Micro-compilador em C++

UNISINOS
Nossa sala de aula é o mundo.

Suelen Fraga

Linguagem Alvo e Objetivos

Mini-linguagem:

- Tipos: int, float, string, bool
- Estruturas: if/else, blocos { }
- Operadores: aritméticos, relacionais, lógicos
- Expressões com precedência real

Objetivo do projeto:

Implementar um pipeline completo:

Léxico → Sintaxe/Gramática → AST → Semântica → Execução

Léxico

Tokens principais:

- Identificadores
- Palavras-chave
- Números (inteiros e decimais)
- Strings
- Operadores (+ - * / = == != <= >= && ||)
- Pontuação (; , ( ) { } [ ])
- Comentários de linha (// ...)

Recursos implementados:

- Rastreamento completo de **linha** e **coluna**
- Detecção de **UNKNOWN**, strings mal formadas
- Saída modo CLI: --tokens

Exemplo de Tokens

--tokens testesimples.txt

```
//teste simples
int x = 42;
string s = "Olá";
if (x > 10 && s != "") {
    x = x + 1;
}
```

- Cada token é mostrado no formato: **TIPO → "lexema" [linha,coluna].**
- O analisador reconhece comentários, operadores compostos (&&, !=), strings, números, pontuação e palavras-chave.

Saída do Analisador

```
COMMENTARIO -> "// teste simples" [1,1]
KEYWORD -> "int" [2,1]
IDENTIFICADOR -> "x" [2,5]
OPERADOR -> "=" [2,7]
NUM_INT -> "42" [2,9]
PONTUACAO -> ";" [2,11]
KEYWORD -> "string" [3,1]
IDENTIFICADOR -> "s" [3,8]
OPERADOR -> "=" [3,10]
STRING -> ""Olá"" [3,12]
PONTUACAO -> ";" [3,18]
KEYWORD -> "if" [4,1]
PONTUACAO -> "(" [4,4]
IDENTIFICADOR -> "x" [4,5]
OPERADOR -> ">" [4,7]
NUM_INT -> "10" [4,9]
OPERADOR -> "&&" [4,12]
IDENTIFICADOR -> "s" [4,15]
OPERADOR -> "!=" [4,17]
STRING -> """" [4,20]
PONTUACAO -> ")" [4,22]
PONTUACAO -> "{" [4,24]
IDENTIFICADOR -> "x" [5,5]
OPERADOR -> "=" [5,7]
IDENTIFICADOR -> "x" [5,9]
OPERADOR -> "+" [5,11]
NUM_INT -> "1" [5,13]
PONTUACAO -> ";" [5,14]
PONTUACAO -> "}" [6,1]
FIM DE ARQUIVO -> "<EOF>" [6,2]
```

Gramática e Parser (AST)

Define as estruturas essenciais da linguagem:

decl -> (int | float | string | bool) IDENT ('=' expr)? ';'
stmt -> decl | ifStmt | assign | block
ifStmt -> "if" "(" expr ")" stmt ("else" stmt)?
block -> "{" stmt* "}"
assign -> IDENT "=" expr ";"

Expressões com precedência: or -> and -> equality -> rel -> add -> mult -> primary;
primary -> IDENT | NUM_INT | NUM_REAL | STRING | true/false | "(" expr ")"

Exemplo de ast

--ast testesimples.txt

```
//teste simples
int x = 42;
string s = "Olá";
if (x > 10 && s != "") {
    x = x + 1;
}
```

- Parser recursivo constrói AST com nós: Program, Block, Decl, Assign, If, Binary, Literal, Identifier.
- Raiz Program com filhos Decl/Assign/If; nós Binary representam operações; Literals e Identifiers são folhas.

Saída do Analisador

```
Decl : "x" [2,1]
        Identifier : "x" [2,5]
        Literal : "42" [2,9]
 Decl : "s" [3,1]
        Identifier : "s" [3,8]
         Literal : ""olái"" [3,12]
 If : "if" [4,1]
        Binary : "&&" [4,12]
                Binary : ">" [4,7]
                        Identifier : "x" [4,5]
                        Literal : "10" [4,9]
                Binary : "!=" [4,17]
                        Identifier : "s" [4,15]
                        Literal : """" [4,20]
        Block : "block" [4,24]
                Assign : "=" [5,5]
                        Identifier : "x" [5,5]
                        Binary : "+" [5,11]
                                Identifier : "x" [5,9]
                                Literal : "1" [5,13]
[Erro semantico] comparacao '!=' exige operandos numericos (4,17)
```

Semântica (checagens de tipos e uso)

```cpp
// Definição dos tipos de dados
enum class TypeKind {
    INT,
    REAL,
    STRING,
    BOOL,
    UNKNOWN
};

// Estrutura de erro semântico
struct SemanticError {
    std::string message;
    int linha;
    int coluna;
};
```

- Tabela de símbolos (nome → tipo) no checker;

  tipos: int, float, string, bool.

- Regras principais:
  - Variável deve ser declarada antes de usar; redeclaração acusa erro.
  - Atribuição e inicialização: tipo do RHS deve ser compatível (promoção int → float permitida; demais incompatibilidades geram erro).
  - Condição de if deve ser bool; operandos de + - * / % devem ser numéricos.
- Mensagens claras com linha/coluna/lexema, ex.: variável 'a' usada sem declarar (2,1); tipos incompatíveis na atribuição....

Executor (modo --run)

- **Interpreta a AST em memória:** ambiente de variáveis com tipo e valor.
- **Suporta:** atribuições, aritmética e comparação (com promoção int → float), lógica (&&, ||), e if/else com blocos.
- **Fluxo:** parse → checagem semântica (se houver erros, aborta) → execução → imprime estado final das variáveis.
- Implementação em exec/exec.cpp, chamada pelo CLI quando a semântica não aponta erros.

```
//teste simples
int x = 42;
string s = "Olá";
if (x > 10 && s != "") {
    x = x + 1;
}
```

```
[Erro semantico] comparacao '!=' exige operandos numericos
```

```
// caso válido: declarações e if/else simples
int a;
float b;
a = 1;
b = a + 2.5;
if (a < b) {
  a = a + 1;
} else {
  b = b - 1;
}
```

```
a = 2
b = 3.5
```

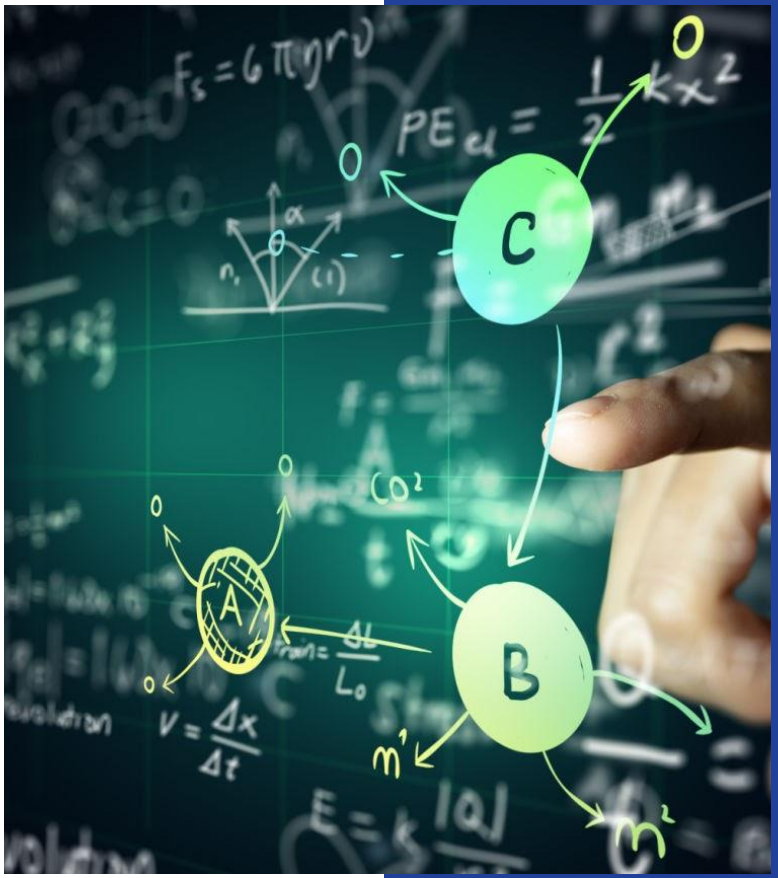
CLI, Build e Organização

- **Build:** nmake (MSVC, x64 Native Tools Prompt) → gera exec\microcompilador.exe.
- Modos da CLI (main/main.cpp):
  - --tokens <arquivo>: imprime tokens com linha/coluna.
  - --ast <arquivo>: gera AST e checa semântica; mostra erros se houver.
  - --run <arquivo>: só executa se semântica passar; imprime valores finais das variáveis.
- Estrutura de pastas:
  - lexer/, parser/, semantic/, exec/, main/ (fonte do pipeline)
  - tests/ com entradas .txt e expected em tokens_out/, ast_out/, run_out/; script tests/run_tests.ps1 compara saídas.
- Como validar tudo: powershell -ExecutionPolicy Bypass -File tests\run_tests.ps1 → resumo PASS/FAIL e diffs se houver.

Testes e Automação

- Casos de entrada (tests/): **ok_basico.txt**, **err_sintatico.txt**, **err_lexico.txt**, **err_semantico_undeclarado.txt**, **err_semantico_tipo.txt**, **err_semantico_if.txt**.
- Expected separados por modo:
  - tokens_out/ para --tokens
  - ast_out/ para --ast
  - run_out/ para --run
- Script de validação: powershell -ExecutionPolicy Bypass -File tests\run_tests.ps1
  - Roda cada modo em todos os arquivos, compara com os .out e mostra PASS/FAIL.
  - Em caso de diferença, gera .actual para inspecionar.
- Demonstração rápida:
  - --tokens tests/err_lexico.txt → mostra UNKNOWN para o caractere inválido.
  - --ast tests/err_semantico_if.txt → erro de condição não bool.
  - --run tests/ok_basico.txt → estado final a = 2, b = 3.5.

UNISINOS



Limitações e Próximos Passos

- **Escopo atual:** variáveis globais simples, sem escopos aninhados ou laços (while/for ausentes).
- **Executor:** sem entrada/saída de usuário além do estado final; sem funções nem arrays.
- **Léxico/Sintaxe:** foca em if/else e expressões; gramática não cobre loops ou estruturas avançadas.
- **Próximos passos:** adicionar laços (while), escopos aninhados, mais tipos/estruturas; melhorar mensagens de erro/acentuação; ampliar suite de testes e casos de execução.