Relatório de Análise do Projeto *compiladores-minipar*

Resumo Executivo

O repositório ***compiladores-minipar*** consiste em um **compilador didático** desenvolvido majoritariamente em **C++**, cobrindo todas as etapas clássicas de compilação: **análise léxica**, **análise sintática**, **geração de árvore sintática abstrata (AST)**, **geração de código intermediário (TAC — Three Address Code)** e **geração final de código assembly ARMv7**.

Além da versão **CLI (linha de comando)**, o compilador também possui uma **interface web** desenvolvida em **React + TypeScript**, que interage com o compilador compilado para **WebAssembly (via Emscripten)**.

O sistema suporta elementos básicos de linguagens imperativas e concorrentes: **blocos SEQ/PAR**, **operações aritméticas**, **comandos de controle (while, if)**, **entrada/saída (print, input)** e **canais**.

Estrutura do Repositório

Mapa de Diretórios (alto nível)

- **compilador/** — diretório binário (não lido)
- **include/** — cabeçalhos públicos do compilador
  Contém: `lexer.h`, `parser.h`, `ast_nodes.h`, `ast_printer.h`, `symbol_table.h`, `tac_generator.h`, `arm_generator.h`, `emscripten_interface.h`
- **src/**
  - **frontend/** — implementação do **lexer**, **parser** e **AST**
    - `lexer.cpp`
    - `parser.cpp`
    - `ast_nodes.cpp`, `ast_printer.cpp`
  - **middle end/** — tabela de símbolos e geração de TAC
    - `symbol_table.cpp`
    - `tac_generator.cpp`

    - **backend/arm/** — geração de assembly ARMv7
        - `arm_generator.cpp`
    - **runtime/** — contém README explicando o runtime
    - `emscripten_interface.cpp`, `main_emscripten.cpp` — integração com WASM
    - `main.cpp` — runner CLI (fluxo completo)
- **testes/** — exemplos `.minipar` (ex.: `teste_simples.minipar`, `fatorial.minipar`, `paraleo.minipar`)
- **web/react/** — interface web (Vite + React + TypeScript)
  Contém `package.json`, `CompilerContext.tsx`, `buildArtifacts.ts`, `App.tsx`, etc.

Fluxo de Compilação

1. **Entrada**
   O código-fonte `.minipar` é lido via **CLI** (`main.cpp`) ou pela função exportada do **WASM** (`compile_minipar`).
2. **Análise Léxica (Lexer)**
    - Identifica **palavras-chave** (`SEQ`, `PAR`, `print`, `input`, `while` etc.), **identificadores**, **números**, **strings**, **operadores** e **delimitadores**.
    - Ignora comentários (`# até fim da linha`) e contabiliza posição (linha/coluna).
    - Implementação em `lexer.cpp` e `lexer.h`.
3. **Análise Sintática (Parser)**
    - Baseada em **análise recursiva descendente**.
    - Constrói uma **ProgramNode** que contém **SeqNode**, **ParNode** e demais nós da AST.
    - Reconhece **atribuições**, **print**, **input**, **while**, **comparações**, **operações aritméticas**.
    - Implementação em `parser.cpp` e `parser.h`.
    - Observação: `parse_while_statement()` possui heurística simples, não interpretando corretamente blocos `{}` aninhados.

4. **Construção da AST (Abstract Syntax Tree)**
    - Estruturas de nós: `ProgramNode`, `SeqNode`, `ParNode`, `AssignmentNode`, `PrintNode`, `InputNode`, `WhileNode`, `BinaryOpNode`, etc.
    - Impressão formatada via `ASTPrinter` (`ast_printer.cpp`).
    - Implementação em `ast_nodes.cpp` e `ast_nodes.h`.
5. **Tabela de Símbolos / Análise Semântica**
    - Implementação simples em `symbol_table.cpp` / `symbol_table.h`.
    - Gerencia variáveis e funções, mas ainda sem verificação completa de tipos ou escopos.
6. **Geração de Código Intermediário (TAC)**
    - Cada operação é traduzida em instruções de três endereços (`arg1 op arg2 → result`).
    - Implementa **labels** e **saltos condicionais** para loops e ifs.
    - Exemplo:

7. **Geração de Código Assembly (ARMv7)**
    - Tradução direta do TAC para instruções ARM (`add`, `sub`, `cmp`, etc.).
    - Utiliza registradores `r0`–`r6` com alocação sequencial.
    - Saída com `printf` e término com syscall (`svc #0`).
    - Implementação em `arm_generator.cpp` e `arm_generator.h`.
8. **Integração WebAssembly (Emscripten)**

- `emscripten_interface.cpp` exporta:
    - `compile_minipar()` — retorna saída textual.
    - `compile_minipar_json()` — retorna JSON com tokens, AST, TAC e ARM.
    - `free_string()` — libera memória alocada.
  - `CompilerContext.tsx` e `buildArtifacts.ts` no React processam a saída e exibem resultados na interface.

9. **Execução via CLI**

- Arquivo: `main.cpp`.
- Executa todas as etapas e imprime o resultado no terminal.

Pontos Fortes

- Implementa **todo o pipeline de compilação** de forma didática e modular.
- Código **organizado por camadas** (frontend, middleend, backend).
- AST e impressor (`ASTPrinter`) bem estruturados e úteis para depuração.
- Integração **Emscripten + React** demonstra aplicação prática e interativa.
- Saída **JSON** via `compile_minipar_json` facilita integração com interfaces e testes.
- Código **legível e comentado**, adequado para fins educacionais.

Limitações e Riscos Identificados

Parser e Blocos

- `parse_while_statement()` e `parse_seq_block()` utilizam heurísticas frágeis para delimitar blocos.
- Não há suporte robusto para `{}` ou blocos aninhados.

Tabela de Símbolos / Semântica

- `SymbolTable` é pouco utilizada ao longo do fluxo.
- Falta **checagem de tipos**, **escopos** e **declarações duplicadas**.

Geração de TAC / Assembly

- Alocação de registradores é **muito simples** e não suporta programas extensos.
- Uso de `printf` em assembly mistura camadas de abstração.
- Ausência de **otimizações** (ex.: eliminação de código morto, constant folding).
- Pequenos problemas no código: retorno duplicado em `TACGenerator::generate()`, verificações frágeis de strings.

Integração Web

- Uso de `new char[] + strcpy` pode ser inseguro para strings com caracteres nulos.
- Necessita melhoria na gestão de memória entre C++ e JavaScript.

Tratamento de Erros

- Mensagens genéricas e pouco informativas.
- Falta de localização precisa de erro (linha e coluna).

Concorrência e Canais

- A sintaxe suporta `PAR` e canais (`send`, `receive`), mas a **execução paralela não é implementada** (tratada de forma sequencial).

Sugestões de Melhoria

Prioridade Alta

1. Implementar parsing robusto de blocos `{}` e corrigir corpo de `while`.
2. Melhorar mensagens de erro com **linha/coluna/token**.

3. Integrar a `SymbolTable` ao parser e adicionar **checagem semântica**.
4. Corrigir gestão de memória para strings no Emscripten (usar `emscripten::val` ou `Module.allocateUTF8`).
5. Corrigir redundâncias e pequenos bugs (retornos duplicados, checagens frágeis).
6. Criar **testes unitários e de integração** (lexer, parser, TAC, CLI, WASM).

Prioridade Média

1. Implementar alocação de registradores mais inteligente (ex.: algoritmo de coloração de grafo).
2. Adicionar passes de otimização simples (constant folding, DCE).
3. Documentar a **gramática formal (BNF/EBNF)** da linguagem.

Prioridade Baixa

1. Expandir `compile_minipar_json` para incluir AST serializada em JSON.
2. Melhorar interface web com **AST navegável e destaque de tokens**.
3. Adicionar **CMakeLists.txt** e pipeline de CI (GitHub Actions).

Conclusão

O projeto **compiladores-minipar** é uma implementação didática sólida de um pipeline completo de compilador, cobrindo os principais conceitos de **Compiladores** e **Sistemas de Linguagens de Programação**.
Apesar de limitações pontuais em parsing e alocação de registradores, apresenta **estrutura modular, clareza e integração prática via WebAssembly**, tornando-se uma excelente base de ensino e extensão para futuras melhorias.