UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Teoria da Computação e Compiladores - SCC0605

COMPILADOR DE PL-0

Artur Brenner Weber - 12675451
Gabriel Franceschi Libardi - 11760739
Guilherme Castanon Silva Pereira - 11801140
Gustavo Moura Scarenci de Carvalho Ferreira - 12547792
Matheus Henrique Dias Cirillo - 12547750

Docente responsável: Prof. Tiago A. S. Pardo

São Carlos
1º semestre/2024

SUMÁRIO

1 Introdução

No contexto da compilação de código, um analisador léxico lê o código-fonte de um programa como uma sequência de caracteres e os separa em tokens da linguagem. Exemplos destes são os identificadores, palavras-chave, literais e dígitos. Sendo a análise léxica a primeira etapa do processo de compilação, esse organiza as construções do código-fonte em átomos léxicos para que as próximas fases do compilador, principalmente a análise sintática, analisem o código e seja feita a síntese do código de máquina. Isto pode ser melhor exemplificado na imagem abaixo, de [1]:

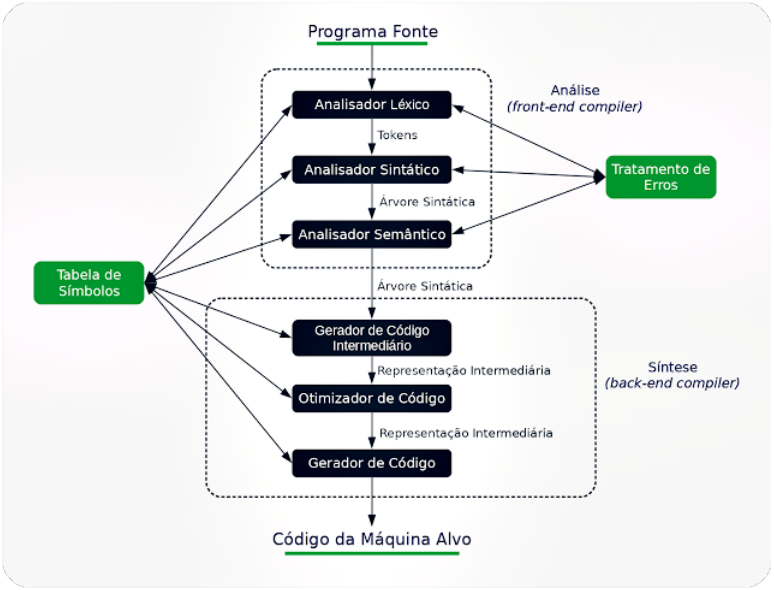

Figura 1: *Etapas de um compilador, exemplificando sua a dependência no analisador léxico [1]*

Uma linguagem de grande influência no estudo do funcionamento de compiladores é a PL/0, apresentada inicialmente por Niklaus Wirth no livro [2], que vem sido muito utilizada para o fim didático desde então. Nesse sentido, o presente projeto consiste na implementação de um analisador léxico para a linguagem PL/0, a qual, apesar de semelhante ao Pascal, é simplificada para diminuir a dificuldade de implementação do compilador. Esta possui apenas um tipo de dado (números inteiros) e as estruturas de controle básicas de linguagens de programação procedurais (WHILE, FOR, IF, PROCEDURE e CALL), além de não fornecer nenhum tipo de paradigma mais avançado para desenvolvimento (como orientação a objetos). PL/0 é apenas uma linguagem de paradigma procedural (como Pascal) com o mínimo de *features*.

Para desenvolvimento do analisador, foi utilizada a gramática da linguagem com suas respectivas regras de geração, obtidas pelo livro [2]. A partir da documentação, criou-se um autômato de estados finitos de Moore, na qual cada estado está associado a uma saída. Em seguida, esse autômato foi implementado na linguagem C. Sua função principal é receber um arquivo *.txt* com código em PL/O e devolver os tokens identificados com suas respectivas classes, bem como mensagens de erros *user-friendly* para os erros léxicos identificados. É importante ressaltar a grande ênfase do projeto em fornecer ao (teórico) usuário a maior quantidade de detalhamento de erro e intuitividade de uso possível.

2 Desenvolvimento do Analisador Léxico

2.1. Autômato de Estados Finitos

O formalismo operacional produzido é um **Autômato de Moore com "retrocessos"** [3], representado na Figura 2.

Na notação da Figura 2, a movimentação da cabeça de leitura (que lê a "fita" de entrada do autômato) é definida junto às transições da máquina. As transições denotadas com `(F)` movimentam a cabeça de leitura para frente (da mesma forma que AFDs normais) e as denotadas com `(B)` mantém a cabeça de leitura fixa mesmo depois que o carácter foi lido. Esse último tipo de transição é uma transição com "retrocesso".

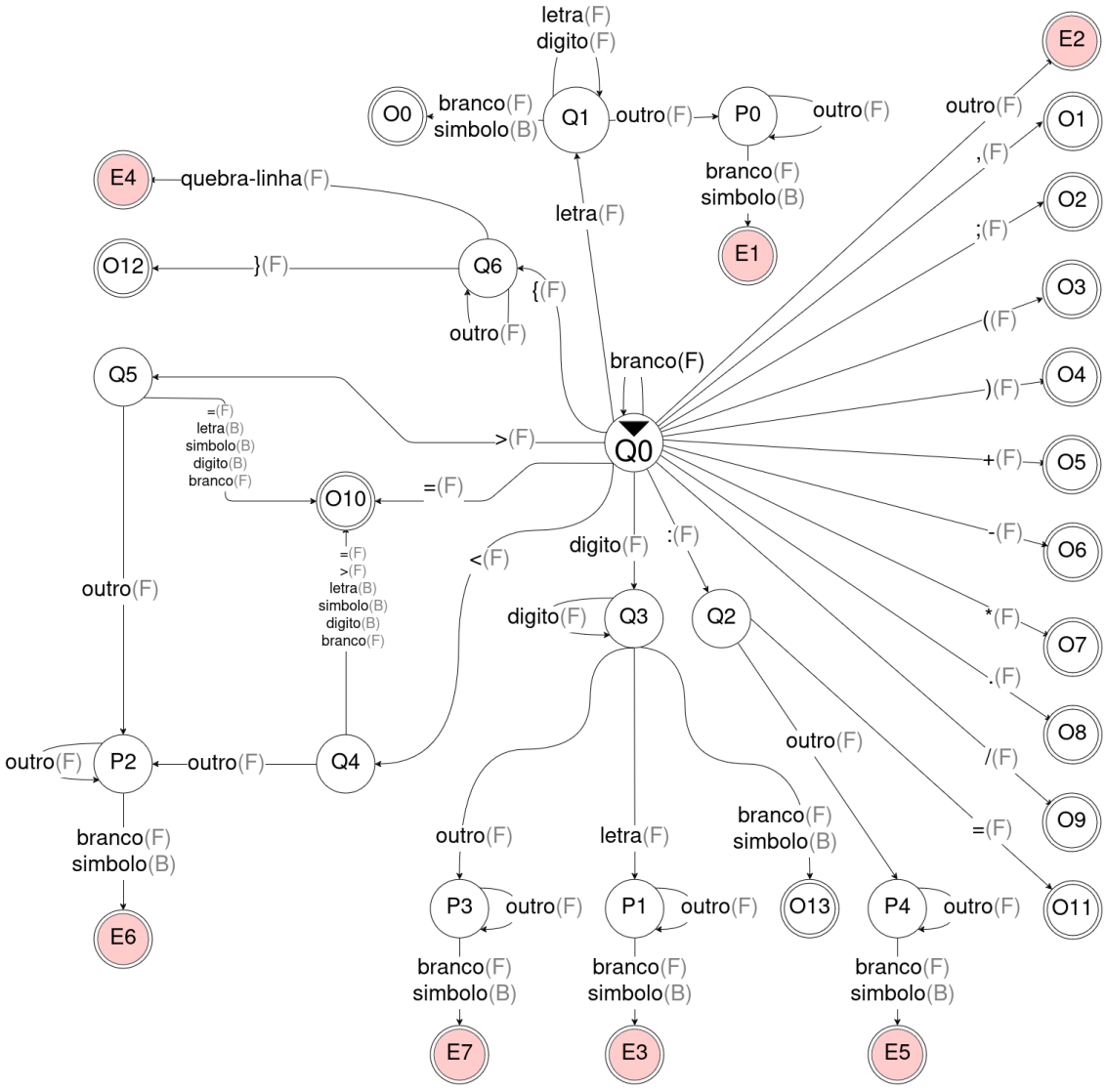

Figura 2: *Representação Gráfica do Autômato correspondente ao analisador léxico de PL/0.*

As entradas `digito`, `simbolo`, `letra`, `branco`, `quebra-linha` são conjuntos de caracteres e definidos na Tabela 1.

Conjunto	Elementos
`dígito`	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
`símbolo`	(|) | * | + | , | - | . | / | : | ; | <| = | >| ?
`letra`	A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
`branco`	'\t' | '\n' | '\v' | '\f' | '\r' | ' '
`quebra-linha`	'\n'

Tabela 1: *Definições de Conjuntos de Caracteres*

Já a entrada `outro` é equivalente à "caso contrário". Se o autômato lê um carácter que não corresponde a nenhuma transição, o autômato faz a transição de rótulo `outro` (quando essa transição existir). A implementação feita para o autômato recebe três arquivos *.csv* que descrevem o autômato: uma tabela de palavras-chave, uma tabela de estados do autômato e suas saídas, e uma tabela de transições de estado. A implementação é flexível o suficiente para interpretar qualquer autômato de Moore descrito na sintaxe desses arquivos *.csv*, e pode ser usada para criar analisadores léxicos de outras linguagens.

Em alguns casos, a representação desse automato é não determinística, entretanto na implementação em C o autômato é determinístico. Transições não determinísticas são transformadas em determinísticas pela ordem da definição delas no arquivo CSV (equivalente à Tabela 3). Portanto, a transição que aparece primeiro tem preferência sobre outra que aparece posteriormente. Isso resolve, por exemplo, o problema que surge na transição entre `Q4` e `O10`, pois acontece para = e `simbolo` (que já inclui =). Este comportamento é necessário, pois ao encontrar = a cabeça de leitura deve mover para frente e, ao encontrar qualquer outro `simbolo`, não deve se mover. Dessa forma, o não-determinismo da Figura 2 é apenas um abuso de notação para as informações caberem na figura, pois o autômato é determinístico.

Os estados dividem-se em 4 diferentes classes: estado inicial, estados intermediários, estados finais de erros e estados finais regulares. Isso permite controlar melhor a análise léxica. Cada estado e sua respectiva saída está mostrado na Tabela 2.

Tabela 2: *Tabela de estados e saídas do autômato*

Nome	Tipo	Saída
Q0	Inicial	ϵ
Q1	Intermediário	ϵ
Q2	Intermediário	ϵ
Q3	Intermediário	ϵ
Q4	Intermediário	ϵ
Q5	Intermediário	ϵ
Q6	Intermediário	ϵ
O0	Final Regular	identifier
O1	Final Regular	comma

Continua na próxima página

Tabela 2: *Tabela de estados e saídas do autômato (continuação)*

Nome	Tipo	Saída
O2	Final Regular	semicolon
O3	Final Regular	left_par
O4	Final Regular	right_par
O5	Final Regular	simb_plus
O6	Final Regular	simb_minus
O7	Final Regular	simb_mult
O8	Final Regular	period
O9	Final Regular	simb_div
O10	Final Regular	rel_op
O11	Final Regular	assign_op
O12	Final Regular	comment
O13	Final Regular	integer_literal
P0	Intermediário	ϵ
P1	Intermediário	ϵ
P2	Intermediário	ϵ
P3	Intermediário	ϵ
P4	Intermediário	ϵ
E1	Final de Erro	*Found invalid character on candidate identifier. Use only letters and digits for identifiers.*
E2	Final de Erro	*Invalid character*
E3	Final de Erro	*Ident beginning with digits or invalid integer literal. Did you mean to write a number?*
E4	Final de Erro	*Unexpected line break. Comments must be inline. Did you mean to close your comment with '}' before skipping to the next line?*
E5	Final de Erro	*Colons are exclusively used for the assignment operator :=. Did you mean to write := ?*
E6	Final de Erro	*Angular brackets are used for relational operators. Did you mean to write $<$, $>$, $<=$ or $>=$?*
E7	Final de Erro	*Found spurious symbol on integer literal. Did you mean to write a number?*

2.2. Implementação do Autômato de Estados Finitos em C

2.2.1 Instruções de execução

Para rodar o código, basta digitar no terminal:

```
$ make run ARGS="<source_file> <output_file>"
```

em que <source_file> é o caminho para o código fonte (que é um arquivo de texto) e <output_file> é a saída pedida com a tabela de tokens e suas respectivas classes. A execução do programa sobrescreve todo o conteúdo de <output_file> com a saída do analisado léxico. Para uma visão geral do funcionamento e implementação do analisador léxico, sugere-se testá-lo com o programa no arquivo `res/program.pl0`, executando o comando:

```
$ make run ARGS="res/program.pl0 saida.txt"
```

2.2.2 Organização do Projeto

Para organização física dos arquivos do projeto, optou-se pela seguinte estrutura de diretórios:

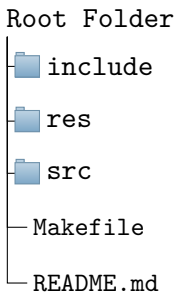

Figura 3: *Organização do diretório de implementação*

Após reflexão da equipe considerando, modularização, tamanho de código e o próximo projeto da disciplina, a estrutura do diretório é formada por três principais pastas, além do *Makefile* e *README*. A primeira, *include*, contém os arquivos header *.h* dos códigos C: `IO_utils`, `lexer`, `state_machine` e `str_utils`, além das estruturas de dados utilizadas. Nela, está inclusa a documentação detalhada de todas as principais funções utilizadas no código.

A pasta *src* contém todos os arquivos fonte do programa em C, enquanto *res* armazena os arquivos `.csv` com as transições, o mapa de estados do programa, o exemplo de programa PL/0 e as *keywords* da linguagem, os quais serão melhor explicados em seguida.

2.2.3 Ambiente de teste e desenvolvimento

Para desenvolvimento do código, utilizou-se o programa *Visual Studio Code* versão 1.89.1 nos sistemas operacionais Windows 11 (utilizando WSL) e Fedora 39. Para teste em cada SO, utilizaram-se os compiladores GCC 11.4.0 e GCC 13.2.1, respectivamente.

2.2.4 O Código

O analisador léxico funciona a partir de uma estrutura chamada (`TokStream`), responsável por armazenar o código-fonte, a lista de `keywords` da linguagem e a máquina de estados (`StateMachine`).

Nada da linguagem é diretamente codificado, todos os estados (Tabela 2), transições (Tabela 3) e *keywords* (Tabela 4) são definidos em arquivos CSV e lidos pelo programa para construir as estruturas. Essa é a lógica pivô e diferencial do código, a qual permite alterar a linguagem sem precisar alterar o código-fonte do programa, dando liberdade e facilitando a solução de erros.

A função central do programa é a `get_next_token()`, que lê caracteres do arquivo-fonte e constrói os tokens. Ela inicia alocando memória para um novo token e lê caracteres um a um, determinando a transição apropriada entre estados. Quando um estado final é alcançado, a função determina o tipo do token e retorna a estrutura `Token` que contem a string e o tipo. Um exemplo do uso de `get_next_token()` pode ser visto no código 1.

Além disso, o programa possui funções para gerenciar a memória dinâmica, garantir a leitura e escrita seguras de arquivos, e lidar com colisões em tabelas de *hash* usadas para armazenar estados. No final, todos os recursos alocados são liberados adequadamente, garantindo uma execução eficiente e sem vazamentos de memória. O código por completo está no GitHub [1].

```c
#include <stdio.h>
#include <stdlib.h>

#include "lexer.h"
#include "IO_utils.h"

int main(int argc, char *argv[]){
    TokStream *b = token_stream_init(argv[1]);
    Token *t = NULL;

    while (t = get_next_token(b)) {
        printf(out_file, "%s , %s\n", t->token_str, t->type);
    }

    token_stream_free(&b);
    return 0;
}
```

Código 1: *Exemplo de uso do `get_next_token()`*

2.2.5 Formato dos Arquivos .csv

Como citado anteriormente, três arquivos `.csv` foram usados para descrever o autômato de estados, e esses são carregados no analisador léxico em tempo de execução. O primeiro é uma tabela que lista as palavras-chave e suas respectivas classes léxicas, seguindo o formato "Keyword|Type". O segundo é uma tabela que enumera os estados, os seus respectivos tipos e suas saídas, e deve seguir o formato "Name|Type|Output": "Name" é o nome do estado (qualquer *string* de caracteres imprimíveis sem quebra de linha), "Type" é um dos quatro tipos de estado (como na Tabela 2) e "Output" é a saída. Por fim, o terceiro arquivo é uma tabela de transições que deve seguir o formato "CurrentState|Input|NextState|Direction": "CurrentState" é o estado de partida da transição, "Input" são as entradas que produzem a transição (essas podem ser simplesmente caracteres ou o nome dos grupos definidos na tabela 1), "NextState" é o estado de chegada da transição e "Direction" é a direção da transição - só pode ser `F`, uma transição regular, ou `B`, uma transição com retrocesso.

[1] https://github.com/GuScarenci/compilador-pl0

3 Conclusão

Para concluir o projeto de desenvolvimento do analisador léxico para a linguagem PL/0, a equipe refletiu sobre os objetivos e resultados alcançados ao longo do processo. O grupo implementou com sucesso um analisador léxico funcional, capaz de processar código-fonte escrito em PL/0 e segmentá-lo em tokens adequados para a análise sintática subsequente. Esta proposta envolveu a criação de um autômato de estados finitos, especificamente uma máquina de Moore com retrocessos, para gerenciar as transições e estados durante a leitura do código-fonte.

A implementação em C foi desafiadora e educacional, proporcionando uma experiência prática com técnicas fundamentais de compilação e design de autômatos. Utilizou-se a gramática formal da linguagem PL/0 para definir as regras de transição do autômato, garantindo que o analisador léxico pudesse reconhecer corretamente os elementos sintáticos da linguagem, como identificadores, literais e operadores.

Uma das características notáveis do projeto foi a modularidade e flexibilidade proporcionada pelo uso de arquivos CSV para definir as transições de estado e as palavras-chave da linguagem. Esta abordagem permitiu ajustes e modificações na linguagem analisada sem a necessidade de alterar o código-fonte, facilitando futuras extensões e adaptações.

Além disso, a estrutura organizada dos arquivos do projeto, com diretórios dedicados para cabeçalhos, fontes, recursos e documentação, contribuiu para uma manutenção eficiente e um desenvolvimento mais colaborativo. O uso de boas práticas de programação, como o gerenciamento adequado da memória e a utilização de tabelas de hash, proporciona a robustez e a eficiência do analisador léxico.

Em suma, o projeto alcançou seus objetivos didáticos, proporcionando um entendimento profundo das etapas iniciais do processo de compilação e das técnicas envolvidas na construção de analisadores léxicos. O produto final é uma ferramenta útil, que pode servir como base para estudos adicionais e projetos mais complexos no campo da compilação e análise de linguagens de programação.

Referências Bibliográficas

[1] J. D. Marangon, “Compiladores para humanos,” 2017. Recuperado de https://johnidm.gitbooks.io/compiladores-para-humanos/content/.

[2] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice Hall, 1976. Includes bibliography and index.

[3] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

Apêndice

Tabela 3: *Tabela de transições do autômato*

Estado Atual	Entrada	Próximo Estado	Direção
Q0	branco	Q0	F
Q0	,	O1	F
Q0	;	O2	F
Q0	(	O3	F
Q0	)	O4	F
Q0	+	O5	F
Q0	-	O6	F
Q0	*	O7	F
Q0	.	O8	F
Q0	/	O9	F
Q0	=	O10	F
Q0	letra	Q1	F
Q0	:	Q2	F
Q0	dígito	Q3	F
Q0	<	Q4	F
Q0	>	Q5	F
Q0	{	Q6	F
Q0	outro	E2	F
Q1	letra	Q1	F
Q1	dígito	Q1	F
Q1	branco	O0	B
Q1	símbolo	O0	B
Q1	outro	P0	F
P0	branco	E1	F
P0	símbolo	E1	B
P0	outro	P0	F
Q2	=	O11	F
Q2	outro	P4	F
P4	branco	E5	F
P4	símbolo	E5	B
P4	outro	P4	F
Q3	branco	O13	B
Q3	símbolo	O13	B
Q3	dígito	Q3	F
Q3	letra	P1	B
Q3	outro	P3	F
P1	branco	E3	F

Continua na próxima página

Tabela 3: *Tabela de transições do autômato (continuação)*

Estado Atual	Entrada	Próximo Estado	Direção
P1	símbolo	E3	B
P1	outro	P1	F
P3	branco	E7	F
P3	símbolo	E7	B
P3	outro	P3	F
Q4	=	O10	F
Q4	>	O10	F
Q4	símbolo	O10	B
Q4	letra	O10	B
Q4	dígito	O10	B
Q4	branco	O10	B
Q4	outro	P2	F
P2	branco	E6	F
P2	símbolo	E6	B
P2	outro	P2	F
Q5	=	O10	F
Q5	letra	O10	B
Q5	símbolo	O10	B
Q5	dígito	O10	B
Q5	branco	O10	B
Q5	outro	P2	B
Q6	outro	Q6	F
Q6	}	O12	F
Q6	quebra-linha	E4	B

Tabela 4: *Definições de keywords*

Keyword	Type
CONST	keyword_const
VAR	keyword_var
PROCEDURE	keyword_proc
CALL	keyword_call
BEGIN	keyword_begin
END	keyword_end
IF	keyword_if
THEN	keyword_then
WHILE	keyword_while
DO	keyword_do
ODD	keyword_odd