Final Project: Cool Compiler

Class: CI2 2024/2025
Course: Compilers
Instructor: Eroui Abdelaziz
Due Date: 21st of February 2025

Summary

This document outlines the final project for the compiler course.
In this project, you will implement a complete compiler for the Cool (Classroom Object-Oriented Language) programming language. You will build all phases of the compiler, from lexical analysis to optimisations and code generation.

Your tasks are to:

1. **Complete the Front end compiler**
    - Finish the implementation of the lexer, parser, and semantic analyser to fully support the cool language as defined in the cool language reference manual
2. **Generate LLVM IR**
    - Translate the AST produced into an LLVM Intermediate representation.
3. **Basic Optimization**
    - Apply a few simple optimization passes to the generated LLVM IR using the **opt** tool.
4. **Binary Generation**
    - Use the LLVM toolchain (llc and a linker like gcc or clang) to produce an executable binary from the optimized LLVM IR.
5. **Extensions**
    - Implement some extensions to the Cool language. Details about extensions below.

Project Objectives

The main objectives of this project are to:

- Deepen understanding of compiler design principles.
- Gain hands-on experience in implementing various phases of a compiler.
- Develop proficiency in the Go programming language (or an approved alternative).
- Enhance problem-solving and software engineering skills.
- Learn more about LLVM.

Project Requirements

The Cool compiler must be able to:

- **Lexical Analysis:** Read Cool source code and produce a stream of tokens.
- **Syntax Analysis:** Parse the token stream and generate an Abstract Syntax Tree (AST).
- **Semantic Analysis:** Check the AST for semantic errors and enforce type safety.
- **IR generation:** Generate LLVM IR
- **Code Generation:** Translate the LLVM IR into executable code (e.g., assembly or machine code).

Implementation Details

Programming Language

- **Primary Language:** Go
- **Alternative Language:** Students may request to use a different language, providing a strong justification and obtaining approval from the instructor.

Testing and Evaluation

Students are expected to:

- Write comprehensive test cases to cover various aspects of the compiler.
- Thoroughly test the compiler's functionality and correctness.
- Evaluate the compiler's performance and efficiency.

Compiler extensions

Extension1: Array Support

The goal is to extend the cool language to support dynamically sized arrays.

Syntax

```
Unset
-- Declaration (with initialization)
myArray : Array[Int] <- new Array[Int](10); -- An array of 10 integers

-- Access myArray[0] <- 5;
x : Int <- myArray[5];

 -- Size
size : Int <- myArray.size();
```

Semantics:

- **Type System**: Introduce a new Array[T] type to the Cool type system, where T can be any valid Cool type. This means that arrays should be able to hold elements of any single type. Decide whether the size of the array is part of its type (like in some languages or not). You need to ensure that array initialization is of the correct type.
- **Sizing:** You can start with a statically sized array, but ideally the array should be dynamically sized.
- **`size()` Method:** Implement a built-in method `size()` that returns the number of elements in an array.
- **Bounds checking:** Implement a runtime bound checking to prevent accessing elements outside the array's bounds.

Code Generation:

- **Allocation:** Generation LLVM IR code to dynamically allocate memory of arrays on the heap (using `malloc` or similar function form your runtime library). The size should be determined by the expression provided in the allocation.
- **Access:** Generate code to access array elements using appropriate LLVM instructions for pointer arithmetics (e.g, `getelementptr`).

- **Bound checking:** Generate code to check if the index is within the valid bounds of the array before each access. If an out-of-bounds access is detected, call an error function in your runtime library to terminate the program.

Runtime Library:

You might need to add functions to your runtime library to support array allocation, deallocation, and error handling…

Extension2: Standard Library: Set or Linked List

Goal

Implement either a Set or a LinkedList data structure in Cool.

Data Structure Choice:

- Set
  - A collection of unique elements (no duplicates).
  - Operations: `add(element)`, `remove(element)`, `contains(element)`, `size()`, `isEmpty()`.
- LinkedList
  - A linear data structure where elements are linked together using pointers.
  - Operations: `addFirst(element)`, `addLast(element)`, `removeFirst()`, `removeLast()`, `get(index)`, `size()`, `isEmpty()`.

Implementation

- **Objects**: Define appropriate classes and methods in Cool to implement the chosen data structure. For example, a LinkedList could have a Node class with data and next attributes.

Integration

Make your data structure available as a standard library that can be used by other Cool programs.

Extension3: Module System

Design and implement a simple module system for Cool.

Syntax Example

Below is just an example of a module system. You could use keywords like `module` and `import`:

```
Unset
-- File: mymodule.cl
module MyModule;

class MyClass {
    -- ...
};
```

```
Unset
-- File: main.cl
import MyModule;

class Main {
    main() : Object {
        let obj : MyModule.MyClass <- new MyModule.MyClass in
        -- ...
    };
};
```

Semantics

- **Module Resolution:** Define how modules are found and loaded based on their names (e.g., using file paths relative to the main file, a module search path, or some other convention).
- **Name Access:** Imported names should be accessed using a qualified name (e.g., `ModuleName.ClassName`, `ModuleName.methodName`).
- **Circular Dependencies:** Detect and report an error if there are circular dependencies between modules.

Implementation:

- **Lexer/Parser:**
    - Modify the lexer to recognize any new keywords (e.g., `module`, `import`).
    - Modify the parser to parse module declarations and import statements.

    - Add new AST nodes to represent modules (e.g., `ModuleDeclaration`, `ImportStatement`
- **Semantic Analysis**
    - **Module Loading:** Implement a mechanism to load and parse imported modules when they are encountered. You might need to maintain a cache of parsed modules to avoid re-parsing them.
    - **Name Resolution:** During type checking, resolve qualified names (like `MyModule.MyClass`) by looking up the module and then the corresponding entity within that module.
    - **Symbol Table:** Adapt the symbol table to handle modules. You might introduce a hierarchical structure to the symbol table, where each module has its own symbol table nested within the global symbol table. Or, you could use a prefixing scheme to store module entities in the global symbol table.
    - **Circular Dependency Detection:** Implement an algorithm (e.g., using depth-first search) to detect circular dependencies between modules.
- **Code Generation:**
    - **Option 1 (Concatenation):** A simple approach is to concatenate the code of all imported modules into a single LLVM module. This might require renaming symbols to avoid conflicts.
    - **Option 2 (Separate Modules):** Generate separate LLVM modules for each Cool module. You would then need to link these modules together in the final binary generation step. This is more complex but allows for separate compilation of modules.

Grading

Your project will be graded based on the following criteria, totaling 6 % for the core requirements and up to 40% extra credit for extensions:

- **Functionality (40%):** Correctness and completeness of the compiler implementation.
- **Code quality (20%):** Is the code well structured, readable, and maintainable? Are there comments explaining the design choices?.
- **Error Handling (10%):** Does the compiler gracefully handle errors in the input program (lexical, syntax, semantic errors)? Does it provide helpful error messages?
- **Testing (10%):** Are there test cases that demonstrate the compiler's functionality and error handling?
- **Documentation (10%):** Is there a README file that explains how to build and run the compiler, describes the language it supports, and discusses any design decisions or limitations?
- **Creativity/Extra Features (10%):** Does the project go beyond the basic requirements? Are there any innovative features or optimizations?

Core requirements (60% of the total grade):

Phase 1: Lexical Analysis (15%):

- **Correctness (10%):** The lexer must correctly tokenize Cool programs, recognizing all keywords, operators, identifiers, literals, and handling comments and whitespace as defined in the Cool Manual. It should pass a comprehensive suite of test cases.
- **Error Handling (5%):** The lexer must detect and report lexical errors with accurate line and column numbers, including informative error messages for invalid tokens, unterminated strings, etc.

Phase 2: Syntax Analysis (Parsing) (25%):

- **Correctness (15%):** The parser must correctly parse all valid Cool programs, building an Abstract Syntax Tree (AST) that accurately reflects the program's structure. It should pass a comprehensive suite of test cases. Operator precedence and associativity must be handled correctly using Pratt parsing.
- **AST Construction (5%):** The generated AST must be well-formed and adhere to the specified AST node structure.
- **Error Handling (5%):** The parser must detect and report syntax errors with clear messages and accurate locations (line and column numbers).

Phase 3: Semantic Analysis (25%):

- **Symbol Table Construction (10%):** The symbol table must be correctly constructed, accurately reflecting the scope of all program entities (classes, attributes, methods, formal parameters, `let` variables).
- **Type Checking (10%):** All expressions and assignments must be type-checked according to Cool's type system, including type conformance and inheritance rules. Type errors must be reported with informative messages and locations.
- **Inheritance Graph and Other Checks (5%):** The inheritance graph must be validated (no cycles, valid parent classes), and other semantic checks (e.g., `Main` class and method, duplicate definitions, `self` usage, `case` branch type uniqueness) must be performed correctly.

Phase 4: LLVM IR Code Generation (25%):

- **Correctness (20%):** The generated LLVM IR code must be valid and correctly implement the semantics of all core Cool features. The generated IR should pass a suite of test cases (that will need to be compiled to an executable and run to verify their behavior).
- **Completeness (5%):** Code generation must be implemented for all core Cool features listed in Section 3.4 of the project description.

Phase 5: Basic Optimization and Binary Generation (10%):

    - **Optimization (5%):** At least two LLVM optimization passes (using `opt`) must be applied and explained in the report. The impact of these passes should be demonstrated or discussed, even if minimal on simple test programs.
    - **Binary Generation (5%):** An executable binary must be successfully generated using `llc` and a linker (e.g., `gcc` or `clang`). The binary should run and produce the correct output for simple Cool programs. You should provide a basic `Makefile` or set of instructions to do this.

Extensions (40% of the total grade)

Each extension, if implemented, is worth a certain % of the total grade. You can choose to implement any combination.

- **Array Support (15%):**
    - **Correctness (8%):** The implementation must correctly handle array declarations, initializations, element access, assignment, and the `size()` method. Runtime bounds checking must be implemented and function correctly.
    - **Completeness (4%):** All aspects of array support (syntax, semantics, code generation) must be fully implemented.
    - **Integration (3%):** Array support should be seamlessly integrated into the existing language and compiler.
- **Standard Library: Set or Linked List (10%):**
    - **Correctness (6%):** The chosen data structure (Set or Linked List) must be implemented correctly in Cool, and all operations must function as specified.
    - **Completeness (4%):** All required operations for the chosen data structure must be implemented..
- **Module System (15%):**
    - **Correctness (8%):** The module system should correctly handle module imports, name resolution, and prevent circular dependencies.
    - **Design (4%):** The design of the module system should be well-reasoned and clearly explained in the report.
    - **Integration (3%):** The module system should be well-integrated into the existing compiler phases (parsing, semantic analysis, code generation).

Additional Notes

- This is an individual project.
- Consult the instructor for any questions or clarifications.