



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE CIÊNCIAS, TECNOLOGIAS E SAÚDE DO CAMPUS ARARANGUÁ
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Helder Henrique da Silva

Compiladores: Compilador Pacial da Linguagem C

Araranguá
2024

Helder Henrique da Silva

Compiladores: Compilador Pacial da Linguagem C

Trabalho Final de Compiladores do Curso de Graduação em Engenharia de Computação do Centro de Ciências, Tecnologias e Saúde do Campus Araranguá da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Alison Roberto Panisson

Araranguá

2024

RESUMO

Este projeto teve como objetivo a implementação parcial de um compilador para a linguagem C, englobando as fases de análise léxica, análise sintática e análise semântica. A análise léxica foi desenvolvida para identificar e classificar tokens no código fonte, utilizando expressões regulares para reconhecer elementos da linguagem. A análise sintática validou a estrutura gramatical do código e construiu a árvore sintática abstrata (AST), aplicando a notação BNF (Backus-Naur Form) para definir a gramática. A análise semântica verificou a consistência do código, garantindo a correta declaração e uso de variáveis, além da compatibilidade de tipos em operações e atribuições, através da implementação de uma tabela de símbolos. Durante o desenvolvimento, foram superados diversos desafios, como o gerenciamento de escopos, a verificação de compatibilidade de tipos e a declaração de structs. Os testes realizados mostraram que o compilador é capaz de reconhecer corretamente os tokens, analisar a estrutura gramatical e verificar a consistência semântica do código fonte. Sugestões para trabalhos futuros incluem a finalização da análise semântica para todos os casos de teste, a implementação de otimizações de código e a adição de suporte para mais estruturas da linguagem C.

Palavras-chave: Compilador. Análise Léxica. Análise Sintática. Análise Semântica. Linguagem C.

SUMÁRIO

1	INTRODUÇÃO	5
1.1	OBJETIVO DO PROJETO	5
1.2	IMPORTÂNCIA DOS COMPILADORES	5
1.3	VISÃO GERAL DAS FASES DO COMPILADOR	6
2	FUNDAMENTOS TEÓRICOS	7
2.1	ANÁLISE LÉXICA	7
2.1.1	Importância da Análise Léxica	7
2.2	ANÁLISE SINTÁTICA	7
2.2.1	Construção da Árvore Sintática Abstrata (AST)	7
2.3	ANÁLISE SEMÂNTICA	7
2.3.1	Verificação de Consistência do Código	7
2.4	LALR (LOOKAHEAD LR)	8
2.4.1	Utilização do LALR em Compiladores	8
2.5	S-ATRIBUTOS E ATRIBUTOS SINTETIZADOS	8
2.5.1	Definição de S-atributos	8
2.5.2	Aplicação dos Atributos Sintetizados na Análise Semântica	8
3	CONFIGURAÇÃO DO AMBIENTE	9
3.1	FERRAMENTAS E TECNOLOGIAS UTILIZADAS	9
3.2	CLONAGEM E CONFIGURAÇÃO DO REPOSITÓRIO	10
3.2.1	Clonar o Repositório	10
3.2.2	Configuração do Ambiente Virtual	10
3.3	COMO EXECUTAR O CÓDIGO	10
3.3.1	Executando o Analisador Léxico	10
3.3.2	Executando o Analisador Sintático	10
3.3.3	Executando o Analisador Semântico	11
3.4	TESTES	11
4	IMPLEMENTAÇÃO DO ANALISADOR LÉXICO	12
4.1	ESTRUTURA DO PROJETO	12
4.2	DEFINIÇÃO DE TOKENS	12
4.3	REGRAS LÉXICAS	12
4.4	TESTES DO ANALISADOR LÉXICO	14
5	IMPLEMENTAÇÃO DO ANALISADOR SINTÁTICO	15
5.1	ESTRUTURA DO PROJETO	15
5.2	DEFINIÇÃO DE GRAMÁTICAS	15
5.3	REGRAS SINTÁTICAS	15
5.4	PRECEDÊNCIA DOS OPERADORES	16
5.5	TESTES DO ANALISADOR SINTÁTICO	18

6	IMPLEMENTAÇÃO DO ANALISADOR SEMÂNTICO	19
6.1	ESTRUTURA DO PROJETO	19
6.2	TABELA DE SÍMBOLOS	19
6.3	REGRAS SEMÂNTICAS	19
6.4	TESTES DO ANALISADOR SEMÂNTICO	21
7	RESULTADOS E DISCUSSÃO	22
7.1	ANÁLISE DOS RESULTADOS	22
7.2	TESTE DE VERIFICAÇÃO DE TIPOS	51
7.3	DESAFIOS ENCONTRADOS	52
7.4	MELHORIAS E TRABALHOS FUTUROS	53
8	CONCLUSÃO	54
8.1	RESUMO DO PROJETO	54
8.2	CONCLUSÕES FINAIS	54
	APÊNDICE A – IMAGENS DOS CÓDIGOS DE TESTE	55

1 INTRODUÇÃO

1.1 OBJETIVO DO PROJETO

Este projeto tem como objetivo a implementação parcial de um compilador para a linguagem C, focando nas fases de análise léxica, análise sintática e análise semântica. A análise léxica já foi previamente implementada, enquanto as fases de análise sintática e análise semântica foram desenvolvidas para validar comandos de atribuição, operações aritméticas simples, estruturas condicionais (simples e aninhadas) e laços de repetição. Além disso, a análise semântica inclui a implementação de três ou mais ações semânticas, tais como verificação de declaração de variáveis antes do uso e verificação de tipagem.

A implementação busca fornecer um entendimento prático das técnicas e conceitos fundamentais envolvidos na construção de compiladores, oferecendo uma base sólida para o desenvolvimento de compiladores completos e robustos. O projeto também serve como uma ferramenta educativa para ilustrar os princípios de análise léxica, sintática e semântica em um contexto real.

1.2 IMPORTÂNCIA DOS COMPILADORES

Compiladores são componentes essenciais na computação moderna, permitindo a tradução de código fonte escrito em linguagens de alto nível para código de máquina executável. Esta tradução é crucial para a execução eficiente de programas em diversos dispositivos e sistemas operacionais.

A importância dos compiladores pode ser destacada em várias áreas:

- **Desenvolvimento de Software:** Compiladores permitem que os desenvolvedores escrevam programas em linguagens de alto nível, que são mais intuitivas e fáceis de usar. O compilador então traduz esses programas para código de máquina, que pode ser executado pelo hardware.
- **Otimização de Código:** Compiladores modernos não apenas traduzem código, mas também otimizam o código gerado para melhorar a eficiência e desempenho do programa. Isso inclui otimizações que reduzem o uso de memória, diminuem o tempo de execução e melhoram a utilização de recursos do sistema.
- **Portabilidade:** Ao traduzir código fonte para diferentes plataformas, os compiladores permitem que o mesmo programa seja executado em diversos tipos de hardware e sistemas operacionais, aumentando a portabilidade do software.
- **Segurança e Confiabilidade:** Compiladores realizam várias verificações durante a tradução do código, identificando erros de sintaxe e semântica que podem causar

falhas na execução do programa. Isso contribui para a criação de software mais seguro e confiável.

1.3 VISÃO GERAL DAS FASES DO COMPILADOR

Um compilador é composto por várias fases, cada uma desempenhando um papel específico no processo de tradução do código fonte para código de máquina. As principais fases incluem:

1. **Análise Léxica:** Esta é a primeira fase do compilador. O analisador léxico, ou lexer, lê o código fonte e o divide em unidades menores chamadas tokens. Cada token representa uma unidade léxica, como palavras-chave, identificadores, operadores e símbolos de pontuação. O objetivo principal desta fase é simplificar o código fonte, removendo espaços em branco e comentários, e classificar os diferentes elementos do código.
2. **Análise Sintática:** Também conhecida como parsing, esta fase verifica se a sequência de tokens gerada pela análise léxica segue as regras gramaticais da linguagem de programação. O analisador sintático, ou parser, constrói uma árvore sintática abstrata (AST) que representa a estrutura hierárquica do programa. Esta árvore é utilizada nas fases subsequentes do compilador para realizar verificações mais detalhadas e otimizações.
3. **Análise Semântica:** Esta fase verifica a consistência semântica do programa. Isso inclui garantir que as variáveis sejam declaradas antes de serem usadas, verificar a compatibilidade de tipos em operações e atribuições, e gerenciar os escopos das variáveis. A análise semântica utiliza a árvore sintática abstrata gerada na fase anterior para realizar essas verificações.
4. **Otimização de Código:** Opcionalmente, os compiladores podem realizar várias otimizações para melhorar o desempenho do código gerado. Essas otimizações podem incluir a eliminação de código morto, a unificação de expressões comuns e a reorganização do código para melhorar a eficiência de cache.
5. **Geração de Código:** Nesta fase, o compilador traduz a árvore sintática abstrata otimizada em código de máquina ou código intermediário que pode ser executado pela máquina alvo. A geração de código envolve a conversão das estruturas de alto nível para instruções de baixo nível específicas do processador.
6. **Ligação e Carregamento:** Finalmente, o código de máquina gerado é ligado com outras bibliotecas e módulos, resultando em um executável final que pode ser carregado e executado pelo sistema operacional.

2 FUNDAMENTOS TEÓRICOS

2.1 ANÁLISE LÉXICA

A análise léxica é a primeira fase de um compilador, onde o código fonte é lido e dividido em unidades menores chamadas tokens. O analisador léxico, ou lexer, é responsável por identificar padrões no código fonte e classificá-los como tokens, que representam palavras-chave, identificadores, operadores, literais e símbolos de pontuação.

2.1.1 Importância da Análise Léxica

A análise léxica simplifica o código fonte ao remover espaços em branco e comentários, e ao classificar os elementos do código em categorias significativas. Isso facilita a análise posterior, como a análise sintática e semântica, permitindo que o compilador se concentre na estrutura e no significado do código, em vez de nos detalhes de sua representação textual.

2.2 ANÁLISE SINTÁTICA

A análise sintática, ou parsing, é a segunda fase de um compilador. Esta fase verifica se a sequência de tokens gerada pela análise léxica segue as regras gramaticais da linguagem de programação. O analisador sintático, ou parser, constrói uma árvore sintática abstrata (AST) que representa a estrutura hierárquica do programa.

2.2.1 Construção da Árvore Sintática Abstrata (AST)

A AST é uma representação em árvore do código fonte, onde cada nó da árvore corresponde a uma construção sintática da linguagem. A AST abstrai os detalhes específicos da sintaxe, focando na estrutura lógica e na hierarquia das operações. Esta árvore é utilizada nas fases subsequentes do compilador para realizar verificações mais detalhadas e otimizações.

2.3 ANÁLISE SEMÂNTICA

A análise semântica é a terceira fase do compilador, responsável por verificar a consistência semântica do código fonte. Esta fase garante que as variáveis sejam declaradas antes de serem usadas, verifica a compatibilidade de tipos em operações e atribuições, e gerencia os escopos das variáveis.

2.3.1 Verificação de Consistência do Código

A análise semântica utiliza a árvore sintática abstrata gerada na fase de análise sintática para realizar verificações mais detalhadas. Isso inclui a verificação de tipos, onde

se assegura que operações aritméticas e lógicas são realizadas entre tipos compatíveis, e a verificação de escopos, onde se garante que variáveis e funções são utilizadas dentro dos contextos apropriados.

2.4 LALR (LOOKAHEAD LR)

LALR (Lookahead LR) é um método eficiente de análise sintática utilizado por muitos compiladores, incluindo aqueles construídos com a biblioteca PLY. O método LALR combina a capacidade de análise eficiente do LR com a compactação de estados, o que reduz a memória necessária para a tabela de análise.

2.4.1 Utilização do LALR em Compiladores

O método LALR é especialmente útil para linguagens de programação complexas, como C, onde a gramática pode ser ambígua e rica em detalhes. Ao utilizar o LALR, é possível construir analisadores sintáticos que são eficientes tanto em termos de tempo quanto de espaço, mantendo a precisão na análise da estrutura do código fonte.

2.5 S-ATRIBUTOS E ATRIBUTOS SINTETIZADOS

Em análise semântica, os atributos podem ser sintetizados ou herdados. Os atributos sintetizados, ou S-atributos, são aqueles que dependem exclusivamente dos valores dos filhos do nó na árvore sintática. Eles são computados de baixo para cima na árvore.

2.5.1 Definição de S-atributos

Os S-atributos são usados para passar informações de nós filhos para seus pais na árvore sintática. Eles são fundamentais para a construção de compiladores S-atribuídos, onde todas as ações semânticas associadas às produções de uma gramática atribuem valores a atributos sintetizados.

2.5.2 Aplicação dos Atributos Sintetizados na Análise Semântica

Os atributos sintetizados são utilizados para calcular e propagar informações durante a análise semântica. Por exemplo, ao analisar uma expressão aritmética, os tipos dos operandos podem ser propagados para o nó pai, permitindo a verificação de compatibilidade de tipos. Esses atributos também são essenciais para a realização de verificações de escopo e para a geração de código intermediário ou final.

3 CONFIGURAÇÃO DO AMBIENTE

3.1 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

Para a implementação do compilador parcial da linguagem C, foram utilizadas diversas ferramentas e tecnologias que facilitam o desenvolvimento e a análise do código. Abaixo, são listadas e descritas as principais ferramentas e bibliotecas utilizadas no projeto:

Ferramenta/Biblioteca	Descrição
Python	A linguagem de programação principal utilizada para o desenvolvimento do compilador.
PLY (Python Lex-Yacc)	Biblioteca utilizada para a implementação dos analisadores léxico e sintático.
astroid	Biblioteca utilizada para a geração de árvores sintáticas abstratas em Python.
Cerberus	Biblioteca de validação de dados em Python.
cfgv	Biblioteca de validação e conversão de arquivos de configuração.
dill	Biblioteca para serialização de objetos Python.
distlib	Biblioteca de distribuição de pacotes em Python.
filelock	Biblioteca para criação de locks em arquivos.
identify	Biblioteca para identificar arquivos e seus tipos.
iniconfig	Biblioteca para leitura e escrita de arquivos de configuração INI.
isort	Ferramenta para ordenar automaticamente as importações de módulos em Python.
mccabe	Biblioteca para análise de complexidade de código em Python.
nodeenv	Biblioteca para criar ambientes virtuais para Node.js.
packaging	Biblioteca para manipulação de pacotes e versões em Python.
platformdirs	Biblioteca para determinar diretórios específicos da plataforma.
pluggy	Biblioteca para criação de sistemas de plugins.
pre-commit	Ferramenta para gerenciamento de hooks de pré-commit.
pylint	Ferramenta para análise estática de código em Python.
pytest	Framework de testes em Python.
PyYAML	Biblioteca para leitura e escrita de arquivos YAML.
setuptools	Biblioteca para gerenciamento de pacotes Python.
tomlkit	Biblioteca para manipulação de arquivos TOML.
virtualenv	Ferramenta para criação de ambientes virtuais Python.

Tabela 1 – Ferramentas e Tecnologias Utilizadas no Projeto

3.2 CLONAGEM E CONFIGURAÇÃO DO REPOSITÓRIO

Para iniciar o desenvolvimento com este projeto, é necessário clonar o repositório e configurar o ambiente de desenvolvimento. As instruções a seguir detalham como realizar esses passos:

3.2.1 Clonar o Repositório

Primeiramente, clone o repositório do GitHub utilizando o seguinte comando:

```
git clone https://github.com/theHprogrammer-UFSCWORKS/compiladores-c-parcial-compiler
cd compiladores-c-parcial-compiler
```

3.2.2 Configuração do Ambiente Virtual

Para garantir que todas as dependências do projeto sejam instaladas corretamente e evitar conflitos com outras bibliotecas, é recomendável utilizar um ambiente virtual. Abaixo estão os comandos para configurar o ambiente virtual:

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

O primeiro comando cria um ambiente virtual chamado `.venv`. O segundo comando ativa o ambiente virtual, e o terceiro comando instala todas as dependências listadas no arquivo `requirements.txt`.

3.3 COMO EXECUTAR O CÓDIGO

Para executar o código do compilador, siga as instruções abaixo para cada tipo de análise.

3.3.1 Executando o Analisador Léxico

Para executar o analisador léxico em um arquivo C, utilize o seguinte comando no terminal:

```
python main.py <caminho/para/arquivo.c> lexical
```

3.3.2 Executando o Analisador Sintático

Para executar o analisador sintático em um arquivo C, utilize o seguinte comando no terminal:

```
python main.py <caminho/para/arquivo.c> syntactic
```

3.3.3 Executando o Analisador Semântico

Para executar o analisador semântico em um arquivo C, utilize o seguinte comando no terminal:

```
python main.py <caminho/para/arquivo.c> semantic
```

Nota: O arquivo `tests/test_case5.c` está bloqueado para o modo semântico devido à complexidade parcial das declarações de estruturas, o que pode acarretar erros.

3.4 TESTES

Para testar os analisadores, os arquivos de teste devem ser colocados dentro do diretório `tests`. Utilize o caminho do teste dentro desse diretório para executar os códigos no terminal.

Abaixo está uma tabela com a lista do que será testado em cada arquivo de teste:

Arquivo de Teste	Aspectos Testados
test_case0.c	Declaração de variáveis, Função <code>main</code> , Retorno de função
test_case1.c	Declaração de variáveis, Estruturas condicionais, Função <code>main</code>
test_case2.c	Declaração de variáveis, Comentários, Estrutura de repetição, Função <code>main</code>
test_case3.c	Declaração de variáveis, Estrutura de repetição, Função <code>main</code>
test_case4.c	Declaração de funções, Chamadas de funções, Estruturas condicionais, Função <code>main</code>
test_case5.c	Declaração de variáveis, Estruturas condicionais, Estruturas de repetição, Declaração de estruturas, Função <code>main</code>
test_case6.c	Declaração de variáveis, Estruturas condicionais, Função <code>main</code>

Tabela 2 – Aspectos testados em cada arquivo de teste.

As imagens dos códigos de teste podem ser encontradas no Apêndice A.

4 IMPLEMENTAÇÃO DO ANALISADOR LÉXICO

4.1 ESTRUTURA DO PROJETO

A estrutura do projeto relacionada ao analisador léxico está organizada no diretório `src/analizador_lexico`. Este diretório contém os arquivos responsáveis pela definição dos tokens e regras léxicas, além do arquivo principal do analisador léxico.

A estrutura dos arquivos e diretórios é a seguinte:

```
src /
|-- analizador_lexico /
|   |-- __init__.py
|   |-- lexer.py
|   |-- token_rules.py
|   |-- README.md
```

- **lexer.py**: Arquivo principal que configura o analisador léxico utilizando a biblioteca PLY.
- **token_rules.py**: Contém a definição dos tokens e as regras léxicas.
- **README.md**: Documentação específica do analisador léxico.

4.2 DEFINIÇÃO DE TOKENS

Os tokens são definidos no arquivo `token_rules.py`. Cada token representa uma unidade léxica, como palavras-chave, identificadores, operadores e símbolos de pontuação. A definição dos tokens é feita utilizando expressões regulares.

Abaixo está uma imagem ilustrando a definição dos tokens, incluindo palavras reservadas e tokens extras:

4.3 REGRAS LÉXICAS

As regras léxicas são responsáveis por identificar e classificar as diferentes unidades léxicas presentes no código fonte. Essas regras são implementadas utilizando expressões regulares e funções que retornam os tokens correspondentes.

No arquivo `lexer.py`, o analisador léxico é configurado utilizando a biblioteca PLY. As regras léxicas são definidas no `token_rules.py` e importadas no `lexer.py`:

```
from ply import lex
from .token_rules import TokenRules

self.lexer = lex.lex(module=self.token_rules)
```



```

1 class TokenRules:
2     # Inicializa as palavras reservadas e os tokens
3     def __init__(self):
4         self.reserved = {
5             'if' : 'IF', 'do' : 'DO', 'int' : 'INT',
6             'for' : 'FOR', 'else' : 'ELSE',
7             'char' : 'CHAR', 'void' : 'VOID',
8             'auto' : 'AUTO', 'case' : 'CASE',
9             'goto' : 'GOTO', 'enum' : 'ENUM',
10            'long' : 'LONG', 'float' : 'FLOAT',
11            'const' : 'CONST', 'while' : 'WHILE',
12            'break' : 'BREAK', 'short' : 'SHORT',
13            'union' : 'UNION', 'return' : 'RETURN',
14            'double' : 'DOUBLE', 'extern' : 'EXTERN',
15            'signed' : 'SIGNED', 'sizeof' : 'SIZEOF',
16            'static' : 'STATIC', 'struct' : 'STRUCT',
17            'switch' : 'SWITCH', 'define' : 'DEFINE',
18            'default' : 'DEFAULT', 'typedef' : 'TYPEDEF',
19            'include' : 'INCLUDE', 'continue' : 'CONTINUE',
20            'register' : 'REGISTER', 'unsigned' : 'UNSIGNED',
21            'volatile' : 'VOLATILE',
22        }
23
24        self.tokens = [
25            'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'POWER', 'MOD', 'INCREMENT', 'DECREMENT',
26            'LT', 'LE', 'GT', 'GE', 'NE', 'COMPARATOR',
27            'EQUALS', 'MOD_ASSIGN', 'PLUS_ASSIGN', 'MINUS_ASSIGN', 'TIMES_ASSIGN', 'DIVIDE_ASSIGN',
28            'AND_ASSIGN', 'OR_ASSIGN', 'XOR_ASSIGN', 'LSHIFT_ASSIGN', 'RSHIFT_ASSIGN',
29            'AND', 'OR', 'NOT',
30            'BITWISE_AND', 'BITWISE_OR', 'BITWISE_XOR', 'LSHIFT', 'RSHIFT', 'BITWISE_NOT',
31            'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE', 'LBRACKET', 'RBRACKET', 'COMMA', 'SEMICOLON', 'COLON', 'NEWLINE',
32            'INTEGER', 'FLOAT_N', 'STRING', 'ID', 'MAIN', 'LIBRARY',
33            'TERNARY', 'ELLIPSIS', 'ARROW', 'DOT', 'HASH', 'DOUBLEHASH'
34        ] + list(self.reserved.values())
35

```

Figura 1 – Definição de tokens no analisador léxico.

Abaixo está uma imagem mostrando algumas regras simples e complexas definidas no arquivo `token_rules.py`:



```

1
2 # ...
3 # Pontuação e Delimitadores
4 t_LPAREN = r'\('
5 t_RPAREN = r'\)'
6 t_LBRACE = r'\{'
7 t_RBRACE = r'\}'
8 t_LBRACKET = r'\['
9 t_RBRACKET = r'\]'
10 t_COMMA = r','
11 t_SEMICOLON = r';'
12 t_COLON = r':'
13
14 # Literais Numéricos e Strings
15 t_INTEGER = r'\d+'
16
17 # ...

```

Figura 2 – Regras léxicas simples.

No arquivo `lexer.py`, o analisador léxico é configurado utilizando a biblioteca PLY. As regras léxicas são definidas no `token_rules.py` e importadas no `lexer.py`.



```
1
2 # Ignora espaços em branco e tabulações
3 t_ignore = ' \t'
4
5 # Define novas linhas
6 def t_NEWLINE(self, t):
7     r'\n+'
8     t.lexer.lineno += len(t.value)
9
10 # Define o token de função principal
11 def t_MAIN(self, t):
12     r'main'
13     return t
14
15 # Define o identificador
16 def t_ID(self, t):
17     r'[a-zA-Z_][a-zA-Z0-9_]*'
18     t.type = self.reserved.get(t.value, 'ID')
19     return t
20
21 # Define o token de número flutuante
22 def t_FLOAT_N(self, t):
23     r'((\d*\.\d+)(E[\+-]?[0-9]\d*)|([1-9]\d+E[\+-]?[0-9]\d*))'
24     t.value = float(t.value)
25     return t
```

Figura 3 – Regras léxicas complexas.

4.4 TESTES DO ANALISADOR LÉXICO

Os testes do analisador léxico são realizados para garantir que os tokens sejam reconhecidos corretamente e que o analisador esteja funcionando conforme o esperado. Os arquivos de teste são colocados no diretório `tests`, e os testes podem ser executados utilizando o caminho dos arquivos de teste.

Para realizar os testes, utilize o comando:

```
python main.py tests/test_case0.c lexical
```

5 IMPLEMENTAÇÃO DO ANALISADOR SINTÁTICO

5.1 ESTRUTURA DO PROJETO

A estrutura do projeto relacionada ao analisador sintático está organizada no diretório `src/analizador_sintatico`. Este diretório contém os arquivos responsáveis pela definição das gramáticas e regras sintáticas, além do arquivo principal do analisador sintático.

A estrutura dos arquivos e diretórios é a seguinte:

```
src/
|-- analisador_sintatico/
|   |-- __init__.py
|   |-- parser.py
|   |-- grammar_rules.py
|   |-- README.md
```

- **parser.py**: Arquivo principal que configura o analisador sintático utilizando a biblioteca PLY.
- **grammar_rules.py**: Contém a definição das gramáticas e as regras sintáticas.
- **README.md**: Documentação específica do analisador sintático.

5.2 DEFINIÇÃO DE GRAMÁTICAS

As gramáticas são definidas no arquivo `grammar_rules.py`. Cada regra gramatical especifica uma produção na linguagem C. A definição das gramáticas é feita utilizando a notação BNF (Backus-Naur Form).

Abaixo está uma imagem ilustrando a definição de algumas gramáticas simples e complexas:

5.3 REGRAS SINTÁTICAS

As regras sintáticas são responsáveis por analisar a estrutura do código fonte e garantir que ele siga as regras gramaticais da linguagem. Essas regras são implementadas utilizando a notação BNF e funções associadas a cada produção.

No arquivo `parser.py`, o analisador sintático é configurado utilizando a biblioteca PLY. As regras sintáticas são definidas no `grammar_rules.py` e importadas no `parser.py`:

```
from ply import yacc
from .grammar_rules import GrammarRules
```

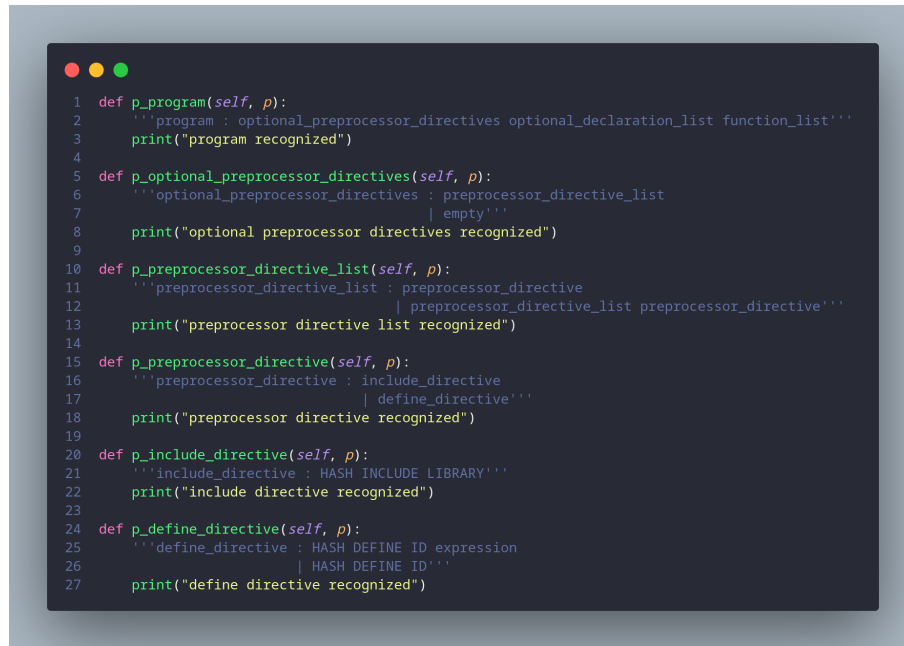



Figura 4 – Definição de gramáticas no analisador sintático.

```
self.parser = yacc.yacc(module=self.grammar)
```

5.4 PRECEDÊNCIA DOS OPERADORES

A precedência dos operadores é definida no arquivo `grammar_rules.py` para garantir que as expressões sejam avaliadas corretamente de acordo com as regras da linguagem C. A precedência é especificada utilizando tuplas que definem a ordem e a associatividade dos operadores.

Exemplo de definição de precedência dos operadores:

```
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('left', 'BITWISE_OR'),
    ('left', 'BITWISE_XOR'),
    ('left', 'BITWISE_AND'),
    ('left', 'COMPARATOR', 'NE'),
    ('left', 'LT', 'LE', 'GT', 'GE'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE', 'MOD'),
    ('left', 'LSHIFT', 'RSHIFT'),
    ('right', 'INCREMENT', 'DECREMENT'),
    ('right', 'DOT'),
)
```

Abaixo está uma tabela detalhando as principais regras gramaticais definidas no arquivo `grammar_rules.py`:

Regra Gramatical	Descrição
<code>program</code>	Reconhece um programa completo, incluindo diretivas do pré-processador, declarações e funções.
<code>optional_preprocessor_directives</code>	Reconhece diretivas do pré-processador opcionais ou nenhuma diretiva.
<code>preprocessor_directive_list</code>	Reconhece uma lista de diretivas do pré-processador.
<code>preprocessor_directive</code>	Reconhece uma única diretiva do pré-processador, como include ou define .
<code>include_directive</code>	Reconhece uma diretiva include .
<code>define_directive</code>	Reconhece uma diretiva define .
<code>optional_declaration_list</code>	Reconhece uma lista opcional de declarações ou nenhuma declaração.
<code>declaration_list</code>	Reconhece uma lista de declarações.
<code>function_list</code>	Reconhece uma lista de funções.
<code>function</code>	Reconhece uma definição de função, incluindo a função principal main .
<code>main_function</code>	Reconhece a definição da função principal main .
<code>parameter_list</code>	Reconhece uma lista de parâmetros de função.
<code>parameter</code>	Reconhece um parâmetro de função.
<code>compound_statement</code>	Reconhece um bloco de código delimitado por chaves <code>{}</code> .
<code>statement_list</code>	Reconhece uma lista de instruções.
<code>statement</code>	Reconhece uma única instrução, que pode ser de diversos tipos (declaração, atribuição, controle de fluxo, etc.).
<code>declaration</code>	Reconhece uma declaração de variável, typedef ou struct .
<code>typedef_declaration</code>	Reconhece uma declaração typedef .
<code>struct_declaration</code>	Reconhece uma declaração de estrutura.
<code>struct_members</code>	Reconhece uma lista de membros de uma estrutura.
<code>struct_member</code>	Reconhece um único membro de uma estrutura.
<code>optional_id</code>	Reconhece um identificador opcional ou nenhum identificador.
<code>init_declarator_list</code>	Reconhece uma lista de inicializadores de declaradores.
<code>init_declarator</code>	Reconhece um inicializador de declarador.
<code>initializer_list</code>	Reconhece uma lista de inicializadores.
<code>type_specifier</code>	Reconhece um especificador de tipo.
<code>non_empty_pre_type_specifier</code>	Reconhece um especificador de tipo não vazio.
<code>base_type</code>	Reconhece um tipo base (int , char , float , etc.).
<code>pre_type_specifier</code>	Reconhece um especificador de tipo prévio (const , volatile , etc.).
<code>assignment</code>	Reconhece uma instrução de atribuição.
<code>expression</code>	Reconhece uma expressão, incluindo operações aritméticas e lógicas.
<code>term</code>	Reconhece um termo em uma expressão aritmética.
<code>factor</code>	Reconhece um fator em uma expressão, como um número ou uma variável.
<code>if_statement</code>	Reconhece uma instrução if (condicional).
<code>while_statement</code>	Reconhece uma instrução while (laço de repetição).
<code>do_while_statement</code>	Reconhece uma instrução do-while (laço de repetição).
<code>for_statement</code>	Reconhece uma instrução for (laço de repetição).
<code>switch_statement</code>	Reconhece uma instrução switch (seleção múltipla).
<code>case_list</code>	Reconhece uma lista de casos em uma instrução switch .
<code>case</code>	Reconhece um caso em uma instrução switch .
<code>default_case</code>	Reconhece o caso padrão em uma instrução switch .
<code>break_statement</code>	Reconhece uma instrução break .
<code>continue_statement</code>	Reconhece uma instrução continue .
<code>return_statement</code>	Reconhece uma instrução return .
<code>function_call</code>	Reconhece uma chamada de função.
<code>argument_list</code>	Reconhece uma lista de argumentos em uma chamada de função.
<code>empty</code>	Reconhece uma produção vazia.
<code>p_error</code>	Função de tratamento de erros sintáticos.

Tabela 3 – Tabela de Regras Gramaticais

5.5 TESTES DO ANALISADOR SINTÁTICO

Os testes do analisador sintático são realizados para garantir que as produções sejam reconhecidas corretamente e que o analisador esteja funcionando conforme o esperado. Os arquivos de teste são colocados no diretório `tests`, e os testes podem ser executados utilizando o caminho dos arquivos de teste.

Para realizar os testes, utilize o comando:

```
python main.py tests/test_case0.c syntactic
```

6 IMPLEMENTAÇÃO DO ANALISADOR SEMÂNTICO

6.1 ESTRUTURA DO PROJETO

A estrutura do projeto relacionada ao analisador semântico está organizada no diretório `src/analizador_semantico`. Este diretório contém os arquivos responsáveis pela definição da tabela de símbolos e das regras semânticas, além do arquivo principal do analisador semântico.

A estrutura dos arquivos e diretórios é a seguinte:

```
src /
|-- analizador_semantico /
|   |-- __init__.py
|   |-- semantic_rules.py
|   |-- semantic_parser.py
|   |-- semantic_actions.py
|   |-- README.md
```

- **semantic_rules.py**: Define a tabela de símbolos e suas operações.
- **semantic_parser.py**: Configura o parser para a análise semântica.
- **semantic_actions.py**: Define as regras gramaticais e as ações semânticas.
- **README.md**: Documentação específica do analisador semântico.

6.2 TABELA DE SÍMBOLOS

A tabela de símbolos é implementada no arquivo `semantic_rules.py`. Ela é essencial para a análise semântica, pois gerencia as variáveis, funções e seus escopos dentro do programa. A tabela de símbolos garante que as variáveis sejam declaradas antes de serem usadas, verifica a compatibilidade de tipos em operações e atribuições, e gerencia os escopos das variáveis. Os principais componentes implementados podem ser verificados na tabela 4.

6.3 REGRAS SEMÂNTICAS

As regras semânticas são implementadas no arquivo `semantic_actions.py`. Elas são responsáveis por verificar a consistência semântica do código fonte. Isso inclui a verificação de declarações de variáveis, compatibilidade de tipos em atribuições e operações, e verificação de declaração de funções. As principais ações semânticas implementadas podem ser observadas na tabela 5.

Componente	Descrição
<code>global_scope</code>	Gerencia as variáveis no escopo global.
<code>local_scope_stack</code>	Pilha que gerencia os escopos locais.
<code>standard_functions</code>	Conjunto de funções padrão reconhecidas pelo compilador (por exemplo, <code>printf</code> , <code>scanf</code>).
<code>enter_scope</code>	Entra em um novo escopo local.
<code>exit_scope</code>	Sai do escopo local atual.
<code>add</code>	Adiciona uma nova variável à tabela de símbolos no escopo atual.
<code>lookup</code>	Procura uma variável na tabela de símbolos, começando pelo escopo mais interno até o global.
<code>__str__</code>	Retorna uma representação em string do escopo global da tabela de símbolos.

Tabela 4 – Principais componentes da tabela de símbolos.

- Declarações de variáveis
- Compatibilidade de tipos em atribuições
- Compatibilidade de tipos em operações aritméticas e lógicas
- Verificação de declaração de funções

Regra Semântica	Descrição
<code>p_function</code>	Adiciona a função à tabela de símbolos e entra em um novo escopo.
<code>p_main_function</code>	Adiciona a função <code>main</code> à tabela de símbolos e entra em um novo escopo.
<code>p_parameter</code>	Adiciona o parâmetro da função à tabela de símbolos.
<code>p_compound_statement</code>	Entra em um novo escopo ao reconhecer um bloco de código e sai do escopo ao final.
<code>p_declaration</code>	Adiciona a variável declarada à tabela de símbolos.
<code>p_struct_declaration</code>	Adiciona a estrutura e seus membros à tabela de símbolos.
<code>p_assignment</code>	Verifica a compatibilidade de tipos na instrução de atribuição.
<code>p_expression</code>	Verifica a compatibilidade de tipos em operações aritméticas e lógicas.
<code>p_term</code>	Verifica a compatibilidade de tipos em termos de uma expressão aritmética.
<code>p_function_call</code>	Verifica se a função chamada foi declarada.

Tabela 5 – Tabela de Regras Semânticas

Explicação Detalhada das Implementações

- ****p_function****: - Adiciona a função à tabela de símbolos. - Entra em um novo escopo ao iniciar a definição da função.

- ****p_main_function****: - Adiciona a função `main` à tabela de símbolos. - Entra em um novo escopo ao iniciar a definição da função `main`.
- ****p_parameter****: - Adiciona o parâmetro da função à tabela de símbolos para garantir que ele seja reconhecido dentro do escopo da função.
- ****p_compound_statement****: - Entra em um novo escopo ao reconhecer um bloco de código delimitado por chaves `{}`. - Sai do escopo ao final do bloco para manter a consistência dos escopos.
- ****p_declaration****: - Adiciona a variável declarada à tabela de símbolos, verificando se a variável já foi declarada no escopo atual para evitar redefinições.
- ****p_struct_declaration****: - Adiciona a estrutura e seus membros à tabela de símbolos, permitindo o uso de tipos definidos pelo usuário.
- ****p_assignment****: - Verifica a compatibilidade de tipos na instrução de atribuição para garantir que o tipo do valor atribuído seja compatível com o tipo da variável.
- ****p_expression****: - Verifica a compatibilidade de tipos em operações aritméticas e lógicas para garantir que as operações sejam realizadas entre tipos compatíveis.
- ****p_term****: - Verifica a compatibilidade de tipos em termos de uma expressão aritmética, garantindo que operações como multiplicação e divisão sejam realizadas entre tipos compatíveis.
- ****p_function_call****: - Verifica se a função chamada foi declarada para garantir que apenas funções conhecidas sejam utilizadas.

6.4 TESTES DO ANALISADOR SEMÂNTICO

Os testes do analisador semântico são realizados para garantir que as regras semânticas sejam aplicadas corretamente e que o analisador esteja funcionando conforme o esperado. Os arquivos de teste são colocados no diretório `tests`, e os testes podem ser executados utilizando o caminho dos arquivos de teste.

Para realizar os testes, utilize o comando:

```
python main.py tests/test_case0.c semantic
```

7 RESULTADOS E DISCUSSÃO

7.1 ANÁLISE DOS RESULTADOS

A implementação do compilador parcial para a linguagem C, incluindo os analisadores léxico, sintático e semântico, foi testada utilizando os testes abordados anteriormente. A visualização dos códigos de teste pode ser encontrada no apêndice A. Nesta seção, serão apresentados e analisados os resultados obtidos para cada um desses testes.

Os resultados foram obtidos executando o comando para o modo semântico, que inclui os resultados das análises léxica e sintática. Para o caso de teste `test_case5.c`, que possui a parte semântica incompleta para declaração de structs, foram executados apenas os comandos para as análises léxica e sintática.

Teste `test_case0.c`:

```
Symbol Table initialized.
Tokens:
LexToken(INT, 'int', 1, 0)
LexToken(MAIN, 'main', 1, 4)
LexToken(LPAREN, '(', 1, 8)
LexToken(RPAREN, ')', 1, 9)
LexToken(LBRACE, '{', 1, 11)
LexToken(INT, 'int', 2, 17)
LexToken(ID, 'x', 2, 21)
LexToken(EQUALS, '=', 2, 23)
LexToken(INTEGER, '10', 2, 25)
LexToken(SEMICOLON, ';', 2, 27)
LexToken(FLOAT, 'float', 3, 33)
LexToken(ID, 'y', 3, 39)
LexToken(EQUALS, '=', 3, 41)
LexToken(FLOAT_N, '20.5', 3, 43)
LexToken(SEMICOLON, ';', 3, 47)
LexToken(CHAR, 'char', 4, 53)
LexToken(ID, 'z', 4, 58)
LexToken(EQUALS, '=', 4, 60)
LexToken(STRING, '"A"', 4, 62)
LexToken(SEMICOLON, ';', 4, 65)
LexToken(RETURN, 'return', 6, 72)
LexToken(INTEGER, '0', 6, 79)
LexToken(SEMICOLON, ';', 6, 80)
LexToken(RBRACE, '}', 7, 82)
```

Parsing and Semantic Analysis:

optional preprocessor directives recognized

empty declaration list recognized

optional declaration list recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

factor recognized

term recognized

expression recognized

init declarator recognized

init declarator list recognized

declaration recognized

statement recognized

statement list recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

factor recognized

term recognized

expression recognized

init declarator recognized

init declarator list recognized

declaration recognized

statement recognized

statement list recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

factor recognized

term recognized


```
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
return statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized
{'x': {'type': ' int', 'lineno': 0},
'y': {'type': ' float', 'lineno': 0},
'z': {'type': ' char', 'lineno': 0},
'main': {'type': ' int', 'lineno': 1}}
```

Teste test_case1.c:

```
Symbol Table initialized.
Tokens:
LexToken(INT,'int',1,0)
LexToken(MAIN,'main',1,4)
LexToken(LPAREN,'(',1,8)
LexToken(RPAREN,')',1,9)
LexToken(LBRACE,'{',1,11)
LexToken(INT,'int',3,18)
LexToken(ID,'age',3,22)
LexToken(EQUALS,'=',3,26)
LexToken(INTEGER,'6',3,28)
LexToken(SEMICOLON,';',3,29)
LexToken(INT,'int',4,35)
LexToken(ID,'level',4,39)
LexToken(EQUALS,'=',4,45)
LexToken(INTEGER,'0',4,47)
```

```
LexToken(SEMICOLON,',';','4,48)
LexToken(IF,'if',6,55)
LexToken(LPAREN,'(',6,58)
LexToken(ID,'age',6,59)
LexToken(LT,'<',6,63)
LexToken(INTEGER,'18',6,65)
LexToken(RPAREN,')',6,67)
LexToken(LBRACE,'{',6,69)
LexToken(ID,'level',7,73)
LexToken(EQUALS,'=',7,79)
LexToken(INTEGER,'1',7,81)
LexToken(SEMICOLON,',';','7,82)
LexToken(RBRACE,'}',8,88)
LexToken(ELSE,'else',8,90)
LexToken(LBRACE,'{',8,95)
LexToken(IF,'if',9,99)
LexToken(LPAREN,'(',9,102)
LexToken(ID,'age',9,103)
LexToken(LE,'<=',9,107)
LexToken(INTEGER,'60',9,110)
LexToken(RPAREN,')',9,112)
LexToken(LBRACE,'{',9,114)
LexToken(ID,'level',10,119)
LexToken(EQUALS,'=',10,125)
LexToken(INTEGER,'2',10,127)
LexToken(SEMICOLON,',';','10,128)
LexToken(RBRACE,'}',11,135)
LexToken(ELSE,'else',11,137)
LexToken(LBRACE,'{',11,142)
LexToken(ID,'level',12,147)
LexToken(EQUALS,'=',12,153)
LexToken(INTEGER,'3',12,155)
LexToken(SEMICOLON,',';','12,156)
LexToken(RBRACE,'}',13,163)
LexToken(RBRACE,'}',14,169)
LexToken(RBRACE,'}',15,171)
```

Parsing and Semantic Analysis:

optional preprocessor directives recognized

empty declaration list recognized
optional declaration list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
factor recognized
term recognized

expression recognized
assignment recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
if statement recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
if statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized

```
function recognized
function list recognized
program recognized
{'age': {'type': ' int', 'lineno': 0},
'level': {'type': ' int', 'lineno': 0},
'main': {'type': ' int', 'lineno': 1}}
```

Teste test_case2.c:

Symbol Table initialized.

Tokens:

```
LexToken(INT, 'int', 2, 1)
LexToken(MAIN, 'main', 2, 5)
LexToken(LPAREN, '(', 2, 9)
LexToken(RPAREN, ')', 2, 10)
LexToken(LBRACE, '{', 2, 11)
LexToken(INT, 'int', 4, 26)
LexToken(ID, 'x', 4, 30)
LexToken(EQUALS, '=', 4, 32)
LexToken(INTEGER, '2', 4, 34)
LexToken(SEMICOLON, ';', 4, 35)
LexToken(FLOAT, 'float', 11, 63)
LexToken(ID, 'babu', 11, 69)
LexToken(EQUALS, '=', 11, 74)
LexToken(FLOAT_N, '2.3', 11, 76)
LexToken(SEMICOLON, ';', 11, 79)
LexToken(WHILE, 'while', 12, 81)
LexToken(LPAREN, '(', 12, 86)
LexToken(ID, 'x', 12, 87)
LexToken(GT, '>', 12, 88)
LexToken(INTEGER, '3', 12, 89)
LexToken(RPAREN, ')', 12, 90)
LexToken(LBRACE, '{', 12, 91)
LexToken(ID, 'printf', 13, 97)
LexToken(LPAREN, '(', 13, 103)
LexToken(RPAREN, ')', 13, 104)
LexToken(SEMICOLON, ';', 13, 105)
LexToken(RBRACE, '}', 14, 107)
LexToken(RBRACE, '}', 15, 109)
```

Parsing and Semantic Analysis:

optional preprocessor directives recognized

empty declaration list recognized

optional declaration list recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

factor recognized

term recognized

expression recognized

init declarator recognized

init declarator list recognized

declaration recognized

statement recognized

statement list recognized

pre-type specifier or empty recognized

non-empty pre-type specifier recognized

base type recognized

complex type specifier recognized

factor recognized

term recognized

expression recognized

init declarator recognized

init declarator list recognized

declaration recognized

statement recognized

statement list recognized

factor recognized

term recognized

expression recognized

factor recognized

term recognized

expression recognized

factor recognized

```
function call recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
while statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized
{'x': {'type': ' int', 'lineno': 0},
'babu': {'type': ' float', 'lineno': 0},
'main': {'type': ' int', 'lineno': 2}}
```

Teste test_case3.c:

```
Symbol Table initialized.
Tokens:
LexToken(INT,'int',1,0)
LexToken(MAIN,'main',1,4)
LexToken(LPAREN,'(',1,8)
LexToken(RPAREN,')',1,9)
LexToken(LBRACE,'{',1,11)
LexToken(INT,'int',3,18)
LexToken(ID,'num',3,22)
LexToken(EQUALS,'=',3,26)
LexToken(INTEGER,'6',3,28)
LexToken(SEMICOLON,';',3,29)
LexToken(INT,'int',4,35)
LexToken(ID,'count_double',4,39)
LexToken(EQUALS,'=',4,52)
LexToken(INTEGER,'0',4,54)
LexToken(SEMICOLON,';',4,55)
LexToken(WHILE,'while',6,62)
LexToken(LPAREN,'(',6,68)
LexToken(ID,'num',6,70)
LexToken(LE,'<=',6,74)
```

```
LexToken(INTEGER,'1',6,76)
LexToken(RPAREN,')',6,78)
LexToken(LBRACE,'{',6,80)
LexToken(ID,'count_double',7,87)
LexToken(EQUALS,'=',7,100)
LexToken(ID,'count_double',7,102)
LexToken(PLUS,'+',7,114)
LexToken(INTEGER,'2',7,115)
LexToken(SEMICOLON,';',7,116)
LexToken(RBRACE,'}',8,122)
LexToken(RBRACE,'}',9,125)
```

Parsing and Semantic Analysis:

```
optional preprocessor directives recognized
empty declaration list recognized
optional declaration list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
```



```
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
while statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized
{'num': {'type': ' int', 'lineno': 0},
'count_double': {'type': ' int', 'lineno': 0},
'main': {'type': ' int', 'lineno': 1}}
```

Teste test_case4.c:

```
Symbol Table initialized.
Tokens:
LexToken(HASH, '#', 1, 0)
```

```
LexToken(INCLUDE, 'include', 1, 1)
LexToken(LIBRARY, '<stdio.h>', 1, 9)
LexToken(VOID, 'void', 3, 20)
LexToken(ID, 'printMessage', 3, 25)
LexToken(LPAREN, '(', 3, 37)
LexToken(RPAREN, ')', 3, 38)
LexToken(LBRACE, '{', 3, 40)
LexToken(ID, 'printf', 4, 46)
LexToken(LPAREN, '(', 4, 52)
LexToken(String, '"Hello, World!\\n"', 4, 53)
LexToken(RPAREN, ')', 4, 70)
LexToken(SEMICOLON, ';', 4, 71)
LexToken(RBRACE, '}', 5, 73)
LexToken(INT, 'int', 7, 76)
LexToken(ID, 'add', 7, 80)
LexToken(LPAREN, '(', 7, 83)
LexToken(INT, 'int', 7, 84)
LexToken(ID, 'a', 7, 88)
LexToken(COMMA, ',', 7, 89)
LexToken(INT, 'int', 7, 91)
LexToken(ID, 'b', 7, 95)
LexToken(RPAREN, ')', 7, 96)
LexToken(LBRACE, '{', 7, 98)
LexToken(RETURN, 'return', 8, 104)
LexToken(ID, 'a', 8, 111)
LexToken(PLUS, '+', 8, 113)
LexToken(ID, 'b', 8, 115)
LexToken(SEMICOLON, ';', 8, 116)
LexToken(RBRACE, '}', 9, 118)
LexToken(INT, 'int', 11, 121)
LexToken(MAIN, 'main', 11, 125)
LexToken(LPAREN, '(', 11, 129)
LexToken(RPAREN, ')', 11, 130)
LexToken(LBRACE, '{', 11, 132)
LexToken(INT, 'int', 12, 138)
LexToken(ID, 'result', 12, 142)
LexToken(EQUALS, '=', 12, 149)
LexToken(ID, 'add', 12, 151)
LexToken(LPAREN, '(', 12, 154)
```

```
LexToken(INTEGER,'3',12,155)
LexToken(COMMA,',',12,156)
LexToken(INTEGER,'4',12,158)
LexToken(RPAREN,')',12,159)
LexToken(SEMICOLON,';',12,160)
LexToken(IF,'if',13,166)
LexToken(LPAREN,'(',13,169)
LexToken(ID,'result',13,170)
LexToken(GT,'>',13,177)
LexToken(INTEGER,'0',13,179)
LexToken(RPAREN,')',13,180)
LexToken(LBRACE,'{',13,182)
LexToken(ID,'printMessage',14,192)
LexToken(LPAREN,'(',14,204)
LexToken(RPAREN,')',14,205)
LexToken(SEMICOLON,';',14,206)
LexToken(RBRACE,'}',15,212)
LexToken(ELSE,'else',16,218)
LexToken(LBRACE,'{',16,223)
LexToken(ID,'printf',17,233)
LexToken(LPAREN,'(',17,239)
LexToken(String,'"Result is less than or equal to 0\\n"',17,240)
LexToken(RPAREN,')',17,277)
LexToken(SEMICOLON,';',17,278)
LexToken(RBRACE,'}',18,284)
LexToken(RETURN,'return',19,290)
LexToken(INTEGER,'0',19,297)
LexToken(SEMICOLON,';',19,298)
LexToken(RBRACE,'}',20,300)
```

Parsing and Semantic Analysis:

```
include directive recognized
preprocessor directive recognized
preprocessor directive list recognized
optional preprocessor directives recognized
empty declaration list recognized
optional declaration list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
```

base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized
statement recognized
statement list recognized
compound statement recognized
function recognized
function list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
parameter recognized
parameter list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
parameter recognized
parameter list recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
return statement recognized
statement recognized
statement list recognized
compound statement recognized
function recognized

function list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
argument list recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
function call recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
factor recognized
term recognized
expression recognized

argument list recognized
function call recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
if statement recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
return statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized
{'printMessage': {'type': ' void', 'lineno': 3},
'a': {'type': ' int', 'lineno': 7},
'b': {'type': ' int', 'lineno': 7},
'add': {'type': ' int', 'lineno': 7},
'result': {'type': ' int', 'lineno': 0},
'main': {'type': ' int', 'lineno': 11}}

Teste test_case5.c:

Tokens:

LexToken(HASH, '#', 1, 0)
LexToken(INCLUDE, 'include', 1, 1)
LexToken(LIBRARY, '<stdio.h>', 1, 9)
LexToken(HASH, '#', 2, 19)
LexToken(INCLUDE, 'include', 2, 20)
LexToken(LIBRARY, '<stdlib.h>', 2, 28)
LexToken(HASH, '#', 4, 40)
LexToken(DEFINE, 'define', 4, 41)
LexToken(ID, 'MAX', 4, 48)
LexToken(INTEGER, '100', 4, 52)

```
LexToken(TYPEDEF, 'typedef', 7, 100)
LexToken(STRUCT, 'struct', 7, 108)
LexToken(LBRACE, '{', 8, 115)
LexToken(FLOAT, 'float', 9, 121)
LexToken(ID, 'x', 9, 127)
LexToken(SEMICOLON, ';', 9, 128)
LexToken(FLOAT, 'float', 10, 134)
LexToken(ID, 'y', 10, 140)
LexToken(SEMICOLON, ';', 10, 141)
LexToken(RBRACE, '}', 11, 143)
LexToken(ID, 'Ponto2D', 11, 145)
LexToken(SEMICOLON, ';', 11, 152)
LexToken(FLOAT, 'float', 14, 209)
LexToken(ID, 'distanciaEntrePontos', 14, 215)
LexToken(LPAREN, '(', 14, 235)
LexToken(ID, 'Ponto2D', 14, 236)
LexToken(ID, 'p1', 14, 244)
LexToken(COMMA, ',', 14, 246)
LexToken(ID, 'Ponto2D', 14, 248)
LexToken(ID, 'p2', 14, 256)
LexToken(RPAREN, ')', 14, 258)
LexToken(LBRACE, '{', 15, 260)
LexToken(FLOAT, 'float', 16, 266)
LexToken(ID, 'dx', 16, 272)
LexToken(EQUALS, '=', 16, 275)
LexToken(ID, 'p1', 16, 277)
LexToken(DOT, '.', 16, 279)
LexToken(ID, 'x', 16, 280)
LexToken(MINUS, '-', 16, 282)
LexToken(ID, 'p2', 16, 284)
LexToken(DOT, '.', 16, 286)
LexToken(ID, 'x', 16, 287)
LexToken(SEMICOLON, ';', 16, 288)
LexToken(FLOAT, 'float', 17, 294)
LexToken(ID, 'dy', 17, 300)
LexToken(EQUALS, '=', 17, 303)
LexToken(ID, 'p1', 17, 305)
LexToken(DOT, '.', 17, 307)
LexToken(ID, 'y', 17, 308)
```

```
LexToken(MINUS, '-', 17, 310)
LexToken(ID, 'p2', 17, 312)
LexToken(DOT, '.', 17, 314)
LexToken(ID, 'y', 17, 315)
LexToken(SEMICOLON, ';', 17, 316)
LexToken(RETURN, 'return', 18, 322)
LexToken(ID, 'sqrt', 18, 329)
LexToken(LPAREN, '(', 18, 333)
LexToken(ID, 'dx', 18, 334)
LexToken(TIMES, '*', 18, 337)
LexToken(ID, 'dx', 18, 339)
LexToken(PLUS, '+', 18, 342)
LexToken(ID, 'dy', 18, 344)
LexToken(TIMES, '*', 18, 347)
LexToken(ID, 'dy', 18, 349)
LexToken(RPAREN, ')', 18, 351)
LexToken(SEMICOLON, ';', 18, 352)
LexToken(RBRACE, '}', 19, 395)
LexToken(INT, 'int', 21, 398)
LexToken(MAIN, 'main', 21, 402)
LexToken(LPAREN, '(', 21, 406)
LexToken(RPAREN, ')', 21, 407)
LexToken(LBRACE, '{', 22, 409)
LexToken(ID, 'Ponto2D', 23, 415)
LexToken(ID, 'ponto1', 23, 423)
LexToken(EQUALS, '=', 23, 430)
LexToken(LBRACE, '{', 23, 432)
LexToken(FLOAT_N, 2.5, 23, 433)
LexToken(COMMA, ',', 23, 436)
LexToken(FLOAT_N, 3.1, 23, 438)
LexToken(RBRACE, '}', 23, 441)
LexToken(SEMICOLON, ';', 23, 442)
LexToken(ID, 'Ponto2D', 24, 448)
LexToken(ID, 'ponto2', 24, 456)
LexToken(EQUALS, '=', 24, 463)
LexToken(LBRACE, '{', 24, 465)
LexToken(FLOAT_N, 4.6, 24, 466)
LexToken(COMMA, ',', 24, 469)
LexToken(FLOAT_N, 5.0, 24, 471)
```



```
LexToken(RBRACE, '}', 24, 474)
LexToken(SEMICOLON, ';', 24, 475)
LexToken(FLOAT, 'float', 27, 542)
LexToken(ID, 'distancia', 27, 548)
LexToken(EQUALS, '=', 27, 558)
LexToken(ID, 'distanciaEntrePontos', 27, 560)
LexToken(LPAREN, '(', 27, 580)
LexToken(ID, 'ponto1', 27, 581)
LexToken(COMMA, ',', 27, 587)
LexToken(ID, 'ponto2', 27, 589)
LexToken(RPAREN, ')', 27, 595)
LexToken(SEMICOLON, ';', 27, 596)
LexToken(ID, 'printf', 28, 602)
LexToken(LPAREN, '(', 28, 608)
LexToken(String, '"A distância entre os pontos é: %.2f\\n"', 28, 609)
LexToken(COMMA, ',', 28, 648)
LexToken(ID, 'distancia', 28, 650)
LexToken(RPAREN, ')', 28, 659)
LexToken(SEMICOLON, ';', 28, 660)
LexToken(FOR, 'for', 31, 705)
LexToken(LPAREN, '(', 31, 709)
LexToken(INT, 'int', 31, 710)
LexToken(ID, 'i', 31, 714)
LexToken(EQUALS, '=', 31, 716)
LexToken(INTEGER, '0', 31, 718)
LexToken(SEMICOLON, ';', 31, 719)
LexToken(ID, 'i', 31, 721)
LexToken(LT, '<', 31, 723)
LexToken(ID, 'MAX', 31, 725)
LexToken(SEMICOLON, ';', 31, 728)
LexToken(ID, 'i', 31, 730)
LexToken(INCREMENT, '++', 31, 731)
LexToken(RPAREN, ')', 31, 733)
LexToken(LBRACE, '{', 32, 739)
LexToken(IF, 'if', 33, 749)
LexToken(LPAREN, '(', 33, 752)
LexToken(ID, 'i', 33, 753)
LexToken(MOD, '%', 33, 755)
LexToken(INTEGER, '2', 33, 757)
```

```
LexToken(COMPARATOR, '==', 33, 759)
LexToken(INTEGER, '0', 33, 762)
LexToken(RPAREN, ')', 33, 763)
LexToken(LBRACE, '{', 34, 773)
LexToken(ID, 'printf', 35, 787)
LexToken(LPAREN, '(', 35, 793)
LexToken(String, '"Número par: %d\\n"', 35, 794)
LexToken(COMMA, ',', 35, 812)
LexToken(ID, 'i', 35, 814)
LexToken(RPAREN, ')', 35, 815)
LexToken(SEMICOLON, ';', 35, 816)
LexToken(RBRACE, '}', 36, 826)
LexToken(ELSE, 'else', 37, 836)
LexToken(LBRACE, '{', 38, 849)
LexToken(ID, 'printf', 40, 913)
LexToken(LPAREN, '(', 40, 919)
LexToken(String, '"Número ímpar: %d\\n"', 40, 920)
LexToken(COMMA, ',', 40, 940)
LexToken(ID, 'i', 40, 942)
LexToken(RPAREN, ')', 40, 943)
LexToken(SEMICOLON, ';', 40, 944)
LexToken(RBRACE, '}', 41, 954)
LexToken(RBRACE, '}', 42, 960)
LexToken(INT, 'int', 43, 966)
LexToken(ID, 'i', 43, 970)
LexToken(EQUALS, '=', 43, 972)
LexToken(INTEGER, '0', 43, 974)
LexToken(SEMICOLON, ';', 43, 975)
LexToken(ID, 'i', 44, 981)
LexToken(INCREMENT, '++', 44, 982)
LexToken(SEMICOLON, ';', 44, 984)
LexToken(ID, 'i', 45, 990)
LexToken(DECREMENT, '--', 45, 991)
LexToken(SEMICOLON, ';', 45, 993)
LexToken(RETURN, 'return', 46, 999)
LexToken(INTEGER, '0', 46, 1006)
LexToken(SEMICOLON, ';', 46, 1007)
LexToken(RBRACE, '}', 47, 1009)
```

Parsing:

include directive recognized
preprocessor directive recognized
preprocessor directive list recognized
include directive recognized
preprocessor directive recognized
preprocessor directive list recognized
factor recognized
term recognized
expression recognized
define directive recognized
preprocessor directive recognized
preprocessor directive list recognized
optional preprocessor directives recognized
empty declaration list recognized
optional id recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
struct member recognized
struct members recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
struct member recognized
struct members recognized
struct declaration recognized
typedef declaration recognized
declaration recognized
declaration list extended
optional declaration list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
simple type specifier recognized
parameter recognized

parameter list recognized
simple type specifier recognized
parameter recognized
parameter list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
expression recognized
factor recognized
term recognized
expression recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
expression recognized
factor recognized
term recognized
expression recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized

term recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized
expression recognized
return statement recognized
statement recognized
statement list recognized
compound statement recognized
function recognized
function list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
simple type specifier recognized
factor recognized
term recognized
expression recognized
initializer list recognized
factor recognized
term recognized
expression recognized
initializer list recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
simple type specifier recognized
factor recognized
term recognized

expression recognized
initializer list recognized
factor recognized
term recognized
expression recognized
initializer list recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
argument list recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
argument list recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized

statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
factor recognized
term recognized
expression recognized
argument list recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized

statement recognized
statement list recognized
compound statement recognized
statement recognized
factor recognized
term recognized
expression recognized
argument list recognized
factor recognized
term recognized
expression recognized
argument list recognized
function call recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
if statement recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
for statement recognized
statement recognized
statement list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
factor recognized
term recognized
expression recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized

expression recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
return statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized

Teste test_case6.c:

Symbol Table initialized.
Tokens:
LexToken(INT, 'int', 2, 1)
LexToken(MAIN, 'main', 2, 5)
LexToken(LPAREN, '(', 2, 9)
LexToken(RPAREN, ')', 2, 10)
LexToken(LBRACE, '{', 3, 12)
LexToken(UNSIGNED, 'unsigned', 4, 18)
LexToken(INT, 'int', 4, 27)
LexToken(ID, 'a', 4, 31)
LexToken(COMMA, ',', 4, 32)
LexToken(ID, 'b', 4, 34)
LexToken(COMMA, ',', 4, 35)
LexToken(ID, 'resultado', 4, 37)
LexToken(SEMICOLON, ';', 4, 46)
LexToken(ID, 'a', 5, 52)
LexToken(EQUALS, '=', 5, 54)
LexToken(INTEGER, '4', 5, 56)

```
LexToken(SEMICOLON, ';' ,5,57)
LexToken(ID, 'b' ,6,63)
LexToken(EQUALS, '=' ,6,65)
LexToken(INTEGER, '6' ,6,67)
LexToken(SEMICOLON, ';' ,6,68)
LexToken(IF, 'if' ,7,74)
LexToken(LPAREN, '(' ,7,77)
LexToken(ID, 'a' ,7,78)
LexToken(GE, '>=' ,7,80)
LexToken(INTEGER, '10' ,7,83)
LexToken(RPAREN, ')' ,7,85)
LexToken(LBRACE, '{' ,8,91)
LexToken(ID, 'resultado' ,9,101)
LexToken(EQUALS, '=' ,9,111)
LexToken(ID, 'a' ,9,113)
LexToken(MINUS, '-' ,9,115)
LexToken(ID, 'b' ,9,117)
LexToken(SEMICOLON, ';' ,9,118)
LexToken(RBRACE, '}' ,10,124)
LexToken(ELSE, 'else' ,11,130)
LexToken(LBRACE, '{' ,12,139)
LexToken(ID, 'resultado' ,13,149)
LexToken(EQUALS, '=' ,13,159)
LexToken(ID, 'a' ,13,161)
LexToken(PLUS, '+' ,13,163)
LexToken(ID, 'b' ,13,165)
LexToken(SEMICOLON, ';' ,13,166)
LexToken(RBRACE, '}' ,14,172)
LexToken(RBRACE, '}' ,15,174)
```

Parsing and Semantic Analysis:

```
optional preprocessor directives recognized
empty declaration list recognized
optional declaration list recognized
pre-type specifier or empty recognized
non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
pre-type specifier or empty recognized
```

non-empty pre-type specifier recognized
base type recognized
complex type specifier recognized
init declarator recognized
init declarator list recognized
init declarator recognized
init declarator list recognized
init declarator recognized
init declarator list recognized
declaration recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
factor recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized

```
statement list recognized
compound statement recognized
statement recognized
factor recognized
term recognized
expression recognized
factor recognized
term recognized
expression recognized
assignment recognized
statement recognized
statement list recognized
compound statement recognized
statement recognized
if statement recognized
statement recognized
statement list recognized
compound statement recognized
main function recognized
function recognized
function list recognized
program recognized
{'a': {'type': 'unsigned int', 'lineno': 0},
'b': {'type': 'unsigned int', 'lineno': 0},
'resultado': {'type': 'unsigned int', 'lineno': 0},
'main': {'type': 'int', 'lineno': 2}}
```

Os resultados das análises mostram que o compilador parcial é capaz de reconhecer corretamente os tokens, analisar a estrutura gramatical e verificar a consistência semântica do código fonte para a maioria dos casos de teste. No entanto, algumas limitações foram encontradas, especialmente no caso de teste `test_case5.c`, onde a parte semântica não está completa.

7.2 TESTE DE VERIFICAÇÃO DE TIPOS

Para validar a implementação da verificação de compatibilidade de tipos, um teste específico foi realizado modificando o arquivo `test_case6.c`. Este teste foi projetado para verificar se o analisador semântico pode identificar corretamente atribuições incompatíveis de tipos.

O código modificado do `test_case6.c` é o seguinte:

```
int main()
{
    unsigned int a, b;
    char resultado;
    a = 4;
    b = 6;
    if (a >= 10)
    {
        resultado = a - b;
    }
    else
    {
        resultado = a + b;
    }
}
```

O objetivo deste teste é verificar se o analisador semântico identifica a atribuição de um valor do tipo `unsigned int` para uma variável do tipo `char`. A expectativa é que o analisador semântico levante uma exceção devido à incompatibilidade de tipos.

A execução deste teste retornou o seguinte resultado:

```
raise Exception(f"Type error: Cannot assign '{expr_type}'
to variable '{p[1]}' of type '{var_info['type']}'".)
Exception: Type error: Cannot assign 'unsigned int'
to variable 'resultado' of type ' char'.
```

Este resultado confirma que o analisador semântico está funcionando corretamente ao detectar a incompatibilidade de tipos entre a variável `resultado`, que é do tipo `char`, e a expressão `a - b`, que resulta em um valor do tipo `unsigned int`. Esta verificação é crucial para garantir a robustez e a segurança do código, prevenindo possíveis erros durante a execução do programa.

7.3 DESAFIOS ENCONTRADOS

Durante o desenvolvimento do compilador parcial, foram encontrados vários desafios:

- ****Gerenciamento de Escopos****: Implementar o gerenciamento de escopos foi desafiador, especialmente para estruturas aninhadas e funções.

- ****Compatibilidade de Tipos****: Verificar a compatibilidade de tipos em atribuições e operações aritméticas/lógicas foi complexo, exigindo uma verificação minuciosa de cada operação.
- ****Erro de Declaração de Variáveis****: Garantir que todas as variáveis fossem declaradas antes de serem usadas envolveu a criação de uma tabela de símbolos eficiente.
- ****Declaração de Funções****: Verificar a declaração de funções e garantir que apenas funções conhecidas fossem utilizadas exigiu atenção especial nas regras semânticas.
- ****Declaração de Structs****: Implementar a verificação correta para a declaração e uso de structs apresentou desafios, especialmente na parte semântica.

7.4 MELHORIAS E TRABALHOS FUTUROS

Para melhorar o compilador parcial e expandir suas capacidades, algumas sugestões de melhorias e trabalhos futuros incluem:

- ****Completar a Análise Semântica****: Implementar as regras semânticas restantes, especialmente para o caso de teste `test_case5.c`.
- ****Otimização de Código****: Implementar otimizações de código para melhorar a eficiência e desempenho do compilador.
- ****Suporte a Mais Estruturas****: Adicionar suporte para mais estruturas da linguagem C, como ponteiros e arrays.
- ****Melhoria na Detecção de Erros****: Aprimorar a detecção e tratamento de erros sintáticos e semânticos para fornecer mensagens de erro mais informativas.
- ****Integração com Ferramentas de Análise Estática****: Integrar o compilador com ferramentas de análise estática para fornecer uma verificação de código mais robusta.

8 CONCLUSÃO

8.1 RESUMO DO PROJETO

Este projeto teve como objetivo a implementação parcial de um compilador para a linguagem C, abrangendo as fases de análise léxica, análise sintática e análise semântica. A análise léxica foi responsável por identificar e classificar tokens no código fonte. A análise sintática validou a estrutura gramatical do código e construiu a árvore sintática abstrata (AST). A análise semântica verificou a consistência semântica do código, garantindo a correta declaração e uso de variáveis, bem como a compatibilidade de tipos em operações e atribuições.

8.2 CONCLUSÕES FINAIS

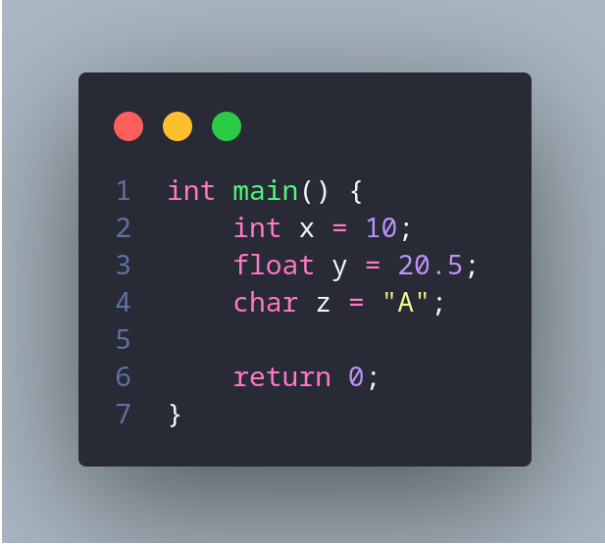
A implementação deste compilador parcial permitiu aplicar na prática diversos conceitos teóricos abordados ao longo do curso. A análise léxica, através do reconhecimento de tokens e expressões regulares, mostrou a importância de identificar corretamente os elementos básicos da linguagem de programação. A análise sintática, com a construção da árvore sintática abstrata (AST) e a verificação da estrutura gramatical, destacou a necessidade de uma gramática bem definida e a aplicação da notação BNF (Backus-Naur Form).

A análise semântica reforçou a importância do gerenciamento de escopos e da verificação de compatibilidade de tipos, garantindo que o código fonte seja não apenas sintaticamente correto, mas também semanticamente consistente. A construção e o gerenciamento da tabela de símbolos foram essenciais para rastrear a declaração e o uso das variáveis ao longo do código.

A prática de implementar um compilador parcial permitiu uma compreensão mais aprofundada dos processos envolvidos na compilação, consolidando o aprendizado teórico com a aplicação prática desses conceitos. A experiência prática contribuiu para um entendimento mais sólido e integrado dos processos envolvidos na compilação, desde a análise léxica até a verificação semântica.


Em suma, a implementação das ações semânticas mostrou como a verificação de contexto é crucial para evitar erros em tempo de execução, como o uso de variáveis não declaradas ou operações entre tipos incompatíveis. A aplicação das regras de precedência e associatividade dos operadores foi fundamental para o desenvolvimento do analisador sintático. Assim, o desenvolvimento deste compilador parcial não só reforçou os conceitos teóricos aprendidos, mas também proporcionou uma visão mais ampla e aplicada dos desafios e soluções no campo da compilação.

APÊNDICE A – IMAGENS DOS CÓDIGOS DE TESTE

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted C language. It defines a `main` function that initializes variables `x`, `y`, and `z`, and then returns 0.

```
1  int main() {  
2      int x = 10;  
3      float y = 20.5;  
4      char z = "A";  
5  
6      return 0;  
7  }
```

Figura 5 – Código do Test Case 0.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a syntax-highlighted C language. It defines a `main` function that initializes `age` and `level`, and then uses nested `if` statements to determine the `level` based on the `age`.

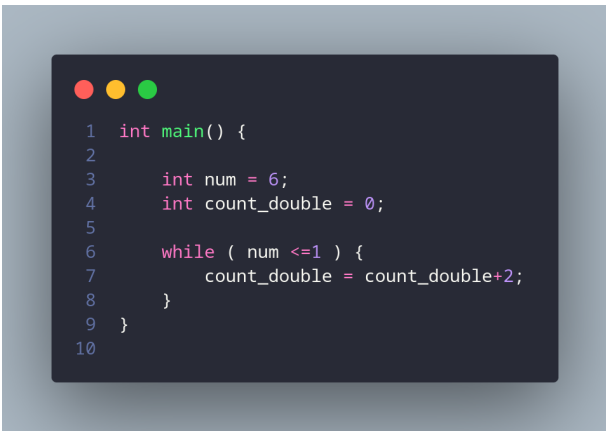
```
1  int main() {  
2  
3      int age = 6;  
4      int level = 0;  
5  
6      if (age < 18) {  
7          level = 1;  
8      } else {  
9          if (age <= 60) {  
10             level = 2;  
11             } else {  
12                 level = 3;  
13             }  
14         }  
15     }
```

Figura 6 – Código do Test Case 1.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in C and consists of 15 lines. It defines a main function that initializes a variable x to 2, prints a series of numbers (3, 4, 5, 6) with asterisks, and then enters a while loop that prints the value of x as long as it is greater than 3.

```
1
2  int main(){
3      // teste
4  int x = 2;
5  // 3
6  // 4
7  /* 5 */
8  /*
9  6
10 */
11 float babu = 2.3;
12 while(x>3){
13     printf();
14 }
15 }
```

Figura 7 – Código do Test Case 2.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in C and consists of 10 lines. It defines a main function that initializes a variable num to 6 and a variable count_double to 0. It then enters a while loop that increments count_double by 2 as long as num is greater than or equal to 1.

```
1  int main() {
2
3      int num = 6;
4      int count_double = 0;
5
6      while ( num >= 1 ) {
7          count_double = count_double+2;
8      }
9  }
10
```

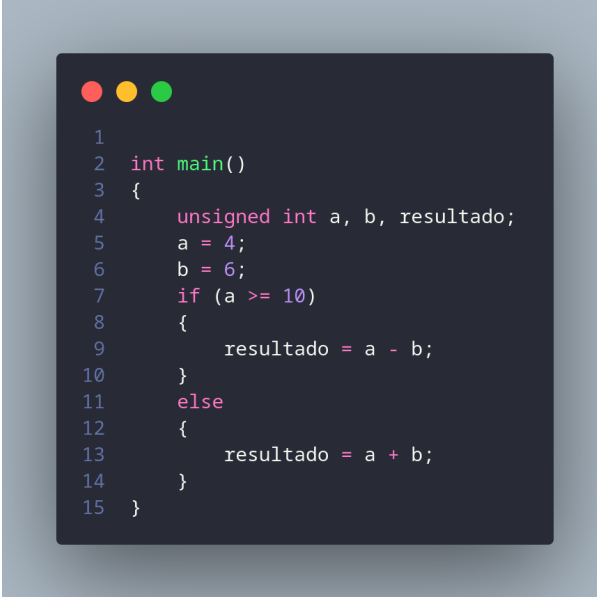
Figura 8 – Código do Test Case 3.

```
1 #include <stdio.h>
2
3 void printMessage() {
4     printf("Hello, World!\n");
5 }
6
7 int add(int a, int b) {
8     return a + b;
9 }
10
11 int main() {
12     int result = add(3, 4);
13     if (result > 0) {
14         printMessage();
15     }
16     else {
17         printf("Result is less than or equal to 0\n");
18     }
19     return 0;
20 }
21
```

Figura 9 – Código do Test Case 4.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 100
5
6 // Estrutura para um ponto em um espaço 2D
7 typedef struct
8 {
9     float x;
10    float y;
11 } Ponto2D;
12
13 // Função para calcular a distância entre dois pontos
14 float distanciaEntrePontos(Ponto2D p1, Ponto2D p2)
15 {
16     float dx = p1.x - p2.x;
17     float dy = p1.y - p2.y;
18     return sqrt(dx * dx + dy * dy); // Uso da função sqrt da biblioteca math
19 }
20
21 int main()
22 {
23     Ponto2D ponto1 = {2.5, 3.1};
24     Ponto2D ponto2 = {4.6, 5.0};
25
26     /* Cálculo e exibição da distância entre dois pontos */
27     float distancia = distanciaEntrePontos(ponto1, ponto2);
28     printf("A distância entre os pontos é: %.2f\n", distancia);
29
30     // Exemplo de loop e condicionais
31     for (int i = 0; i < MAX; i++)
32     {
33         if (i % 2 == 0)
34         {
35             printf("Número par: %d\n", i);
36         }
37         else
38         {
39             // Comentário dentro de um bloco else
40             printf("Número ímpar: %d\n", i);
41         }
42     }
43     int i = 0;
44     i++;
45     i--;
46     return 0;
47 }
```

Figura 10 – Código do Test Case 5.



```
1
2  int main()
3  {
4      unsigned int a, b, resultado;
5      a = 4;
6      b = 6;
7      if (a >= 10)
8      {
9          resultado = a - b;
10     }
11     else
12     {
13         resultado = a + b;
14     }
15 }
```

Figura 11 – Código do Test Case 6.