Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Engenharia de Computação - 2024.1

Trabalho 1: analisador léxico para a linguagem PL/0

SCC0605 - Teoria da Computação e Compiladores

Mateus Santos Messias - NºUSP: 12548000

Pedro Borges Gudin - NºUSP: 12547997

Resumo

Este trabalho descreve o desenvolvimento de um analisador léxico para a linguagem PL/0, como parte da disciplina SCC0605 da Universidade de São Paulo. A gramática da linguagem foi analisada para identificar tokens, e autômatos de estados finitos foram projetados e implementados em C para reconhecê-los. Várias decisões de projeto foram tomadas, incluindo a escolha das estruturas de dados (tabela hash), o tratamento de erros léxicos e a organização do código. O programa resultante lê um arquivo de texto contendo um programa em PL/0 e gera um arquivo de saída com os tokens identificados e erros léxicos. O relatório cobre desde a definição dos tokens até a implementação e execução do programa, destacando as decisões de projeto e os resultados obtidos.

Introdução

Este trabalho apresenta o desenvolvimento de um analisador léxico para a linguagem de programação PL/0, conforme especificado na disciplina SCC0605 – Teoria da Computação e Compiladores da Universidade de São Paulo. O analisador léxico é responsável por converter a entrada de um programa em uma sequência de tokens, as menores unidades significativas da linguagem.

Para a criação do analisador léxico, a gramática da linguagem PL/0 foi analisada para identificar os diferentes tipos de tokens. Em seguida, foram projetados autômatos de estados finitos para reconhecer cada tipo de token, que foram implementados em linguagem C, utilizando técnicas de retrocesso e "lookahead".

Durante o desenvolvimento, várias decisões de projeto foram tomadas, como a escolha das estruturas de dados para representar os autômatos, a forma de tratamento dos erros léxicos e a organização do código para facilitar a manutenção e expansão futura. O resultado final é um programa que aceita um arquivo de texto com um programa escrito em PL/0 e gera um arquivo de saída com os tokens identificados, indicando eventuais erros léxicos. Este relatório detalha cada etapa do desenvolvimento, desde a definição dos tokens até a implementação dos autômatos e a execução do programa.

O Analisador Léxico

1. Gramática da linguagem PL/0

```
<programa> ::= <bloco> .
<bloco> ::= <declaracao> <comando>
<declaracao> ::= <constante> <variavel> <procedimento>
<constante> ::= CONST ident = numero <mais_const> ; | λ
<mais_const> ::= , ident = numero <mais_const> | λ
<variavel> ::= VAR ident <mais_var> ; | λ
<mais_var> ::= , ident <mais_var> | λ
<procedimento> ::= PROCEDURE ident ; <bloco> ; <procedimento> | λ
<comando> ::= ident := <expressao>
| CALL ident
| BEGIN <comando> <mais_cmd> END
| IF <condicao> THEN <comando>
| WHILE <condicao> DO <comando>
| λ
<mais_cmd> ::= ; <comando> <mais_cmd> | λ
<expressao> ::= <operador_unario> <termo> <mais_termos>
<operador_unario> ::= - | + | λ
<termo> ::= <fator> <mais_fatores>
<mais_termos> ::= - <termo> <mais_termos> | + <termo> <mais_termos> | λ
<fator> ::= ident | numero | ( <expressão> )
<mais_fatores> ::= * <fator> <mais_fatores> | / <fator> <mais_fatores>
| λ
<condicao> ::= ODD <expressao>
| <expressao> <relacional> <expressao>
<relacional> ::= = | <> | < | <= | > | >=
```

Além disso:

- comentários são de única linha, entre chaves { }
- identificadores são formados por letras e dígitos, começando por letra
- só há números inteiros, formados por um ou mais dígitos (entre 0 e 9)

2. Definição de Tokens

Baseado-se na gramática, os tokens podem incluir:

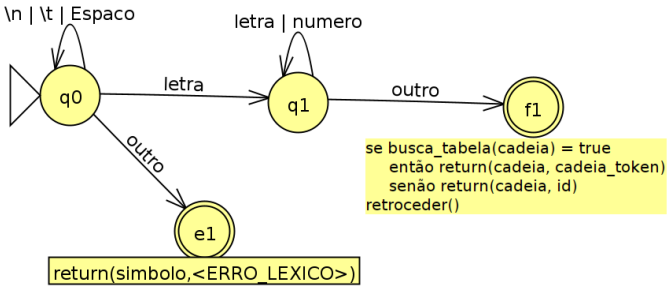
- **Palavras-reservadas**: `CONST`, `VAR`, `PROCEDURE`, `CALL`, `BEGIN`, `END`, `IF`, `THEN`, `WHILE`, `DO`, `ODD`
- **Identificadores**: Sequência de letras e dígitos, começando por letra.
- **Números inteiros**: Sequência de um ou mais dígitos.
- **Operadores**: `+`, `-`, `*`, `/`, `=`, `<>`, `<`, `<=`, `>`, `>=`
- **Símbolos especiais**: `:=`, `;`
- **Delimitadores**: `(`, `)`, `,`, `.`
- **Comentários**: `{` até `}` (ignorar o conteúdo)

2.1. **Autômato para identificadores ou palavras-reservadas**

Os identificadores e palavras-reservadas são definidos como sequências de letras e dígitos, começando por uma letra. O autômato a seguir reconhece essas sequências:

- Estado inicial: q0
- Estado final: f1 ou e1 (identificador/palavra-reservada ou erro léxico)

Diagrama do autômato:

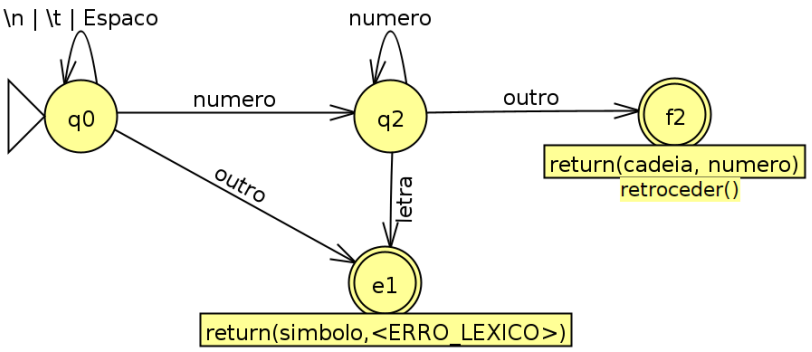


2.2. **Autômato para os números inteiros**

Os números inteiros são definidos como sequências de um ou mais dígitos. O autômato a seguir reconhece essas sequências:

- Estado inicial: q0
- Estado final: f2 ou e1 (número inteiro ou erro léxico)
- Transições:
    - q0 -> q2 com número
    - q0 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))
    - q2 -> q2 com número
    - q2 -> f2 com outro (return(cadeia, número), retroceder())
    - q2 -> e1 com numero (return(símbolo, <ERRO_LEXICO>))

Diagrama do autômato:

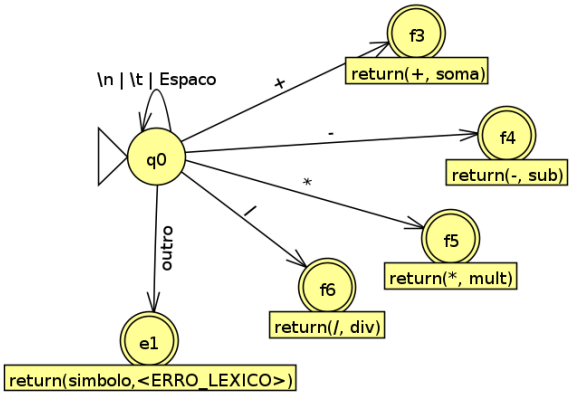


2.3. **Autômato para os operadores aritméticos**

Os operadores aritméticos incluem +, -, *, e /. O autômato a seguir reconhece esses operadores:

- Estado inicial: q0
- Estado final: q3 (operador aritmético)
- Transições:
    - q0 -> f3 com '+'
    - q0 -> f4 com '-'
    - q0 -> f5 com '*'
    - q0 -> f6 com '/'
    - q0 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))

Diagrama do autômato:

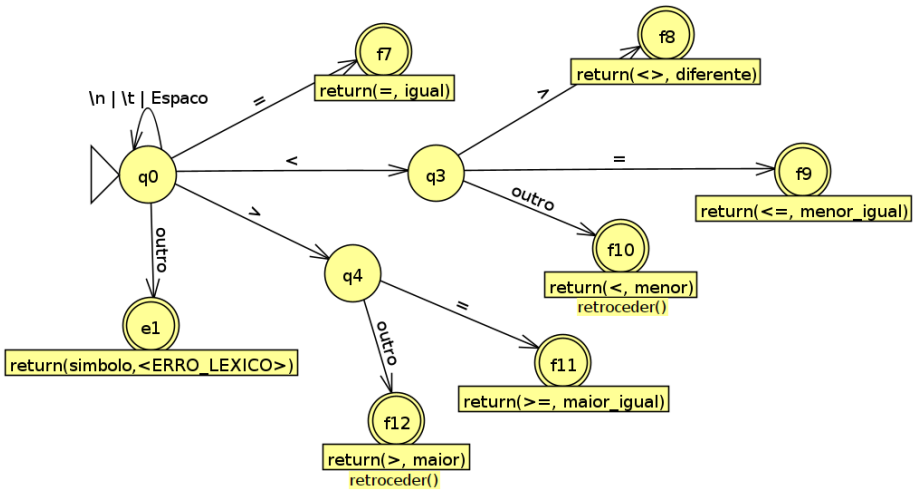


2.4. **Autômato para os operadores relacionais**

Os operadores relacionais incluem =, <>, <, <=, >, e >=. O autômato a seguir reconhece esses operadores:

- Estado inicial: q0
- Estado final: q4 (operador relacional)
- Transições:
    - q0 -> q3 com '='
    - q3 -> f7 com '=' (return('=', igual))
    - q0 -> q3 com '<'
    - q3 -> f8 com '>' (return('<>', diferente))
    - q0 -> q3 com '>'
    - q3 -> f9 com '=' (return('<=', menor_igual))
    - q0 -> q4 com '<'
    - q4 -> f10 com '=' (return('<', menor))
    - q0 -> q4 com '>'
    - q4 -> f11 com '=' (return('>=', maior_igual))
    - q4 -> f12 com outro (return('>', maior), retroceder())
    - q3 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))
    - q0 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))

Diagrama do autômato:



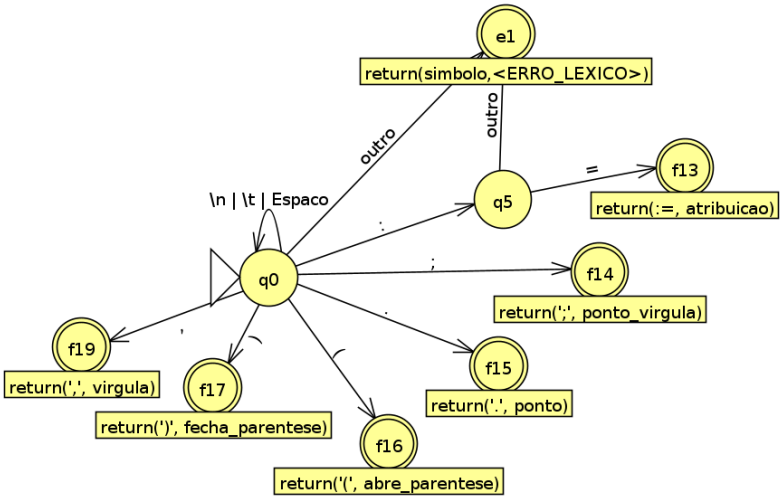
2.5. **Autômato para os símbolos especiais e delimitadores**

Os símbolos especiais e delimitadores incluem :=, ;, (,), ,, e . O autômato a seguir reconhece esses símbolos:

- Estado inicial: q0
- Estado final: q5 (símbolo especial ou delimitador)
- Transições:
    - q0 -> q5 com ':'

- q5 -> f13 com '=' (return(':=', atribuicao))
- q0 -> f14 com ';' (return(';', ponto_virgula))
- q0 -> f15 com '.' (return('.', ponto))
- q0 -> f16 com '(' (return('(', abre_parentese))
- q0 -> f17 com ')' (return(')', fecha_parentese))
- q0 -> f19 com ',' (return(',', virgula))
- q0 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))

Diagrama do autômato:

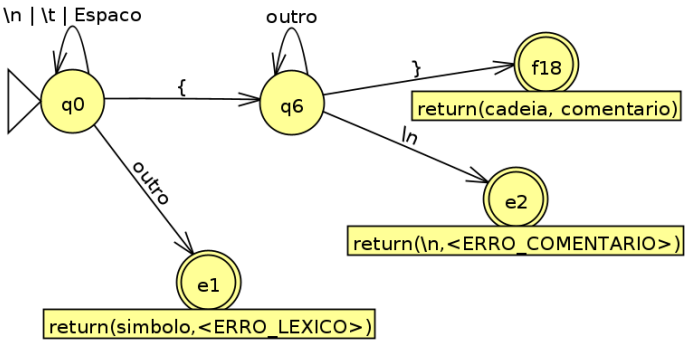


2.6. **Autômato para os comentários**

Os comentários são definidos entre chaves { }. O autômato a seguir reconhece comentários:

- Estado inicial: q0
- Estado final: q6 (comentário)
- Transições:
  - q0 -> q6 com '{'
  - q6 -> f18 com '}' (return('', COMENTARIO))
  - q6 -> e2 com '\n' (return('\n', <ERRO_COMENTARIO>))
  - q6 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))
  - q0 -> e1 com outro (return(símbolo, <ERRO_LEXICO>))

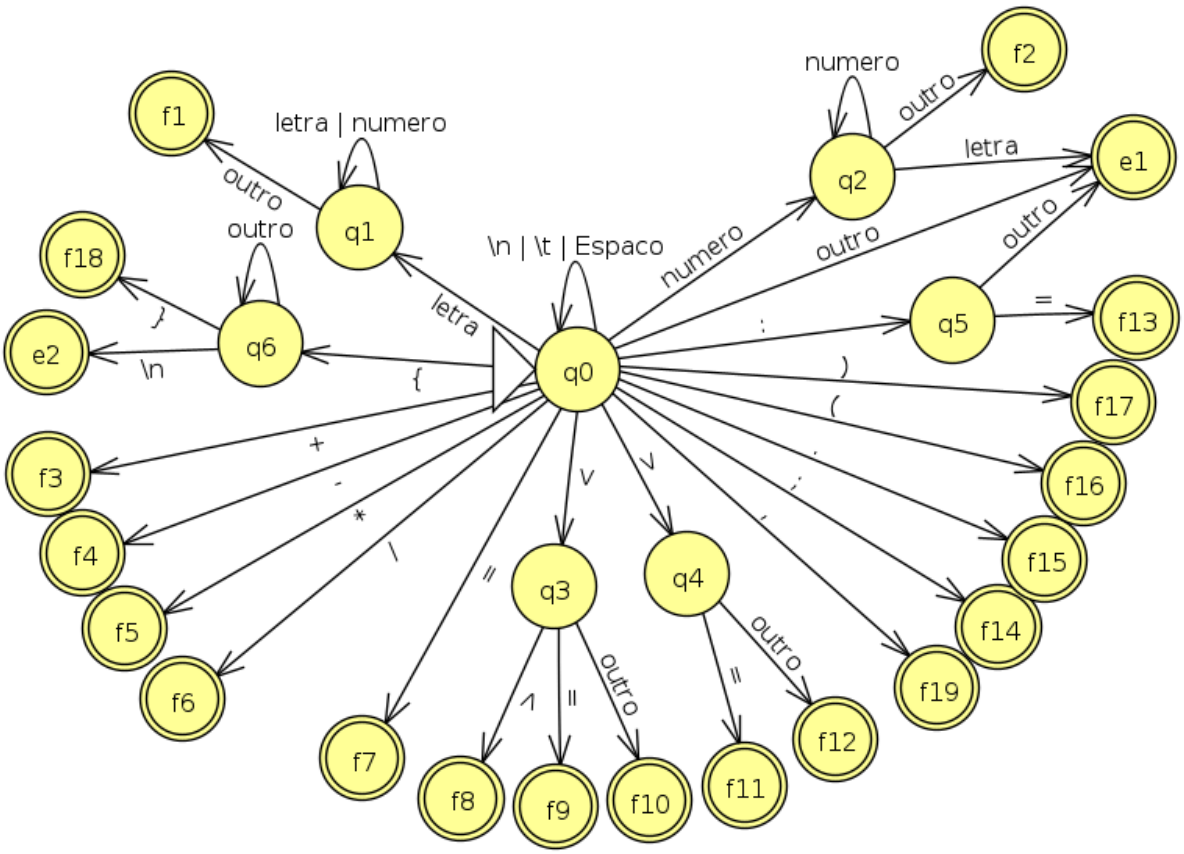
Diagrama do autômato:



2.7. **Autômato final**

O autômato final combina os autômatos individuais para identificar todos os tokens da linguagem PL/0.

Diagrama do autômato final:



3. **Tabela de estados do autômato**

Estado	Letra	Número	+	-	*	/	<	>	=	(	)	;	:	,	.	{	}	\n	\t	Espaço	Outro
q0	q1	q2	f3	f4	f5	f6	q3	q4	f7	f16	f17	f14	q5	f19	f15	q6	e1	q0	q0	q0	e1
q1	q1	q1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1	f1
q2	e1	q2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2	f2
q3	f10	f10	f10	f10	f10	f10	f10	f8	f9	f10	f10	f10	f10	f10	f10	f10	f10	f10	f10	f10	f10
q4	f12	f12	f12	f12	f12	f12	f12	f12	f11	f12	f12	f12	f12	f12	f12	f12	f12	f12	f12	f12	f12
q5	e1	e1	e1	e1	e1	e1	e1	e1	f13	e1	e1	e1	e1	e1	e1	e1	e1	e1	e1	e1	e1
q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	q6	f18	e2	q6	q6	q6

Decisões de Projeto

Durante o desenvolvimento do analisador léxico, tomamos várias decisões de projeto para garantir a eficiência e a facilidade de manutenção do código:

1. **Estruturas de Dados**: Optamos pelo uso de tabelas hash para mapear todo autômato, essa tabela foi feita com base na tabela de estados do autômato.
2. **Modularidade do Código**: Organizamos o código em funções separadas, facilitando a adição de novos estados no autômato com poucas mudanças no código.
3. **Função lookahead**: Não foi implementada uma função lookahead, pois toda vez que o autômato lê um caractere “outro” e não é um estado de erro e nem o estado q6(estado que consome os comentários) ele não coloca no buffer e retrocede o caractere lido para ser lido em uma próxima chamada da função léxico.
4. **Retorna token de comentário**: O autômato retorna um token de comentário
5. **Automatização da Execução**: Utilizamos o utilitário `make` para automatizar a compilação e execução do programa.

Enfrentamos desafios, como garantir a correta implementação dos autômatos finitos e o tratamento eficiente de comentários e espaços em branco. Para superar esses desafios, realizamos testes extensivos com diversos programas escritos em PL/0 e ajustamos nossos autômatos conforme necessário.

Executando e Compilando o programa

1. **Requisitos de sistema**

- Sistema Operacional: Linux Ubuntu 22.04
- Compilador: GCC
- Ferramentas adicionais: make

2. **Como executar o programa**

Para compilar e executar o programa, siga os seguintes passos:

1. Navegue até o diretório onde o código-fonte está localizado.
2. Para compilar o programa digite `make` ou `make all` no terminal
3. Para rodar basta digitar `make run ARGS=<nome_do_programa_de_entrada.txt>`, ja existe um programa teste na pasta que utilizaremos, portanto o comando para rodar fica:
   `make run ARGS=teste.txt`.
4. O arquivo de entrada deve estar no formato de um programa PL/0.

2.1 **Saída do programa**

Ao executar o programa com um arquivo de entrada será gerado um arquivo de saída chamado `saida.txt` que contém os tokens gerados pelo analisador léxico.

Exemplo de Entrada:

```
CONST ond = 00x1F;
VAR var_1,
1{ asdfkj asld 23 */d{ {[]];/?!@#$%*(")@}

1{ asdfkj asld 23 */d{ {[]];/?!@#$%*(")@
END.
{asdfjalsdfj
```

Exemplo de saída:

```
CONST, CONST
ond, id
=, igual
00x, <ERRO_LEXICO>
1F, <ERRO_LEXICO>
;, ponto_virgula
VAR, VAR
var, id
_, <ERRO_LEXICO>
1, numero
,, virgula
1, numero
{ asdfkj asld 23 */d{ {[]];/?!@#$%*(")@}, comentario
```

```
1, numero
\n, <ERRO_COMENTARIO>
END, END
., ponto
{asdfjalsdfj , <ERRO_LEXICO>
```