Relatório do Projeto de Compiladores

Construção de compiladores

Discente: Assis Thiago;

Docentes: Danilo Teixeira, Eduardo Rocha, Lucas Mourato e Matheus Fabiano;

1. Introdução

O projeto implementa um analisador léxico (Lexer) e um analisador sintático (Parser) para interpretar um subconjunto de uma linguagem de programação fictícia. O fluxo básico do programa é:

1. **Análise Léxica**: Quebra o código-fonte em tokens.
2. **Análise Sintática**: Constrói uma estrutura em árvore (AST) com base nas regras gramaticais.
3. **Execução**: O código utiliza um exemplo de entrada, analisa-o, e imprime os tokens gerados e a AST correspondente.

O código suporta declarações de variáveis, atribuições e expressões aritméticas básicas.

---

2. Análise Léxica

O **Lexer** é responsável por transformar o código-fonte em uma sequência de tokens. Cada token representa uma unidade lexical, como palavras-chave, identificadores, números, operadores ou delimitadores.

Principais Funcionalidades:

1. **Avanço de Caracteres (avancar)**:
    - Atualiza o caractere atual do código-fonte.
2. **Ignorar Espaços (ignorar_espacos)**:
    - Remove espaços em branco e tabulações.

3. **Ignorar Comentários (ignorar_comentarios)**:
    - Suporta comentários de linha (//) e de bloco (/* ... */).
4. **Geração de Tokens (proximo_token e gerar_tokens)**:
    - Reconhece:
        - Palavras-chave, como int, float, char.
        - Identificadores (ex.: nomes de variáveis).
        - Números inteiros e decimais.
        - Operadores e símbolos (ex.: +, -, =, ;, {, }).

Exemplo de Saída:

Com o código-fonte:

```
int x = 10;
x = x + 5;
```

O Lexer gera os tokens:

```
Token(TIPO, 'int')
Token(ID, 'x')
Token(EQUALS, '=')
Token(NUM_INT, '10')
Token(SEMICOLON, ';')
Token(ID, 'x')
Token(EQUALS, '=')
Token(ID, 'x')
Token(PLUS, '+')
Token(NUM_INT, '5')
Token(SEMICOLON, ';')
Token(EOF, None)
```

---

3. Análise Sintática

O **Parser** utiliza os tokens gerados pelo Lexer para construir uma Árvore Sintática Abstrata (AST). Essa estrutura representa a hierarquia lógica do programa.

Principais Funcionalidades:

1. **Consumir Tokens (consumir)**:
    - Verifica se o token atual corresponde ao esperado.
    - Avança para o próximo token caso seja válido.
2. **Construção da AST**:
    - **Declaração de Variáveis (declaracao_variavel)**:
        - Representa variáveis com seu tipo, nome e valor inicial (se existir).
    - **Atribuições (declaracao_atribuicao)**:
        - Modela a atribuição de valores a variáveis.
    - **Expressões Matemáticas (expressao)**:
        - Implementa operadores binários como soma e multiplicação.
3. **Detecção de Erros**:
    - Identifica e sinaliza tokens inesperados ou estruturas inválidas.

Exemplo de AST Gerada:

Com o código-fonte na linguagem C :

```
int x = 10;
int y = 5;
x = x + y;
```

A AST gerada será:

```
DeclaracaoVariavel(tipo=int, nome=x, valor=Valor(valor=10))
DeclaracaoVariavel(tipo=int, nome=y, valor=Valor(valor=5))
```

Atribuicao(nome=x, expressao=ExpressaoBinaria(operador=+, esquerda=Valor(valor=x), direita=Valor(valor=y)))

---

4. Classe de Nós da AST

Os elementos da AST são representados por classes específicas:

- **DeclaracaoVariavel**: Representa a declaração de uma variável.
- **Atribuicao**: Modela atribuições de valores a variáveis.
- **ExpressaoBinaria**: Representa operações como soma, subtração, multiplicação e divisão.
- **Valor**: Representa valores constantes (números ou identificadores).

Cada classe implementa o método __repr__ para exibir sua representação textual, facilitando o debug.

---

5. Exemplos de Testes

Código de Entrada :

```
int x = 10;
int y = 5;
int z = 2;
x = x + (y * z);
```

Tokens Gerados:

```
Token(TIPO, 'int')
Token(ID, 'x')
Token(EQUALS, '=')
Token(NUM_INT, '10')
Token(SEMICOLON, ';')
Token(TIPO, 'int')
Token(ID, 'y')
```

Token(EQUALS, '=')
Token(NUM_INT, '5')
Token(SEMICOLON, ';')
Token(TIPO, 'int')
Token(ID, 'z')
Token(EQUALS, '=')
Token(NUM_INT, '2')
Token(SEMICOLON, ';')
Token(ID, 'x')
Token(EQUALS, '=')
Token(ID, 'x')
Token(PLUS, '+')
Token(LPAREN, '(')
Token(ID, 'y')
Token(MULTIPLY, '*')
Token(ID, 'z')
Token(RPAREN, ')')
Token(SEMICOLON, ';')
Token(EOF, None)

AST Gerada:

DeclaracaoVariavel(tipo=int, nome=x, valor=Valor(valor=10))
DeclaracaoVariavel(tipo=int, nome=y, valor=Valor(valor=5))
DeclaracaoVariavel(tipo=int, nome=z, valor=Valor(valor=2))
Atribuicao(nome=x,expressao=ExpressaoBinaria(operador=+,
esquerda=Valor(valor=x),
direita=ExpressaoBinaria(operador=*,esquerda=Valor(valor=y),direita=Valor(valor=z))))

---

6. Conclusão

O código desenvolvido cumpre seu objetivo de implementar as fases de análise léxica e sintática de um compilador. Ele gera tokens de maneira

eficiente, ignora espaços e comentários, e constrói uma AST clara e organizada.

---

7. Código Completo

O código está funcional e serve como base para etapas futuras do compilador, como análise semântica e geração de código.