Católica
Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Bacharelado em Ciência da Computação
Trabalho da Disciplina Linguagens Formais, Autômatos e Compiladores .

Analisador sintático

Autores:
João Victor Ribeiro Oliveira;
Juan Alberto Bezerra Jeronimo;
Júlia Gabriela Gomes Da Silva;
Lara Ewellen De Carvalho Rocha
Leandro Veras De Souza.
Orientador: Prof. MARCELO EUSTAQUIO SOARES DE LIMA JUNIOR

Brasília - DF
2024

JOÃO VICTOR RIBEIRO OLIVEIRA;
JUAN ALBERTO BEZERRA JERONIMO;
JÚLIA GABRIELA GOMES DA SILVA;
LARA EWELLEN DE CARVALHO ROCHA;
LEANDRO VERAS DE SOUZA.

Analisador sintático

Documento apresentado ao Curso de graduação de Bacharelado em Ciência da Computação da Universidade Católica de Brasília, como requisito parcial para obtenção da aprovação na disciplina de Linguagens Formais, autômatos e compiladores.

Orientador: Prof. MARCELO EUSTAQUIO SOARES DE LIMA JUNIOR

Brasília
2024

RESUMO

Este trabalho apresenta o desenvolvimento de um analisador léxico e sintático para a linguagem MicroPascal (m-Pascal), que ao receber um código-fonte como entrada, possa identificar e categorizar tokens e analisar sua conformidade com a gramática formal especificada. Para isso, utilizamos a linguagem de programação C, que nos permitiu construir um analisador do zero, desde a identificação de padrões de tokens até a validação da estrutura sintática do código. Esse desenvolvimento não só reforçou o entendimento prático dos conceitos de análise léxica e sintática como também exigiu uma organização rigorosa de funções e estrutura de dados, além de uma abordagem cuidadosa para o tratamento de erros. Este relatório apresenta detalhadamente o processo de construção, explicações das principais funções e estruturas utilizadas, e os testes realizados com programas válidos e com erros. Os resultados confirmam que o analisador é capaz de processar adequadamente o código, fornecendo uma sequência de regras de produção para cada código correto e mensagens de erro detalhadas nos casos de erro.

Palavras-chave: Analisador léxico, Analisador sintático, MicroPascal, Linguagem C, Gramática formal, Tokens, Análise léxica, Análise sintática, Tratamento de erros, Estrutura de dados, Funções, Regras de produção, Testes.

ABSTRACT

This paper presents the development of a lexical and syntactic analyzer for the MicroPascal (m-Pascal) language, which, upon receiving source code as input, is able to identify and categorize tokens and analyze their conformity with the specified formal grammar. For this, we used the C programming language, which allowed us to build an analyzer from scratch, from token pattern recognition to syntactic structure validation. This development not only reinforced the practical understanding of lexical and syntactic analysis concepts but also required a rigorous organization of functions and data structures, as well as a careful approach to error handling. This report details the construction process, explains the main functions and structures used, and presents the tests conducted with valid programs and those with errors. The results confirm that the analyzer is capable of properly processing the code, providing a sequence of production rules for each correct code and detailed error messages in case of errors.

Keywords: *Lexical analyzer, Syntax analyzer, MicroPascal, C programming language, Formal grammar, Tokens, Lexical analysis, Syntax analysis, Error handling, Data structures, Functions, Production rules, Testing*

SUMÁRIO

INTRODUÇÃO

Neste projeto, o desafio foi implementar do zero um analisador léxico e sintático para MicroPascal. Isso implicou a criação manual de uma função para identificar e validar tokens e uma função CasaToken para comparar os tokens esperados pela gramática com os tokens presentes no código-fonte. Um dos maiores desafios foi garantir que o analisador conseguisse interpretar as estruturas complexas da gramática, como declarações, comandos e expressões, em conformidade com as regras estabelecidas. Além disso, o desenvolvimento manual do tratamento de erros exigiu uma abordagem cuidadosa para identificar e relatar erros de maneira clara e precisa.

OBJETIVO

1.1 OBJETIVO GERAL

Desenvolver um analisador léxico e sintático em linguagem C para a linguagem MicroPascal, capaz de verificar a conformidade do código-fonte com uma gramática formal especificada, detectando e relatando erros de maneira detalhada e gerando uma saída que evidencie a sequência de regras de produção usadas para analisar programas válidos.

1.2 OBJETIVOS ESPECÍFICOS

1. Implementar a função **CasaToken** para verificar a correspondência entre o token esperado e o encontrado pelo analisador léxico.
2. Desenvolver procedimentos para a análise de cada símbolo não-terminal da gramática de MicroPascal.
3. Testar o analisador sintático com códigos-fonte válidos e inválidos em MicroPascal.
4. Documentar o desenvolvimento do analisador, explicando o funcionamento de cada função e estrutura de dados utilizada.

Estruturas de Dados

1. Token

Propósito: Representar um token identificado pelo analisador léxico.

- ➔ **tipo:** Tipo do token (e.g., `id`, `program`, `integer`).
- ➔ **valor:** Valor literal associado ao token, como o nome de um identificador ou o valor de uma constante.
- ➔ **linha:** Linha do código onde o token foi encontrado (para mensagens de erro).
- ➔ **coluna:** Coluna inicial do token no código-fonte (para mensagens detalhadas de erro).

2. ErroSintatico

Propósito: Representar informações sobre um erro sintático detectado.

- ➔ **mensagem:** Descrição detalhada do erro.
- ➔ **linha:** Linha onde o erro ocorreu.
- ➔ **coluna:** Coluna onde o erro ocorreu.
- ➔ **token_esperado:** Token ou símbolo esperado (opcional).

3. PilhaSintatica

Propósito: Simular a pilha utilizada pelo analisador durante a derivação das regras.

- ➔ **topo:** Índice do elemento no topo da pilha.
- ➔ **elementos:** Vetor para armazenar símbolos da gramática (não terminais e terminais).

Funções Principais

1. void inicializarAnalisador()

Propósito: Configurar o ambiente inicial do analisador sintático. Isso inclui inicializar a pilha sintática, carregar a gramática e preparar os dados de entrada.

Responsabilidades:

- ➔ Limpar estruturas de dados.

- ➔ Inicializar o vetor de tokens recebido do analisador léxico.
- ➔ Configurar a pilha inicial com o símbolo de partida da gramática (`S`).

2. void analisarTokens()

Propósito: Realizar a análise sintática percorrendo os tokens gerados pelo analisador léxico e aplicando as regras da gramática.

Responsabilidades:

- ➔ Consumir tokens sequencialmente e compará-los com os símbolos esperados pela gramática.
- ➔ Expansão de não terminais com base nas regras da gramática.
- ➔ Detectar e reportar erros sintáticos.

3. int consumirToken(char *esperado)

Propósito: Verificar se o próximo token na entrada corresponde ao símbolo esperado.

Parâmetros:

- ➔ esperado: O tipo de token ou símbolo terminal esperado.

Retorno:

- ➔ 1 (verdadeiro) se o token foi consumido com sucesso.
- ➔ 0 (falso) se o token não corresponde ao esperado.

Responsabilidades

- ➔ Avançar no vetor de tokens se o consumo for bem-sucedido.
- ➔ Gerar mensagens de erro caso haja discrepância.

4. void aplicarProducao(char *nao_terminal, char *regra)

Propósito: Expandir um não terminal na pilha com base em uma produção da gramática.

Parâmetros:

- ➔ nao_terminal: O símbolo não terminal no topo da pilha.
- ➔ regra: Produção que será aplicada.

Responsabilidades:

- ➔ Substituir o não terminal na pilha pelos símbolos presentes na regra da produção.
- ➔ Registrar a produção utilizada no processo de análise.

5. `void registrarErro(char *mensagem, char *token_esperado)

Propósito: Criar um objeto de erro sintático com as informações do problema identificado.
Parâmetros:

- ➔ mensagem: Descrição do erro.
- ➔ token_esperado: Token ou símbolo que era esperado, se aplicável.

Responsabilidades:

- ➔ Salvar a descrição e a localização do erro.
- ➔ Interromper a análise sintática se o erro for crítico.

6. void exibirProducao()

Propósito: Imprimir no console ou salvar em arquivo a sequência de regras de produção utilizadas durante a análise.
Responsabilidades:
Garantir que as regras sejam apresentadas na ordem correta.
Auxiliar na depuração e validação do analisador.

7. void finalizarAnalisador()

Propósito: Realizar limpeza de memória e finalização das estruturas de dados.
Responsabilidades:

- ➔ Liberar memória alocada dinamicamente.
- ➔ Gerar relatórios finais, se necessário.

Fluxo de Operação

1. **Entrada:**

O analisador recebe como entrada um vetor de tokens do analisador léxico.

2. **Inicialização:**

A pilha é inicializada com o símbolo inicial da gramática.

3. **Análise:**

A função analisarTokens consome tokens, compara com os símbolos esperados, e aplica produções conforme necessário.

Erros são registrados e reportados conforme surgem.

4. **Saída:**

Exibição da sequência de produções aplicadas.
Indicação de sucesso ou falha, com detalhamento dos erros encontrados.

Como o analisador funciona:

1. **Gramática**

O código contém funções que representam as regras de produção da gramática, como `AnalisarPrograma`, `AnalisarBloco`, `AnalisarComando`, entre outras. Essas funções correspondem aos não-terminais da gramática e são chamadas recursivamente para verificar a estrutura do código-fonte.

2. **Entrada do Analisador**

A função `ProximoToken` é responsável por fornecer tokens ao analisador, um de cada vez, a partir do arquivo fonte. Esses tokens incluem lexemas, tipos e números de linha.

3. **Saída do Analisador**

Sequência de Produções: Para cada código-fonte submetido, o analisador imprime as regras de produção aplicadas. Por exemplo:
rust
Copiar código

```
S -> program id ; S
S -> S ; S
```

- ➔ Essa saída é gerada pelas chamadas recursivas e pelos `printf` nas funções que correspondem aos não-terminais.
- ➔ **Erros Sintáticos:** Se o código não seguir a gramática esperada, mensagens de erro são exibidas usando a função `Erro`.

4. Integração com a Tabela de Símbolos

A tabela de símbolos (`tabelaSimbolos`) é usada para registrar identificadores e seus tipos. Isso ajuda na verificação de declarações de variáveis e evita símbolos não declarados.

5. Erros Tratados

O analisador também lida com:

- ➔ Fatores inválidos.
- ➔ Operadores relacionais ausentes.
- ➔ Falta de fechamento de blocos e comandos.
- ➔ Símbolos não declarados.

Programas certos

1-

```
program exemplo;
var
    x, y: integer;
    z: real;
begin
    x := 10;
    y := x + 2;
    if x > y then
        z := 1.5
    else
        z := 2.0;
    while x < 20 do
        x := x + 1;
end.
```

2-

```
program exemplo2;
var
    x, y: integer;
    z: real;
```

```
begin
    x := 10;
    y := x + 2;
    if x > y then
        z := 1.5
    else
        z := 2.0;
    while x < 20 do
        x := x + 1;
end.
```

3-

```
program exemplo3;
var
    a, b: integer;
    c: real;
begin
    a := 5;
    b := 10;
    c := a + b;  // Soma de inteiros, atribuição para real
    if c > 10 then
        b := b + 1;
    while b < 15 do
        b := b + 1;
end.
```

Programas errados

1-

```
program erro1;
var
    b: integer;
begin
    b := 8;
    if b > 5  { Erro aqui: falta o "then" após a condição do "if" }
        writeln('B é maior que 5')
    else
        writeln('Else executado');
```

```
end.
```

2-

```
program exemplo2;
var
    x, y: integer;
    z: real;
begin
    x := 10;
    y := x + 2;
    if x > y then
        z := 1.5
    else
        z := 2.0;
    while x < 20 do
        x := x + 1;
end.
```

3-

```
program exemplo3;
var
    a, b: integer;
    c: real;
begin
    a := 5;
    b := 10;
    c := a + b;  // Soma de inteiros, atribuição para real
    if c > 10 then
        b := b + 1;
    while b < 15 do
        b := b + 1;
end.
```

CONCLUSÃO

Este projeto representou um desafio importante para a compreensão e a aplicação prática dos fundamentos de análise léxica e sintática no desenvolvimento de compiladores.Exigiu um alto nível de rigor na implementação, desde o reconhecimento dos tokens até a interpretação correta das regras gramaticais.

A implementação manual do tratamento de erros foi particularmente desafiadora, exigindo uma estrutura robusta para identificar e relatar erros específicos e detalhados, como a linha do código e o lexema inesperado. O sucesso deste projeto reforça a importância de uma estrutura organizada e da compreensão profunda das regras gramaticais na criação de um analisador capaz de interpretar corretamente um código-fonte e fornecer feedback claro e informativo. O projeto fornece uma estrutura modular para análise sintática, permitindo fácil integração com o analisador léxico e relatórios claros para depuração. Cada função e struct desempenham um papel específico, garantindo que o analisador seja eficiente e mantenha o código organizado.