in areas of language semantics, type systems, and interpreter design.

Remarks

For optimal experience, please visit the website `https://why-py.vercel.app/`