Yanli Wang(yw6822), Weijun Huang(wh1322), Yunkai Zhang(yz9522), Cheng Ma(cm1122)

50007 WACC FINAL REPORT.

1 FINAL PRODUCT

1.1 Functional Specification Compliance

The WACC compiler's front-end phase possesses several key functionalities for parsing and semantic analysis. Pattern matching simplifies the handling of various syntaxes. The error-handling mechanism produces precise feedback on syntax and semantic errors. Connecting AST nodes to scopes ensures accurate symbol resolution and scope management for semantic integrity. Improvements in error messages and pattern matching efficiency could significantly enhance usability and robustness.

On the back-end, translating high-level program constructs into intermediate representation before final code generation shows potential for optimizations and cleaner representation. The code generation abstraction layer offers flexibility and ease of extension, while register mapping management ensures memory-mapped register mechanism correctness.

1.2 Further Development

While the current design offers robustness, it poses challenges for future development. Rapid integration of cutting-edge language features or optimization techniques may require extensive testing and refactoring. The reliance on a complex abstraction system for register mappings and IR management could hinder new developers, slowing innovation. Balancing performance optimization and codebase complexity is crucial for maintaining system maintainability and preventing regressions as the compiler evolves.

1.3 Performance Characteristics

The compiler exhibits a structured approach to code generation and optimization, laying a foundation for efficiently executable code. Yet, true performance entails optimizing at multiple compilation stages, including dead code elimination and constant folding. These optimizations directly enhance runtime efficiency, reducing execution time and resource consumption. Leveraging existing C library functions further suggests intelligent reuse of optimized code, potentially boosting output performance without redundant effort.

2 PROJECT MANAGEMENT

2.1 Team Composition and Roles

Our team operates without a central leader, fostering a collaborative atmosphere where everyone stays in sync. Our project workflow is methodical, starting with collective task volume analysis. Weekly group discussions address conflicts and monitor individual progress, ensuring open communication and keeping the project on track. Each team member has distinct responsibilities:

1. **Yunkai Zhang** Tasked with constructing the main framework and codebase of the project and allocating general tasks among the team members. Yunkai's role is pivotal in setting the project's foundation and ensuring that each team member has clear objectives and responsibilities.

2. **Weijun Huang** Responsible for the primary codebase for the project as well. Weijun's expertise in coding and software engineering is crucial for the development of functional and efficient software solutions.

3. **Yanli Wang** Overseen the overall progress of the project. Yanli's role ensures that the project meets its milestones on schedule, alongside conducting analytical reviews for quality assurance.

4. **Cheng Ma** Focused on debugging and code cleanliness improvement. Cheng's attention to problem-solving skills is essential for maintaining the integrity of the code and harmonizing team dynamics.

2.2 Management Tools

We leverage various tools to enhance project management and execution. `Git` is used for version control and detailed task allocation, facilitating a clear and efficient development process. `LucidChart` aids in the conceptualization of project architecture, offering a visual understanding of task inter-dependencies. The adoption of `CI/CD` pipelines automates key stages of code integration, testing, and deployment, streamlining the development cycle. For testing, we conduct integration tests by scripting in Python. This script compiles for all `.wacc` files in the given paths. We utilize `Docker` to emulate an x86 Linux environment for cross-platform compatibility. To streamline the testing process, we automate test execution by scripting within a `Makefile`.

2.3 Discussion

Despite our successes, we've encountered overly specific task differentiation, which leads to reduced coding time and early-stage code conflicts due to communication gaps. To avoid such a situation for future projects, we aim to shorten the conceptual phase and enhance team understanding of each other's work. Reflecting on our workflow and the challenges encountered, we are committed to implementing lessons learned to enhance future project outcomes.

3 DESIGN CHOICES AND IMPLEMENTATION DETAILS

3.1 Rust

Rust distinguishes itself from the other two popular choices for WACC, Haskell and Scala, in compiler implementation through its deterministic memory management and robust performance.

1. **Memory Safety without Garbage Collection** Rust's ownership model ensures memory safety and efficiency and eliminates common bugs like dangling pointers and buffer overflows without runtime overhead.
2. **Functional Programming Features** Rust's pattern matching, closure system, and functional features are powerful tools for succinctly expressing complex logic, particularly in concisely parsing and navigating ASTs, facilitating easier maintenance and extension of compiler functionalities.
3. **External Library Management System** Rust has a uniform system for the use of external libraries (crates), with ease to utilize and manage various external tools from the community.

3.2 Parser Tool: Chumsky

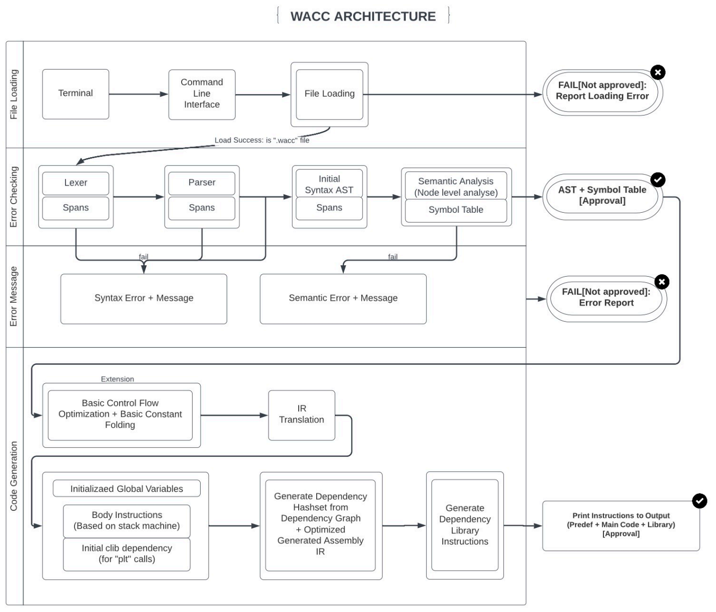
There exist multiple parser tools in Rust such as the famous **nom** library, featuring highly customisable parser choices and there is **lalrpop**, a parser generator library. Considering the need for error location reporting and our preference for parser combinator approach, we eventually picked `Chumsky`, having its elegant error formatting tool `Ariadne` for errors with a native Rust flavour, and its multiple functions such as Pratt parsing and location mapping.

3.3 Architecture for Compiler & Design Pattern

The architecture of the WACC compiler is meticulously designed to leverage Rust's strengths across its front-end and back-end phases.

In the front-end phase, the compilation process begins with the loading of a WACC source file through a command-line interface. The lexer then tokenizes the input, and the parser analyzes the tokens to construct an initial Abstract Syntax Tree (AST). If any errors are encountered during these stages, corresponding error messages are generated and the process halts. Successful parsing leads to semantic analysis, where the AST is validated and a symbol table is constructed. This stage enforces semantic rules and outputs the Program's full AST along with its symbol table for the next phase.

The back-end phase is focused on transforming the approved AST into executable code. Initially, the AST undergoes optimization, including basic control flow and constant folding. Subsequent IR (Intermediate Representation) Translation prepares the code for machine-level representation. This includes the initialization of global variables and construction of stack-based body instructions with library dependencies. A dependency graph further refines the IR, which ultimately leads to the generation of the final assembly code. This code, which comprises pre-defined structures, the main code body, and linked library functions, is the final output of the compiler, ready for execution on the target x86 architecture.


WACC ARCHITECTURE
File Loading
Terminal
Command Line Interface
File Loading
FAIL[Not approved]: Report Loading Error
Load Success: is ".wacc" file
Error Checking
Lexer
Spans
Parser
Spans
Initial Syntax AST
Spans
Semantic Analysis (Node level analyse)
Symbol Table
AST + Symbol Table [Approval]
fail
fail
Error Message
Syntax Error + Message
Semantic Error + Message
FAIL[Not approved]: Error Report
Code Generation
Extension
Basic Control Flow Optimization + Basic Constant Folding
IR Translation
Initializaed Global Variables
Body Instructions (Based on stack machine)
Initial clib dependency (for "plt" calls)
Generate Dependency Hashset from Dependency Graph + Optimized Generated Assembly IR
Generate Dependency Library Instructions
Print Instructions to Output (Predef + Main Code + Library) [Approval]

The development of WACC also emphasized design patterns and development methodologies:

1. **TDD** We deployed Test-Driven Development (TDD) for the WACC compiler to prioritize code quality and functional correctness from the beginning. By writing tests before actual code, we ensured each feature met its specifications precisely, fostering a modular design. TDD not only helped in detecting issues early but also served as documentation, guiding future enhancements.

2. **Visitor Patterns** The visitor pattern encapsulates algorithms that work on a diverse collection of objects. It enables the creation of multiple algorithms for the same data without altering the data itself or its core behavior. Additionally, this pattern separates the traversal of object collections from the actions performed on each object. By employing this strategy, code becomes more structured, leading to enhanced readability.

4 BEYOND THE SPECIFICATION

4.1 Extensions

4.1.1 Optimisation: Constant Folding & Control-Flow Processing

Overview and Restrictions One part of our extension were about focusing on optimisation of generated assembly code to make it less tedious and remove some redundant assembly instructions. All the optimisation process would eventually contribute to code cleanup, improving the efficiency and running performance of the target architecture, and generating readable and precise assembly code.

Via compiler designing principles, the optimisations in assembly code could be generalised into several categories: local optimisation, regional optimisation, global optimisation, and inter-procedural optimisation. Each of the optimisation method will hold a different perspective of the IR structure and therefore will require different levels of information.

Regional optimisations will consider a subset formed by several nodes (for example, in some cases, a node) in the control-flow graph (CFG) as a whole and perform optimisations based on the shape, inbound edges, and in some cases actual running setup when entering a certain block. Similarly but on a higher level, global and interprocedural optimisations will be dependent on the CFG, however focusing more on the shape of the whole graph and larger blocks. We have attempted to implement a minimal CFG design sketch in one of our attempts, however did not have enough availablility to complete such design with capability to support regional optimisations. Therefore we would be focusing on local optimisation and AST-level optimisation in our final design to guarantee correctness.

Constant Folding One of the ideas of conducting optimisation is on folding constants. Given a long expression such as `1 + (2 * (344 % 9 + 277))`, it is totally feasible to calculate the expression at compile-time internally, without generating codes and AST nodes for each of the smaller expressions (with 'depth = 4'). Consider a program with a huge amount of longer but calculatable expressions, these internal folding processes could largely reduce the size of the code generated and therefore complete optimisation. Basic Constant Folding reduces compile-time known expressions to their simplest form automatically.

We did not implement the full constant propagation and folding as this would as well rely on the CFG and is considered as a regional optimisation. Instead, we implemented a partial interpreter (focusing on the simplification of expressions) and would simply our AST after semantic checking and type inference. Therefore, multi-leveled expression nodes would be abstracted into one single constant expression node when possible via introducing `BasicValueType(Value)`. This will calculate all expressions that could directly result in calculating a basic value without further context to be converted at compile time. However, notice that expression calculations that will induce runtime errors (such as divide by zero, arithmetic overflow) should still be reserved so the correct error types could be properly handled.

We mimic the behaviours of expressions as close to its natural calculation as possible. Therefore we implemented relative '`std::ops`' traits to mimic natural calculation for 'BasicValue' types.

Control-Flow Processing Similar to constant folding, control-flow optimisations is another approach we can apply on the AST. Consider the `'if'` and `'while'` statements:

According to the semantics of `'if'` statements. If the condition `cond` evaluates to true, we will automatically execute `st1` then exit the if-statement, and vice versa. Similarly, although we cannot determine the behaviour of `while true ...` patterns (as internally variable changes and it requires the internal body to reach a certain state), we can immediately determine that `while cond ...` statements are equivalent to a `skip` if `cond` evaluates to `false`, because the body statements will never be executed.

Therefore, by combining with the previous part, we can simply control-flow statements of certain patterns and (partially) avoid dead code at compile-time.

Peephole Optimisation According to the Dragon Book, Peephole Optimisation is also a type of local optimisation that could be completed via inspecting only the nearby lines. It has been generally generalised into the following patterns:

1. Redundant `MOV` instructions: Instructions in the format of `Mov A A` are totally unnecessary as this does no effective operation
2. Duplicated `LOAD` and `STORE` statements (this is more related to `aarch64` as compared to x86)
3. Algebraic Simplification

Algebraic Simplification including the general reduction: $a \pm 0 = a$, $a * 0 = 0$, $a * 1 = a$, and $a/1 = a$.

We would evaluate the left and right hand sides of the given expression and check whether they match the given patterns above. If any match is found, the pattern would be reduced immediately to its simplest form (as given on the right hand side).

Optimisation Visualisation Our implementation by default turns on the optimisation, which is the version provided to the integration test. To turn off optimisation, we provided another bash script `unoptimised-compiler` to view the originally generated unoptimised version of assembly codes. Similar to running `./compile target.wacc`, just run `./unoptimised-compiler target.wacc` to view the generated assembly code.

4.1.2 Local Type Inference

In the local type inference extension, we allow programmers to implicitly indicate the variable types and function return types rather enforcing explicit casts. Our compiler will deduct the actual type using the surrounding codes as hints. We used a new keyword `val` to indicate implicit types.

To complete the initial inference, there are two cases. In variable declarations, we first evaluate the value on the RHS of `=`. This allows us to determine both the type and value for the declaration. This approach remains viable as we do not permit declarations without variable signatures. For instance "`val x = 5`" will add seen as "`int x = 5`", and in this case `x` with type int will be added into symbol table. The process is similar for return type inference. We infer the implicit return type by calculating the expression in return statements. For example, "`return true`" will infer the return type as `bool`.

For the error reporting phase, we will infer the actual types for variables during declaration or infer return types when encountering a return statement for the first time within a function. Subsequently, we conduct semantic checks by comparing the expected types with the actual types during expression evaluation and statement analysis. If there is a type mismatch, we generate a report indicating the location of the error.

4.1.3 Global Monomorphic Type Inference

In case of global monomorphic type inference, there are several additional factors. Firstly, we introduced inferred types for function arguments. Furthermore, we aimed to infer a function's return type within its declaration via `<rvalue>` manipulation.

The process of type inference is conducted within the semantic checker. When a statement invokes another function, a call stack is established to document such functions. Should a function not be previously recorded in `AVAILABLE_FUNCTIONS`, it signifies the absence of inferred parameter types. Consequently, the inference process for the current function is halted, and efforts are redirected towards inferring types within the called function, subsequent to which the function is registered in `AVAILABLE_FUNCTIONS`.

Upon the completion of the initial inference phase, a reevaluation of loop functions within `AVAILABLE_FUNCTIONS` is undertaken to ensure the inference of all function return types. This is followed by a comprehensive iteration over all functions to guarantee the updated accuracy of statements. The augmented Abstract Syntax Tree (AST) is then returned. Any discrepancies unearthed during the inference and semantic verification stages are meticulously reported and showcased.

Consider, for example, two functions, *f* and *g*, with the main function invoking *f* at the declaration phase. In this context, *f* is relegated to the calling stack, incorporated into `AVAILABLE_FUNCTIONS`, and its parameter type is deduced. Should function *f*, within its ambit, call function *g* in another declaration statement and return this variable, the analysis of the current line in *f* is suspended to facilitate the inference of *g*'s return type.

In instances where function *g* does not invoke additional functions, its return value is ascertainable through the expression, thereby enabling the retrospective inference of the variable type in the main function. Conversely, scenarios involving mutual recursion between functions *f* and *g*, akin to factorial computation, introduce heightened complexity. Here, a reciprocal calling pattern is established, necessitating a nuanced approach to avoid redundant evaluations. Specifically, upon returning to function *f* and identifies the invocation of *g*, the return type of *f* is deduced to match that of *g*.

This iterative process extends to a subsequent loop through `AVAILABLE_FUNCTIONS` to affirm the return types. This confirmation phase plays a pivotal role in steering the process of deducing the return types of other functions whose return types have yet to be established. Once all function arguments and return types are inferred, a final review of all functions is conducted to confirm the inference of their declaration types. This exhaustive process culminates in the generation of an AST, the integrity of which can be validated through semantic check outcomes and direct inference results, thereby ensuring the program's functionality.

4.2 Future Extension: Efficient Register Allocation and Regional Optimisation

Given more time, we plan to fully implement a functional CFG design, deploy the regional optimisation methods, and apply an efficient register allocation algorithm via graph colouring.

As much of our initial construction for extension would rely on a functionally correct CFG design, we would try to construct such graph and apply all further changes.

Fixing and fully implementing the CFG structure would be the starting point of all the further improvements. CFG will record the trace of register usage and flow trends in the whole program, allowing us to analyze the program structure as an entire entity.

Our current register allocation algorithm rely on a pure stack-machine approach to reduce stack operations, minimise unnecessary memory accesses, and improve efficiency as it favours register reuse over stack and heap usages.

Other possible extensions to consider are the common regional and global optimisation methods such as full constant propagation and control-flow analysis. With a full CFG model, it would be possible to simplify redundant branches (edges in the graph) and dead-code. The graphical representation would allow us to detect the jumping strategy of the program and redesign or optimise the control logic.