**TINY**

Tokenised Interpreter for Non-complex Yield

-

1. Project Vision and Scope

*TINY* is a JIT interpreter coded in C++ for a custom basic arithmetic programming language. The language is designed with education and restricted challenge in mind. The intention behind the project is to be an entry point for people who want to learn basic programming, as well as experienced developers eager to experiment with more complex projects without the steep entry curve.

*TINY* is open-source – unlike many beginner-friendly languages – so you can easily see what the interpreter is doing under the hood. In addition, we provide official documentation describing how it works, which makes learning more approachable. Coding enthusiasts can even fork and modify the source code to tailor their needs.

We will never introduce high level functionality or deliver a versatile product meant for serious production use. *TINY* is only supposed to be a concrete foundation on which anyone is able to expand with little effort and time.

Scratch is an accessible tool for game dev. *TINY* is the equivalent for JIT interpretation. We are all about introducing a way to learn not only through engaging the language itself, but also by diving into the cogs turning within, cracking the books of code execution.

1. User Personas

* The curious of all ages with no prior experience with programming languages who want to learn numerical computing in a simple inviting environment
  + Users will want to use our product for its ease of use and readily available sources
* Coders on an intermediate skill level who want to familiarize themselves with JIT interpreter implementation and lower-level programming
  + Users will want to use our product because of its open-source nature
  + Users can build upon TINY by designing new features or experimenting with modifications to its syntax or core functionality
  + There’s no popular alternative for this user group
* Hobbyists, educators, and enthusiasts interested in language design.
  + This appeals to those interested in gaining practical experience with language engineering and compiler/interpreter design.

1. Use Cases
2. For diving into coding

|  |  |  |
| --- | --- | --- |
|  | Steps | Outcomes |
| 1. | Download the TINY.exe application (a pre-compiled interpreter ready for use) from our website | You’re all set up for coding in TINY. |
| 2. | (optional) Create a \*.tiny file to write your code in | This will allow you to write complex sequences of commands to be executed at once in series. |
| 3. | (when using an IDE to write code) Visit the IDE guide on our website for instructions on how to install essential plugins for your editor | Our plugins provide syntax highlighting, auto-completion, facilitated access to offline documentation and various project templates. |
| 4. | Follow the online documentation and write your commands | This will enhance your problem-solving capacity and make you ready for writing more complex code in other programming languages without overwhelm. |
| 5. | In case of encountering issues, refer to our online Get Started Guide | The Get Started Guide provides a simpler version of the text instructions from the documentation expanded for those less knowledgeable of computers. |

1. For advanced tinkering

|  |  |  |
| --- | --- | --- |
|  | Steps | Outcomes |
| 1. | Download the TINY source code or fork the GitHub repository | You’re all set up for tampering with the codebase and learning to navigate through the modules. |
| 2. | Read the documentation and build the interpreter binary | This will provide the foundation for modifying and testing the language as you add or change features. |
| 3. | Visit the official TINY Devlog YouTube channel | This will provide you with a more in-depth look into the structure and implementation with commentary. |
| 4. | Modify the source code, implement new features, or experiment with altering the existing modules | This will familiarize you with the hows and whys of interpreters and improve your skills and understanding. |
| 5. | Learn about parsing and math operation mechanisms work | This will soothe your curiosity about the potential implementation of the inner workings of parsing strings to arithmetic. |
| 6. | Get to know the Triggered Injection for New Yield API and learn how to hook into internal events | This allows you to add custom behaviours or modify existing ones in an intuitive but extremely flexible way. |
| 7. | (optional) Use the TINY.exe executable as a CLI to create a project with the Triggered Injection for New Yield API template | The template provides a basic structure, boilerplate code and some examples, which facilitate the modification process. |
| 8. | Use the CLI to compile all created projects and integrate them into the compiler by patching the TINY.exe executable | This tool provides an automatic method of applying your work. |

1. Experimenting with Language Design and Extension

|  |  |  |
| --- | --- | --- |
|  | Steps | Outcomes |
| 1. | Fork the TINY GitHub repository or download the source code | You're all set up for experimenting with the codebase and exploring language design possibilities. |
| 2. | Familiarise yourself with the documentation and build the interpreter | This will provide the foundation for modifying and testing the language as you add or change features. |
| 3. | Visit the official TINY Devlog YouTube channel | This will provide you with a more in-depth look into the structure and implementation with commentary. |
| 4. | Implement new syntax or tweak the language rules, experimenting with language behaviour | By modifying TINY's syntax or structure, you'll gain insight into language design principles and the foundational components of programming languages. |
| 5. | Get to know the LDE (Language Design Extension) API and learn how to create custom syntax | This API allows you to implement custom syntax patterns and integrate them into the language. |
| 6. | (optional) Use the TINY.exe executable as a CLI to create a project with the LDE API template | The template provides a basic structure, boilerplate code and some examples, which facilitate the design process. |
| 7. | Observe and analyse how changes impact parsing and interpretation mechanics | This step will deepen your understanding of parsing, tokenisation, and JIT interpretation, enhancing your grasp of low-level programming and interpreter design. |

1. Project Planning and Schedule

**1. Project Planning and Research (Week 1-2)**

* **Key Tasks**:
  + Define project scope and vision in detail.
  + Research JIT interpreters and simple programming language design.
* **Milestone**: Finalised project requirements and technical foundation.
* **Estimated Time**: 2 weeks
* **Dependencies**: Project Vision (already established).
* **Responsibility**: Lead Developer/Researcher.

**2. Initial Development and Basic Functionality Implementation (Week 3-6)**

* **Key Tasks**:
  + Develop core syntax rules for TINY.
  + Implement basic arithmetic operations and tokenisation.
  + Establish system commands (exit, clear).
* **Milestone**: First working version of TINY capable of handling basic operations and system commands.
* **Estimated Time**: 4 weeks
* **Dependencies**: Completion of planning phase and research.
* **Responsibility**: Development Team.

**3. Advanced Feature Implementation (Week 7-12)**

* **Key Tasks**:
  + Implement variable and constant creation (vint and cint).
  + Develop overflow protection for arithmetic operations.
  + Enable functionality for comments and substitution of variables.
* **Milestone**: Advanced version of TINY with variable handling, comments, and overflow protection.
* **Estimated Time**: 6 weeks
* **Dependencies**: Basic functionality implementation.
* **Responsibility**: Development Team, Quality Assurance (for testing).

**4. Documentation and Educational Resources Creation (Week 13-15)**

* **Key Tasks**:
  + Write comprehensive documentation explaining TINY’s syntax, features, and underlying JIT processes.
  + Develop a "Getting Started Guide" and official documentation for beginner users.
* **Milestone**: Complete user documentation and resources for learning and experimenting with TINY.
* **Estimated Time**: 3 weeks
* **Dependencies**: Finalised advanced features.
* **Responsibility**: Documentation Team, Content Creator.

**5. Testing and Quality Assurance (Week 16-18)**

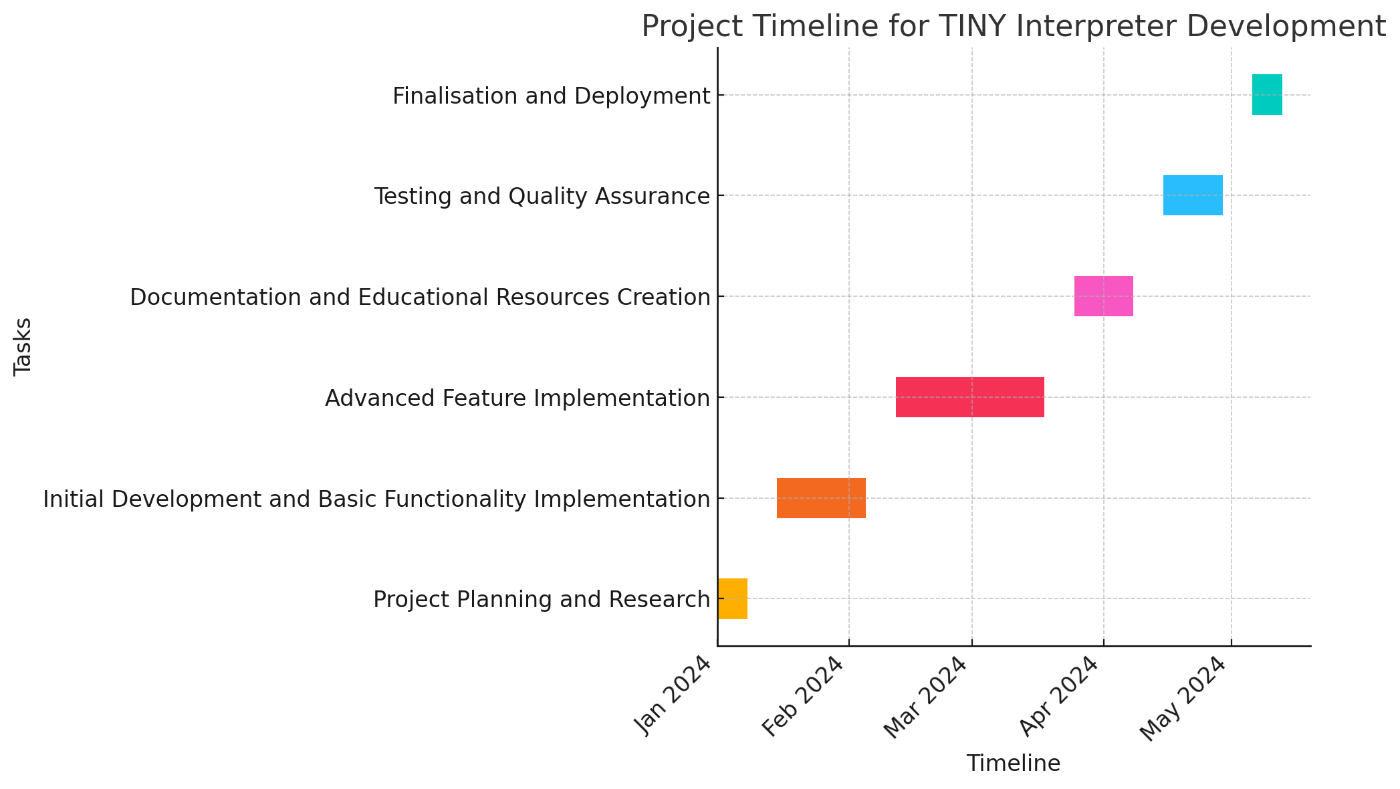
* **Key Tasks**:
  + Conduct thorough testing of arithmetic operations, variable handling, and overflow protection.
  + Identify and resolve bugs or issues.
  + Implement unit tests to ensure code stability and performance.
* **Milestone**: Stable, tested version of TINY ready for public release.
* **Estimated Time**: 3 weeks
* **Dependencies**: Completed feature implementation and documentation.
* **Responsibility**: Quality Assurance Team.

**6. Finalisation and Deployment (Week 19-20)**

* **Key Tasks**:
  + Package the interpreter for distribution (e.g., .exe file).
  + Deploy the source code and binaries on GitHub.
  + Promote the interpreter through coding communities.
* **Milestone**: Official launch of TINY with public access to documentation and code.
* **Estimated Time**: 2 weeks
* **Dependencies**: Completion of testing and quality assurance.
* **Responsibility**: Project Lead, Development Team, Marketing/Outreach.

**7. Post-Launch Support and Iterative Improvements (Ongoing)**

* **Key Tasks**:
  + Address user feedback and update the interpreter as needed.
  + Improve documentation based on user input.
  + Plan for potential future feature additions.
* **Milestone**: Continuous improvements and support for TINY.
* **Estimated Time**: Ongoing
* **Dependencies**: Initial deployment and user adoption.
* **Responsibility**: Development Team, Documentation Team.



1. Tooling Selection and Justification

**Tools List and Purpose**

1. **Programming Language: C++**
   * **Purpose:** Used for implementing the TINY interpreter due to its high performance, low-level memory control, and compatibility with JIT compilation techniques. C++ is also well-suited for managing the intricacies of tokenisation and parsing.
2. **Integrated Development Environment (IDE): CLion**
   * **Purpose:** Provides a robust development environment with features like code navigation, debugging, and syntax highlighting for C++. Its seamless integration with CMake simplifies project configuration and management.
3. **Version Control System: GitHub**
   * **Purpose:** Serves as the central repository for the TINY project, enabling version control, collaboration, and open-source contributions. GitHub's issue tracker and pull request workflow streamline development and community interaction.
4. **Build System: CMake**
   * **Purpose:** Facilitates the generation of build files for multiple platforms, ensuring compatibility and a smooth build process for contributors. It integrates well with CLion for streamlined project management.
5. **Continuous Integration/Continuous Deployment (CI/CD) System: GitHub Actions**
   * **Purpose:** Automates testing and deployment processes, ensuring that changes to the codebase are validated before merging. CI/CD minimises manual errors and maintains code stability.
6. **Documentation Platform: MkDocs**
   * **Purpose:** Used to generate user-friendly and professional documentation for the TINY interpreter. MkDocs enables the creation of well-organised, searchable documentation with ease.
7. **Collaboration Tool: Slack**
   * **Purpose:** Facilitates communication between team members for brainstorming, updates, and issue resolution. Slack's integration with GitHub ensures real-time notifications of changes and progress.
8. **Design Tool: Draw.io**
   * **Purpose:** Used for designing the architecture and flow diagrams of the interpreter. This visual aid is essential for both documentation and developer understanding.

**Justification for Tool Selection**

1. **C++:** Chosen for its execution speed and ability to handle low-level memory operations, essential for a performance-critical project like a JIT interpreter.
2. **CLion:** Its robust debugging capabilities and built-in CMake support make it the ideal choice for a project of this nature, enhancing development efficiency.
3. **GitHub:** As an open-source project, GitHub provides visibility, facilitates collaboration, and manages version control effectively.
4. **CMake:** Ensures that the build process remains platform-independent, essential for an open-source project that may be used across different systems.
5. **GitHub Actions:** Automates repetitive tasks such as building and testing, improving productivity and reducing the risk of introducing bugs.
6. **MkDocs:** Ensures that documentation is well-structured and accessible, which is crucial for the project's educational focus.
7. **Slack:** Promotes efficient communication, especially in a two-person team, and prevents bottlenecks in collaboration.
8. **Draw.io:** Provides an intuitive way to represent the interpreter's architecture, essential for explaining complex concepts to contributors and learners.

**Integration and Complementarity**

1. **C++ and CLion:** CLion supports C++ development with advanced debugging and refactoring tools, ensuring code quality and productivity.
2. **GitHub and GitHub Actions:** GitHub Actions is directly integrated with the GitHub repository, enabling automatic builds and tests on every code push.
3. **CMake and CLion:** Both tools work seamlessly to manage and build the project, reducing configuration overhead.
4. **MkDocs and GitHub:** Documentation generated with MkDocs can be hosted directly on GitHub Pages, making it easily accessible to users and contributors.
5. **Slack and GitHub:** Slack notifications about GitHub activity ensure that team members are always aware of updates and issues.
6. **Draw.io and MkDocs:** Flowcharts and diagrams from Draw.io are integrated into the documentation to enhance user comprehension
7. Testing Plan

Types of tests

1. Unit Tests
   * **Description**: Validate individual components (e.g., lexer, parser, interpreter functions) to ensure correctness in isolation.
   * **Purpose**: Identify and fix bugs early in the development cycle.
   * **Tools**: Google Test framework for C++.
2. IntegrationTests
   * **Description**: Test the interactions between different modules of the interpreter (e.g., lexer output as input for the parser, parser feeding into the interpreter).
   * **Purpose**: Ensure seamless collaboration between system components.
   * **Tools**: Custom C++ test scripts combined with Google Test/Catch2.
3. User Acceptance Tests
   * **Description**: Simulate end-user workflows to confirm the interpreter meets usability and educational goals.
   * **Purpose**: Validate the product from the user's perspective and ensure the language behaves as intended for educational purposes.
   * **Tools**: Script-based test cases, manual testing by developers and volunteers.

Test cases

Unit Tests

1. Lexical Analysis (Lexer)
   * **Description**: Test if the lexer correctly tokenizes a valid TINY expression, e.g. “1 + 2 \* 3”.
   * **Steps**:
     + Input: 1 + 2 \* 3.
     + Expected Output:  
       [NUMBER: 1], [PLUS: +], [NUMBER: 2], [MULTIPLY: \*], [NUMBER: 3].
   * **Outcome**: Tokens are correctly identified without errors.
2. Syntax Parsing (Parser)
   * **Description**: Test if the parser generates a correct Abstract Syntax Tree (AST) for an arithmetic expression.
   * **Steps**:
     + Input: 3 + (4 \* 5).
     + Expected Output: AST resembling (PLUS 3 (MULTIPLY 4 5)).
   * **Outcome**: Correct hierarchical AST structure.

Integration Tests

1. Full Expression Evaluation
   * **Description**: Test if the interpreter evaluates a valid arithmetic expression correctly.
   * **Steps**:
     + Input: 3 + 4 \* 5.
     + Expected Output: 23.
   * **Outcome**: Interpreter outputs the correct result.
2. Error Handling
   * **Description**: Test the interpreter's response to a syntactically incorrect expression.
   * **Steps**:
     + Input: 3 + \* 5.
     + Expected Output: Error message:   
       Syntax Error: Unexpected token '\*' at position 4.
   * **Outcome**: Error is reported with accurate feedback.

User Acceptance Tests

1. Basic Workflow
   * **Description**: Validate that a beginner can use TINY to evaluate arithmetic expressions.
   * **Steps**:
     + Follow the guide to input 6 / 3 + 2 \* (4 - 1).
     + Expected Outcome: Correct result 8 is displayed with no system crashes or misbehavior.
   * **Outcome**: Usability is confirmed.
2. Documentation Usability
   * **Description**: Verify that users can navigate and understand the official documentation to write and evaluate simple programs.
   * **Steps**:
     + Read a sample tutorial and execute the given examples.
     + Expected Outcome: User successfully executes all examples without external assistance.
   * **Outcome**: Documentation meets educational standards.

Testing schedule

|  |  |  |
| --- | --- | --- |
| **Phase** | **Type of Test** | **Responsible** |
| Week 1 | Unit Testing (Lexer, Parser) | Developers |
| Week 2 | Integration Testing | Developers |
| Week 3 | User Acceptance Testing | Volunteers and developers |
| Ongoing Post-Launch | Regression Testing | Developers |

Testing tools and frameworks

1. Google Test
2. Manual Testing Tools
3. Documentation Usability Testing
4. Functional and Non-Functional Requirements

Functional requirements

1. Arithmetic expression evaluation
   * **Description**: The interpreter must evaluate arithmetic expressions (e.g., 1 + 2 \* 3) and return correct results.
   * **Testable**: Given an input expression, the output must match the expected result.
2. Error handling
   * **Description**: The system should detect and report syntax errors with clear error messages (e.g., Unexpected token '\*').
   * **Testable**: Invalid input should produce the correct error message.
3. Tokenization and parsing
   * **Description**: The interpreter must tokenize input strings and parse them into an Abstract Syntax Tree (AST) for evaluation.
   * **Testable**: Tokens and AST must match expected structures for valid inputs.
4. Interactive console
   * **Description**: Provide an interactive shell where users can enter and evaluate expressions in real time.
   * **Testable**: Users should be able to input expressions one at a time and get results immediately.
5. Triggered Injection for New Yield API and Language Design Extension API
   * **Description**: The interpreter should be easily extensible through the use of the Triggered Injection for New Yield API and the Language Design Extension API
   * **Testable**: Features introduced using these APIs should seamlessly integrate with the rest of the language.
6. Cross-platform support
   * **Description**: The interpreter must work on major operating systems (Windows, macOS, Linux).
   * **Testable**: Verify functionality on different OS environments.

Non-functional requirements

1. Simplicity
   * Both the interpreter and the language itself should be easy to understand by beginners and experienced developers.
   * **Method of testing**: UAT by volunteers and developers.
2. Usability of the official documentation
   * Documentation should clearly describe installation, usage, and key features for beginners
   * **Method of testing**: Evaluate user feedback and successful execution of examples.
3. Performance in growing scale
   * The interpreter should handle increasingly complex expressions without significant degradation in performance.
   * **Method of testing**: Evaluate performance for expressions of varying complexity.