Universidade de São Paulo - São Carlos

Departamento de Ciências de Computação

Trabalho 1: Analisador Léxico do PL/0

Gustavo B. Sanchez
11802440

Mateus C. de Goes
12675997

Matheus dos Santos Inês
12546784

Docente: Prof. Thiago A. S. Pardo

SCC0605 - Teoria da Computação e Compiladores

São Carlos
Maio de 2024

Conteúdo

1 Introdução

A análise léxica é uma etapa fundamental do processo de compilação de linguagens de programação, nas quais o texto de entrada do código fonte é convertido em uma sequência de *tokens*, que são a base para a próxima etapa da compilação: a análise sintática. Assim, a análise léxica é crucial para a compreensão de como os compiladores interpretam e traduzem linguagens de programação para código executável.

Este projeto tem como objetivo desenvolver um analisador léxico para a linguagem PL/0, uma linguagem simplificada para fins educacionais no âmbito de design de compiladores. O analisador deverá ser capaz de ler um arquivo no formato *.txt* contendo código em PL/0 e produzir outro arquivo *.txt* com os *tokens* identificados conforme sua classe gramatical. Além disso, o analisador deverá identificar e reportar erros léxicos, melhorando a robustez e usabilidade da ferramenta.

2 Materiais e Métodos

2.1 Ferramentas Utilizadas

O desenho e teste dos autômatos foi realizado utilizando a ferramenta JFlap, seguindo o que se aprendeu durante as aulas da disciplina. Vale ressaltar que o JFlap não apenas possibilita a criação dos autômatos de forma visual, como também permite a execução de testes com strings de entrada, característica fundamental para validar a lógica antes de implementá-la em código.

Quanto ao código do analisador léxico, este foi desenvolvido utilizando a linguagem C, conforme indicado na descrição do trabalho. A familiaridade da equipe com essa linguagem facilitou a implementação dos autômatos e das estruturas de dados necessárias.

2.2 Decisões de Projeto

2.2.1 Estrutura de Dados

O analisador léxico foi implementado para processar a entrada de caracteres diretamente do arquivo fonte usando a função `fgetc`, que lê um caractere por vez. A função `ungetc` é utilizada para devolver um caractere não consumido ao fluxo de entrada, permitindo assim uma análise que antecipa o caractere seguinte (lookahead). Esta abordagem facilita o reconhecimento de padrões de tokens que exigem mais de um caractere para sua definição, como é o caso de operadores compostos (por exemplo, `:=` e `<=`).

2.2.2 Autômatos e Tokens

Os autômatos foram projetados para identificar todos os tipos de tokens definidos pela gramática PL/0, abrangendo identificadores, números, palavras-chave, símbolos e operadores. Uma decisão relevante foi tratar espaços em branco não como tokens, mas como elementos ignorados durante a análise, exceto quando sua presença impacta a validação de tokens adjacentes. Além disso, comentários foram tratados como um token único (desde a abertura de chaves `"{"`até que haja seu fechamento com `"}"`.

2.2.3 Tratamento de Erros

Foi desenvolvido um sistema de tratamento de erros para identificar e reportar erros léxicos de forma eficaz. Qualquer caractere que não corresponda a um autômato conhecido é reportado como erro léxico, incluindo caracteres especiais não reconhecidos e sequências malformadas, seja de identificadores ou números.

2.2.4 Testes

Implementou-se um conjunto abrangente de testes para assegurar a robustez do analisador. Estes testes cobrem todos os tipos de tokens e diversas condições de erro, servindo para validar tanto a precisão da análise léxica quanto a efetividade do sistema de tratamento de erros implementado.

3 Projeto dos Autômatos

Nesta seção, serão apresentados os autômatos finitos projetados para identificar e classificar os diversos componentes léxicos da linguagem PL/0. Cada autômato é especializado na detecção de uma categoria específica de tokens, incluindo comentários, espaços, identificadores alfanuméricos, e uma ampla gama de símbolos e operadores. As figuras a seguir ilustram a estrutura e as transições de cada autômato, permitindo uma compreensão visual do processo de análise léxica implementado para interpretar e preparar o código fonte para as fases subsequentes de compilação.

3.1 Autômato para Comentários

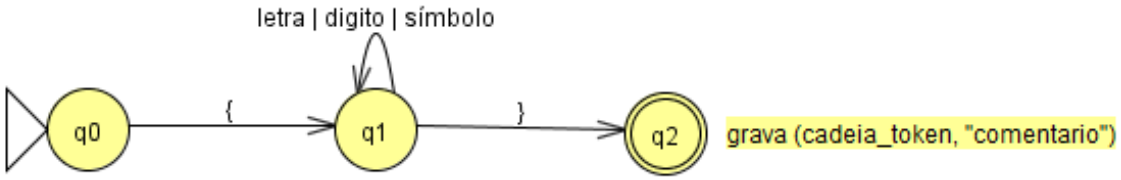

Figura 1: Diagrama do Autômato para Comentários

Este autômato demonstra o processo de reconhecimento de comentários na linguagem PL/0. Ao encontrar o caractere ‘{‘, o autômato transita para o estado q1, onde todos os caracteres subsequentes são lidos e ignorados até que o caractere ‘}‘ seja encontrado, indicando o fim do comentário. Neste ponto, o autômato grava a ocorrência de um comentário no arquivo de saída e retorna para a função principal, continuando a análise do código restante. Essa abordagem garante que o conteúdo dentro dos comentários não afete o processamento ou a análise do código fonte.

3.2 Autômato para Espaços

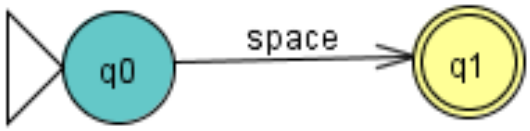

Figura 2: Diagrama do Autômato para Espaços

Este autômato trata a identificação e o manuseio de espaços em branco no código PL/0. Quando um espaço é encontrado no texto do programa, o autômato transita do estado inicial q0 para o estado q1. No estado q1, o espaço é reconhecido e ignorado pelo analisador, pois não afeta a semântica do programa. Este processo ajuda a simplificar o texto do código, removendo espaços desnecessários antes da análise sintática subsequente.

3.3 Autômato para Vírgulas

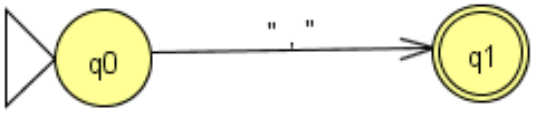

Figura 3: Diagrama do Autômato para Vírgulas

Este autômato trata a identificação e o manuseio de vírgulas no código PL/0. Quando uma vírgula é encontrado no texto do programa, o autômato transita do estado inicial q0 para o estado q1. No estado q1, o caractere vírgula é reconhecido e ignorado pelo analisador, pois não afeta a semântica do programa. Este processo, assim como o autômato para espaços, ajuda a simplificar o texto do código, removendo espaços desnecessários antes da análise sintática subsequente.

3.4 Autômato para Alfanuméricos

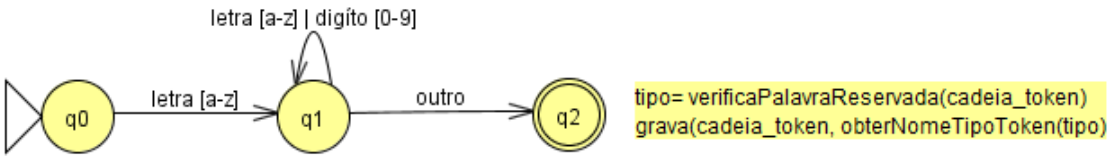

Figura 4: Diagrama do Autômato para Alfanuméricos

Este autômato é projetado para o processamento de tokens alfanuméricos no analisador léxico de PL/0. O processo começa no estado q0, onde, ao identificar o início de um token com uma letra (a-z), o autômato avança para o estado q1. Neste estado, ele continua a acumular letras e dígitos, construindo um token alfanumérico. Este estado captura toda a sequência de caracteres que formam um identificador ou palavra potencialmente reservada.

Quando um caractere não alfanumérico é encontrado, o autômato transita para o estado q2. Neste estado final, a função `verificaPalavraReservada` é chamada para determinar se o token acumulado é uma palavra reservada conhecida, como por exemplo "VAR", "CONST", ou "PROCEDURE". Se o token for reconhecido como uma palavra reservada, ele é categorizado de acordo com seu tipo específico. Caso contrário, ele é classificado como `ident`, indicando um identificador comum.

O token e o seu tipo são gravados no arquivo de saída, seja como uma palavra reservada específica ou como um identificador. Este mecanismo assegura que todos os identificadores e palavras-chave sejam corretamente identificados e classificados antes de prosseguirem para as próximas fases de processamento do compilador, facilitando a gestão dos elementos do código em etapas subsequentes exclusivamente no contexto da análise léxica.

3.5 Autômato para Números

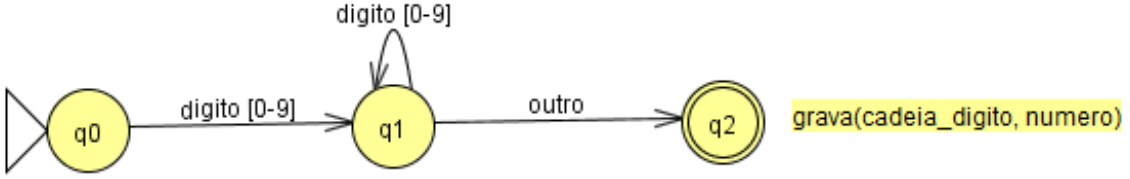

Figura 5: Diagrama do Autômato para Números

Este autômato identifica sequências numéricas em PL/0. Inicia no estado q0 e transita para o q1 ao detectar um dígito, acumulando mais dígitos até encontrar um caractere não numérico. Isso desencadeia a transição para o estado q2, onde o número completo é registrado como um token do tipo `numero`.

3.6 Autômato para Símbolos e Operadores Aceitos

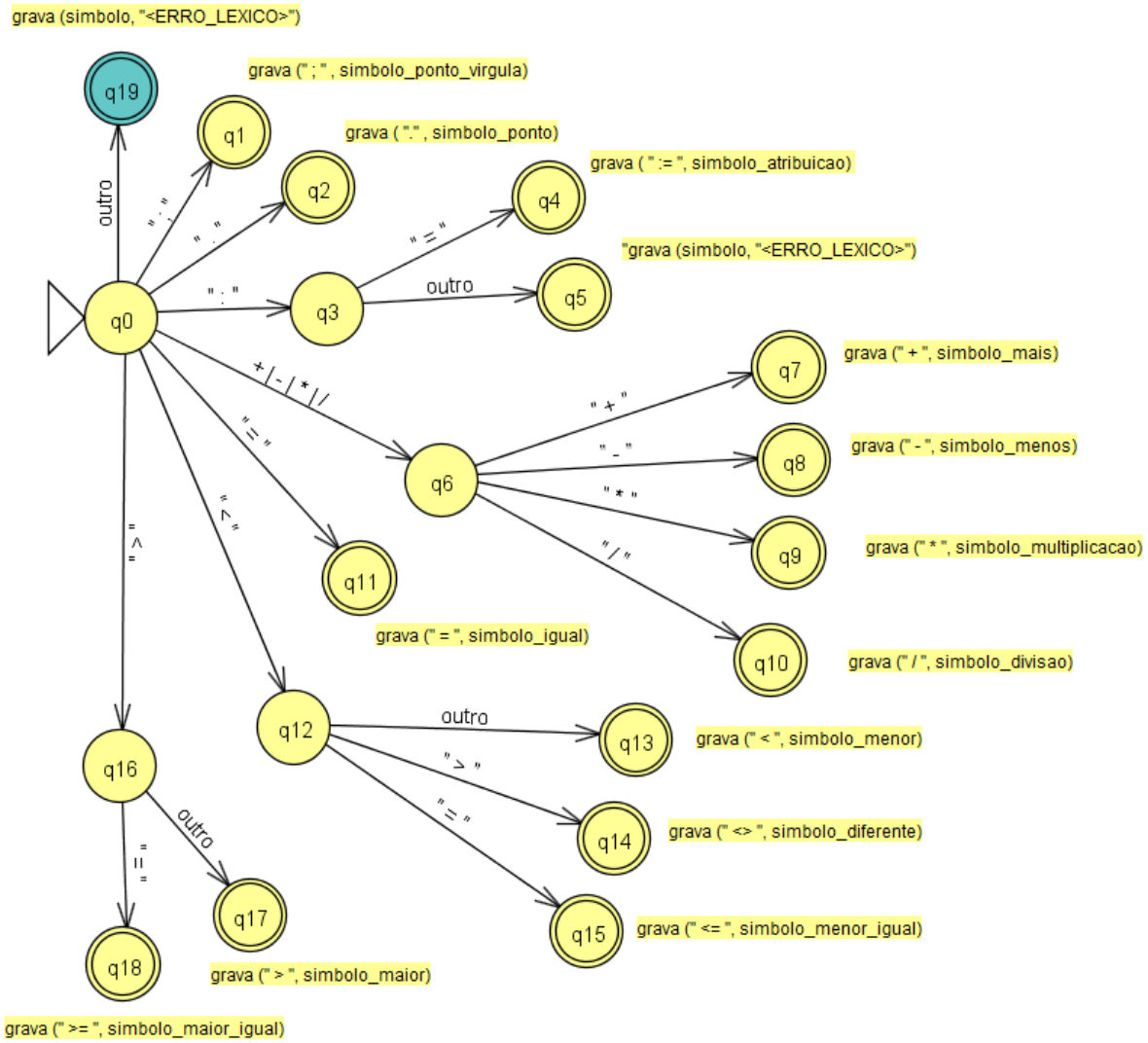

Figura 6: Diagrama do Autômato para Símbolos e Operadores Aceitos

O autômato apresentado para "Símbolos e Operadores Aceitos" ilustra de forma abrangente o método pelo qual o analisador léxico de PL/0 identifica e processa diversos símbolos e operadores, adaptando-se às variações que podem surgir na linguagem. O autômato começa no estado q0 e possui várias transições baseadas no símbolo inicial lido do código-fonte.

3.6.1 Visão Geral do Autômato:

Estado Inicial (q0):

A partir deste estado, o autômato verifica o primeiro caractere e direciona para diferentes estados baseados nesse caractere. Se o caractere é um operador ou símbolo conhecido como +, -, *, /, e assim por diante, o autômato transita diretamente para um estado que grava esse token específico. Para caracteres como =, <, >, e :, que podem formar operadores compostos (como <=, >=, :=), o autômato transita para estados intermediários onde lê o próximo caractere para confirmar se forma um operador composto ou se o caractere inicial deve ser gravado sozinho.

Estados Intermediários (q3, q6, q12, q16):

Esses estados são responsáveis por verificar a formação de operadores compostos. Por exemplo, após ler < no estado q0, se o próximo caractere é =, o autômato move-se para o estado q15 onde <= é gravado como "simbolo_menor_igual". Se o próximo caractere não forma um operador composto, o caractere inicial é gravado como um operador simples ("simbolo_menor").

Estado de Erro (q19):

Qualquer caractere que não corresponda a um operador ou símbolo conhecidos leva ao estado q19, onde é classificado como um "<ERRO_LEXICO>", destacando os caracteres inválidos ou inesperados no código.

Transição e Gravação:

Cada transição específica no autômato é projetada para lidar com a natureza única dos operadores em PL/0, decidindo rapidamente se um token deve ser gravado como um operador simples, um operador composto ou um erro. Este processo permite uma identificação precisa e eficiente dos elementos do código que são críticos para a análise e interpretação do programa.

4 Instruções do Código Fonte

Esta seção trata das instruções referentes a compilar e executar o código fonte do analisador léxico desenvolvido para a linguagem PL/0.

4.1 Requisitos do Sistema

- Compilador de C: GCC (Linux/macOS) ou MinGW (Windows);
- Editor de texto: Qualquer editor de texto para editar e analisar o código. O código da equipe foi desenvolvido com o VS Code;
- Terminal ou Prompt de Comando: Para executar comandos de compilação e execução, incluindo o uso de `make`.

4.2 Passos para Compilar e Executar

1. Baixar o código `main.c` e o `Makefile` incluídos no mesmo diretório do arquivo `.zip` deste relatório.
2. Abrir o terminal (Linux/macOS) ou prompt de comando/powershell (Windows).
3. Compilar o Código
   - Navegar até o diretório onde os arquivos foram salvos.
   - Executar o seguinte comando para compilar o código usando o Makefile:

     ```
     make
     ```

   - O Makefile automaticamente compila o código e exibe uma mensagem indicando sucesso na compilação, além de instruções para execução.
4. Executar o Analisador Léxico
   - Para executar o analisador léxico, você precisará de um arquivo de texto (.txt) contendo o código em PL/0 que deseja analisar. Este arquivo deve estar localizado no mesmo diretório do `main.c`.
   - Suponha que o arquivo de entrada se chame `input.txt` e você deseja que a saída seja gravada em `output.txt`. Estes nomes podem ser alterados conforme necessário.
   - Execute o analisador léxico com o seguinte comando:

     ```
     make run INPUT=input.txt OUTPUT=output.txt
     ```

   - O comando acima utiliza as variáveis `INPUT` e `OUTPUT` do Makefile para especificar os arquivos de entrada e saída.

4.3 Conclusão

Ao seguir estes passos, o arquivo `output.txt` conterá os resultados da análise léxica do arquivo `input.txt`, detalhando os tokens identificados e os erros léxicos encontrados.

5 Exemplo de Execução

Nesta seção, apresenta-se um exemplo prático da execução do analisador léxico para a linguagem PL/0. O exemplo demonstra a análise de um arquivo de entrada contendo código PL/0, evidenciando a capacidade do analisador em identificar e categorizar tokens conforme sua estrutura sintática e semântica. São detalhados o arquivo de entrada e a saída gerada, mostrando como o analisador processa declarações, procedimentos, estruturas de controle e identifica erros léxicos.

5.1 Arquivo de Entrada

O arquivo de entrada contém o código fonte na linguagem PL/0 que será analisado pelo analisador léxico. Apesar de estar majoritariamente correto, foram insiridos tokens inválidos propositalmente para indicar o tratamento de erros léxicos (":"e "@"). Abaixo está o conteúdo do arquivo de entrada utilizado no exemplo:

```
CONST pi = 3; { definição de constante }
VAR x, y, z; { declaração de variáveis }

{ Definição de procedimento com um bloco interno }
PROCEDURE calcArea;
VAR radius;
BEGIN
    :radius := 2;
    x := pi * radius * radius; { cálculo da área do círculo }
END;

BEGIN
    x := 10;
    y := 20;
    z := x + y;
    CALL calcArea; { chamada de procedimento }
    IF x > y THEN { condicional }
        z :=@ x - y;
    WHILE z <> 0 DO { loop }
        z := z - 1;

    { Teste de operadores unários e relacionais }
    IF ODD x THEN
        z := -z;
    IF x = y THEN
        z := z / 2;
    IF x <= y THEN
        z := z * 2;
    IF x >= y THEN
        z := z + 100;
END.
```

Listing 1: input.txt

5.2 Arquivo de Saída

O arquivo de saída mostra os tokens identificados pelo analisador léxico após a análise do código fonte. Cada linha contém um token e seu tipo correspondente, incluindo os erros léxicos passados no arquivo de entrada. Abaixo está o conteúdo do arquivo de saída gerado pelo analisador:

```
CONST,CONST
pi,ident
=,simbolo_igual
3,numero
;,simbolo_ponto_virgula
{ definição de constante }, comentario
VAR,VAR
x,ident
y,ident
z,ident
;,simbolo_ponto_virgula
{ declaração de variáveis }, comentario
{ Definição de procedimento com um bloco interno }, comentario
PROCEDURE,PROCEDURE
calcArea,ident
;,simbolo_ponto_virgula
VAR,VAR
radius,ident
;,simbolo_ponto_virgula
BEGIN,BEGIN
:,<ERRO_LEXICO>
radius,ident
:=,simbolo_atribuicao
2,numero
;,simbolo_ponto_virgula
x,ident
:=,simbolo_atribuicao
pi,ident
*,simbolo_multiplicacao
radius,ident
*,simbolo_multiplicacao
radius,ident
;,simbolo_ponto_virgula
{ cálculo da área do círculo }, comentario
END,END
;,simbolo_ponto_virgula
BEGIN,BEGIN
x,ident
:=,simbolo_atribuicao
10,numero
;,simbolo_ponto_virgula
y,ident
:=,simbolo_atribuicao
20,numero
;,simbolo_ponto_virgula
z,ident
:=,simbolo_atribuicao
```

```
x,ident
+,simbolo_mais
y,ident
;,simbolo_ponto_virgula
CALL,CALL
calcArea,ident
;,simbolo_ponto_virgula
{ chamada de procedimento }, comentario
IF,IF
x,ident
>,simbolo_maior
y,ident
THEN,THEN
{ condicional }, comentario
z,ident
:=,simbolo_atribuicao
@,<ERRO_LEXICO>
x,ident
-,simbolo_menos
y,ident
;,simbolo_ponto_virgula
WHILE,WHILE
z,ident
<>,simbolo_diferente
0,numero
DO,DO
{ loop }, comentario
z,ident
:=,simbolo_atribuicao
z,ident
-,simbolo_menos
1,numero
;,simbolo_ponto_virgula
{ Teste de operadores unários e relacionais }, comentario
IF,IF
ODD,ODD
x,ident
THEN,THEN
z,ident
:=,simbolo_atribuicao
-,simbolo_menos
z,ident
;,simbolo_ponto_virgula
IF,IF
x,ident
=,simbolo_igual
y,ident
THEN,THEN
z,ident
:=,simbolo_atribuicao
z,ident
/,simbolo_divisao
2,numero
;,simbolo_ponto_virgula
IF,IF
```

```
x,ident
<=,simbolo_menor_igual
y,ident
THEN,THEN
z,ident
:=,simbolo_atribuicao
z,ident
*,simbolo_multiplicacao
2,numero
;,simbolo_ponto_virgula
IF,IF
x,ident
>=,simbolo_maior_igual
y,ident
THEN,THEN
z,ident
:=,simbolo_atribuicao
z,ident
+,simbolo_mais
100,numero
;,simbolo_ponto_virgula
END,END
.,simbolo_ponto
```

Listing 2: output.txt

6 Conclusão

O trabalho realizado pelo grupo na construção dos autômatos para o analisador léxico da linguagem PL/0 alcançou sucesso tanto na visualização com o uso do JFlap quanto na implementação prática com o código em C. A habilidade de descrever visualmente os autômatos facilitou a compreensão e o design da lógica de análise léxica, enquanto o código em C permitiu a efetiva execução da análise léxica sobre o código fonte escrito em PL/0.

Ao longo do desenvolvimento e testes do analisador léxico, o grupo demonstrou um robusto conhecimento dos conceitos envolvidos na análise léxica e na construção de compiladores sugeridos pela matéria. O projeto não só reforçou o conhecimento teórico adquirido em sala de aula, mas também proporcionou uma experiência prática fundamental no desenvolvimento de componentes essenciais para a compilação de linguagens de programação.

Além disso, o trabalho serviu como uma base sólida para futuras expansões no desenvolvimento do compilador, particularmente para a próxima fase do processo de compilação: a análise sintática, capaz de utilizar os tokens gerados para construir a árvore sintática do programa. Assim, este projeto não apenas cumpriu seus objetivos imediatos mas também estabeleceu uma sólida fundação para o avanço contínuo no estudo e implementação de compiladores.