COOL Compiler Project

Welcome to the COOL Compiler Project – an implementation of a complete compiler for the Classroom Object-Oriented Language (COOL). This project was developed out of curiosity about how programming languages work and is implemented in Go. It supports all the core phases of compilation – from lexical analysis to LLVM IR generation, optimization, and binary creation.

Table of Contents

1. Overview
2. Project Structure
3. Features
4. Compilation Phases
    - Lexical Analysis
    - Parsing
    - Importing modules
    - Semantic Analysis
    - IR Generation & Method Generation
    - Optimizations & Binary Generation
    - LLVM IR Optimizations
5. Context-Free Grammar (CFG)
6. Building & Usage
7. Testing
8. Makefile & Integration

Overview

This project implements a COOL compiler that processes COOL source code and generates LLVM Intermediate Representation (IR) code. The compiler supports a set of features including:

- Classes and inheritance
- Methods and attributes (both static and dynamic dispatch)
- Basic types: Int, String, Bool
- Control structures: if/then/else, while, and case expressions
- Let expressions and basic arithmetic operations

Project Structure

The repository is organized into several directories and files that reflect the major components of the compiler:

```
cool-compiler/
├── linker/                 # Imports and modules integration
├── structures/             # Symbol table & AST definitions
├── lexer/                  # Lexical analyzer for tokenization
├── parser/                 # Recursive descent parser
├── semantic/               # Semantic analysis and symbol table construction
├── irgen/                  # LLVM IR generator and method generation (e.g., irGenerator.go)
├── testcodes/              # Sample COOL programs for testing
├── main.go                 # Compiler entry point orchestrating all phases
├── Makefile                # Build automation and integration
├── go.mod                  # Go module files and dependency listings
└── README.md               # This merged documentation file
```

Features

- **Full COOL Manual implementation:**
  Implements classes, inheritance, methods, attributes, and control structures as defined in the COOL reference manual.

- **Robust Lexical and Syntax Analysis:**
  Tokenizes input source code, removes comments, and builds an Abstract Syntax Tree (AST) using a recursive descent parser with Pratt parsing for operator precedence.

- **Semantic Analysis:**
  Performs type checking, symbol table construction, scope resolution, and verifies proper inheritance and method overriding.

- **LLVM IR Code Generation:**
  Translates the AST into LLVM IR code. The IR generator handles class layout, virtual method tables (vtables), object creation, and expression evaluation.

- **Optimizations:**
  Applies LLVM IR optimization passes such as constant propagation and dead code elimination before binary generation.

Compilation Phases

Lexical Analysis

- **Description:**
  The lexer reads the COOL source code character by character, skips whitespace and comments, and produces a stream of tokens.

- **Key Details:**
    - Comment removal (single-line and block comments)
    - Tokenization functions for identifiers, numbers, and strings
- **Relevant Files:** `lexer/lexer.go`, `lexer/remove_comments.go`

Parsing

- **Description:**
  The parser processes the token stream to ensure the code conforms to COOL's context-free grammar and builds an Abstract Syntax Tree (AST).
- **Key Details:**
    - Recursive descent parsing with Pratt parsing to handle operator precedence
    - Error handling with detailed line and column reporting
- **Relevant Files:** `parser/parser.go`, `structures/ast.go`

Importing modules

- **Description:**
  The compiler supports importing COOL source files from the same directory, allowing modular program organization and code reuse.
- **Key Details:**
    - Automatic detection and loading of imported files from the current directory
    - Recursive import resolution while avoiding circular dependencies
    - Validation of imported class names and inheritance relationships
- **Relevant Files:** `linker/linker.go`

Example usage:

```
import module.cl
```

Semantic Analysis

- **Description:**
  Semantic analysis checks the AST for type correctness and builds a symbol table containing all classes, attributes, and methods.
- **Key Details:**
    - Construction of nested scopes for symbol resolution
    - Type checking and inheritance validation
    - Special checks for the presence of the Main class and a no-parameter `main()` method
- **Relevant Files:** `semantic/semantic.go`, `structures/symbol_table.go`

IR Generation & Method Generation

- **Description:**
  The IR generator translates the AST into LLVM IR code. It converts COOL classes into LLVM struct types and builds virtual tables for dynamic dispatch.
- **Key Details:**
  - Generation of LLVM IR for user-defined classes
  - Management of temporary registers and string constants
  - Generation of method bodies using pre-computed method signatures
- **Relevant Files:** `irgen/irGenerator.go`

Optimizations & Binary Generation

- **Description:**
  The generated LLVM IR is optimized using standard LLVM passes (e.g., constant propagation and dead code elimination). The optimized IR is then compiled into an executable binary using the LLVM toolchain (llc, opt) and a linker (gcc or clang).
- **Key Details:**
  - LLVM IR optimization passes
  - Integration with the system's LLVM toolchain

LLVM IR Optimizations

- **Overview:**
  - **Constant Propagation:** Precomputes expressions at compile time.
  - **Dead Code Elimination:** Removes redundant or unused code.

These optimizations help reduce runtime workload and improve performance.

Context-Free Grammar (CFG)

The COOL language grammar defines the structure of valid programs. A simplified version of the grammar is as follows:

```
<program>          ::= <class-list>

<class-list>       ::= <class> <class-list>
                     | <class>

<class>            ::= "class" <TYPEID> [ "inherits" <TYPEID> ] "{" <feature-list> "}"

<feature-list>     ::= <feature> ";" <feature-list>
                     | ε

<feature>          ::= <attribute>
                     | <method>

<attribute>        ::= <OBJECTID> ":" <TYPEID> [ "<-" <expr> ]

<method>           ::= <OBJECTID> "(" <formal-list> ")" ":" <TYPEID> "{" <expr> "}"

<formal-list>      ::= <formal> <formal-list-rest>
                     | ε

<formal-list-rest>::= "," <formal> <formal-list-rest>
                     | ε

<formal>           ::= <OBJECTID> ":" <TYPEID>

<expr>             ::= <OBJECTID> "<-" <expr>
                     | <expr> "@" <TYPEID> "." <OBJECTID> "(" <expr-list> ")"
                     | <expr> "." <OBJECTID> "(" <expr-list> ")"
                     | "if" <expr> "then" <expr> "else" <expr> "fi"
                     | "while" <expr> "loop" <expr> "pool"
                     | "let" <let-binding> { "," <let-binding> } "in" <expr>
                     | "case" <expr> "of" <case-branch-list> "esac"
                     | "new" <TYPEID>
                     | "isvoid" <expr>
                     | <expr> "+" <expr>
                     | <expr> "-" <expr>
                     | <expr> "*" <expr>
                     | <expr> "/" <expr>
                     | "~" <expr>
                     | <expr> "<" <expr>
                     | <expr> "<=" <expr>
                     | <expr> "=" <expr>
                     | "not" <expr>
                     | "(" <expr> ")"
```

```
                    | <OBJECTID>
                    | <INT_CONST>
                    | <STR_CONST>
                    | "true"
                    | "false"

<let-binding>     ::= <OBJECTID> ":" <TYPEID> [ "<-" <expr> ]

<case-branch-list>::= <case-branch> { ";" <case-branch> }

<case-branch>     ::= <OBJECTID> ":" <TYPEID> "=>" <expr>

<expr-list>       ::= <expr> { "," <expr> }
```

For a full description, please refer to the COOL reference manual.

Building & Usage

Prerequisites

- Go 1.23.4 or higher
- LLVM toolchain (llc, opt, clang)
- Make

Building

1. **Clone the repository:**

   ```
   git clone <this repository url>
   cd cool-compiler
   ```

2. **Build the compiler:**

   ```
   make all I=pathtocoolfile [O=pathtooutputbinary]
   ```

Usage

Run the compiler with:

```
./cool-compiler -i <input-file.cl> [-d]
```

Options:

- `-i`: Path to the input COOL source file
- `-d`: Enable debug output (optional)

Example:

```
./cool-compiler -i hello_world.cl
```

This generates a `.out.ir` file containing the LLVM IR code.

Testing

The project includes a comprehensive test suite:

- **Testing error detection:**
  Located in the `tests/errors_detection` directory, covering lexical analysis, parsing, semantic analysis, and code generation.

  ```
  python3 test_compiler.py
  ```

  use that python script for automated batch testing

- **Writing New Tests:**
  Create new `.cl` file and enjoy coding.

Makefile & Integration

A Makefile is provided to automate the build process:

```
I ?= tests/semantictest.cool
O ?= cool_program

CC = clang
LLC = llc
OPT = opt
GO_BUILD = go build -o cool-compiler
COMPILER = ./cool-compiler

build:
	$(GO_BUILD)

ir: build
	$(COMPILER) -i $(I)

opt: ir
	$(OPT) -passes=mem2reg,dce .out.ir -o .out.opt.ll

asm: opt
	$(LLC) .out.opt.ll -o .out.s

link: asm
	$(CC) -no-pie .out.s -o $(O)

clean:
	rm -f cool-compiler .out.ir .out.opt.ll .out.s

all: link clean
```

This file compiles the compiler, generates LLVM IR, applies optimizations, and finally produces an executable binary.