ECOLE POLYTECHNIQUE DE BRUXELLES

2025-2026

INFO-F403

yaLcc project : part 3

Bataille Alex

Gilles Geeraerts

Tajani Mohamed

Arnaud Leponce

2025

Table des matières

1 Introduction

In this final phase of the project, we implemented the `LLVMGenerator` class. Its role is to translate the abstract syntax tree (represented as a `ParseTree`) into LLVM Intermediate Representation (IR), adhering to the informal semantics of the YALCC language. The generated code is fully compatible with `llvm-as` and can be executed using `lli`.

2 Implementation

This section details the strategic choices made to satisfy the project's technical requirements.

2.1 Architectural Choices and Justifications

— **Stack-Based Variable Management** (`alloca`) **:** As required by the semantics, variables are stored in memory locations rather than simple LLVM virtual registers. We manage all local variables using the `alloca` instruction.
Justification : This ensures compliance with the requirement that assignments store values in memory and simplifies the handling of variable updates across different basic blocks.

— **Recursive Parse Tree Traversal :** The generator employs a top-down recursive traversal.
Justification : This approach directly mirrors the LL(1) grammar, ensuring that the hierarchy of operations is strictly preserved during code generation.

— **Global and Local Management via** `StringBuilder` **:** We use separate `StringBuilder` instances for global declarations (external functions, format strings) and the `main` function body.
Justification : This allows the compiler to inject necessary external declarations (like `printf` or `scanf`) dynamically as they are encountered in the source code.

2.2 Expressions, Priority, and Associativity

Special attention was paid to the implementation of arithmetic and boolean expressions :

— **Operator Precedence :** The traversal follows the priorities defined in the project specifications, from unary operators to the implication operator.

— **Right-Associativity :** For operators like the unary minus and the implication (`->`), the generator ensures right-to-left evaluation to match the required semantics (e.g., `1->T->` evaluating to `T`).

— **Boolean Logic :** Comparisons (`==`, `<=`, `<`) generate `i1` results via `icmp`, while the implication is implemented following its specific truth table.

2.3 Implementation of Control Flow

The implementation of `If` and `While` structures follows the required sequential and conditional semantics :

— **Unique Labeling :** A `labelCounter` generates unique identifiers for branching (e.g., `if_then`, `if_else`, `while_cond`).

— **Branching Strategy :** Conditions are evaluated to an `i1`, followed by a `br` instruction to redirect execution flow according to the YALCC logic.

2.4 Standard I/O Strategy

Input and output operations leverage the standard C library :

— **Print :** Translated to `printf` calls using a predefined `"%d\n"` format string.

— **Input :** Translated to `scanf` calls, storing the read integer directly into the variable's memory location.

2.5 Hypotheses and Constraints

1. **Type Uniformity :** All variables are handled as 32-bit signed integers (`i32`).
2. **Implicit Declaration :** Following the project's flexible nature, variables are allocated upon their first occurrence.
3. **Single Scope :** All variables are treated as local to the `main` function.

3 Test Files

The following files demonstrate the correctness of the compiler across the whole YALCC language.

3.1 Euclid.ycc

Tests nested `While` loops, input/output, and the handling of both types of comments.

3.2 check_positive.ycc

Validates the `If-Then-Else` structure and basic comparisons.

3.3 plus_minus_game.ycc

A complex test case involving interactive I/O, nested conditionals, and counter management, proving the robustness of the control flow.

3.4 calculator.ycc

Specifically tests operator precedence and the integration of multiple arithmetic operations in a single program.

4 Conclusion

This project allowed us to implement a complete compilation pipeline from YALCC to LLVM IR. By strictly following the requirements for operator associativity, memory-based variable storage, and standard I/O integration, we have produced a compiler that generates robust and executable machine code. The modular design of the `LLVMGenerator` ensures that it correctly handles the language's semantics while remaining extensible for future improvements.