8bit-Compiler

A compiler for SimpleLang

Shankhadeep Mandal

Architecture

We develop a simple 3 pass compiler for each of the three phases, using C++20 features.

1. The key entrypoint is `src/main.cpp` where all the three passes are integrated.
2. The lexer interface is defined in a header file.

Almost all code is developed in `::impl` namespaces for easier namespace hiding of undesired namespaces (e.g, `std::`), this also allows for anonymous namespace inlining if desired.

The AST

1. Several AST designs were evaluated on the basis of their *pros* and *cons* however to keep debugging simplistic and avoid templates [1], the final design of **enum tags + virtual functions + inheritance** was chosen.
2. Other AST designs considered were: purely enum tag based, discriminated union based, and `std::variant` based designs.
3. The ast divides each construct into: **statements** and **expressions**.
4. Concrete constructs are derived from the two where statements provide wrappers above expression for easier derivations.

[1] templates hinder IDE auto completions, although due to `concepts` the error handling is a lot better

The Parser

1. The parser does recursive descent and based on locality and goes for a longest common match allowing for complex expressions like

$$a = 2 + 3 - a + b;$$

2. The key entrypoint is `parse_context()` method which makes the root node `context` and then allows other statements to be attached as nodes.
3. The integers are chosen to be 8-bit wide. (Implementation value type is `int8_t`.
4. Other parser designs considered was a **PEG** based parser (based on the excellent **cpp-peglib**) and a Pratt Parser.

CodeGen & Further

- While the codegen core functionality is in place, the register allocation methods (along with memory-data storage) implementation is due.
- Small semantic checks are in place to ensure **correctness** as well as **minor storage safety** (if variable is declared before or not), null safety (liberal asserts are applied to make sure no node is null).
- CodeGen does a *R-L* style code generation so that ast nodes like `infix_expr` can deduce the r,l-value and generate accordingly.
- Parser exposes global `error` methods for appropriate error reporting and termination.

Conclusion and Further Work

Further work:

1. The first task would be to complete the code gen.
2. The ast can be further modularised via the **visitor** pattern to separate codegen logic.
3. The assembler can be integrated directly to compile directly.

Conclusion: This document explains the design choices and some of the architecture developed for the implementation of the compiler.

Thank You.