



Universidade de São Paulo (USP)
Instituto de Ciências Matemáticas e
de Computação (ICMC)



Trabalho 1 - Teoria da computação e compiladores

Beatriz Lomes da Silva

NUSP: 12548038

Hugo Hiroyuki Nakamura

NUSP: 12732037

Isaac Santos Soares

NUSP: 12751713

Nicholas E. P. de O. R. Bragança

NUSP: 12689616

São Carlos - São Paulo

17 de maio de 2024

1 Decisões de projeto

O presente trabalho consiste no projeto e implementação de um Analisador Léxico da linguagem de programação PL/0 na forma de um Autômato de Estados Finitos Determinístico.

A seguir estão descritas algumas das decisões de projeto que guiaram o desenvolvimento:

1.1 Projeto do Autômato de Estados Finitos Determinístico

- Optou-se por usar a ferramenta JFLAP [1], apesar das desvantagens que ela apresenta, sendo a principal delas não permitir a realização de testes com transições genéricas como "Dígito" ou "Outro".
- Ao invés de especificar transições para cada possível símbolo de entrada, optou-se por flexibilizar a notação, fazendo o uso de transições amplas como: "Não-dígito", "Dígito" e "Outro". Assim, foi possível obter um autômato mais legível. A Tabela 1 mapeia as variáveis flexibilizadas com os respectivos caracteres.

Tabela 1: Tabela de notações variável-carácter.

Variável	Caracteres associados
Dígito	_,a,...,z, A,...,Z
Não-Dígito	0,...,9

- Definiu-se uma função de saída para cada estado final, que retorna o par (*Token*, *Tipo-Token*) conforme solicitado pelo Analisador Sintático. Se necessário, as funções de saída podem consultar a Tabela de Palavras Reservadas e/ou retroceder a cadeia.
- O autômato desconsidera a verificação de um número com zeros à esquerda.
- O autômato não trata comentários que comecem em uma linha e termine em outra, apenas comentários *inline*.

1.2 Implementação em Linguagem C

- Optou-se por implementar o autômato com a tabela de transição definida explicitamente, para melhor modularização do código. Assim, as tabelas de caracteres específicos foram separados em arquivos: *Tabelas/palavras_reservadas.txt* e *Tabelas/trasicoes.txt*

Tabela 2: mapeamento de funções de saída para os estados de saída

Estado final	Procedimentos	Retorno
S0	verifica se palavra reservada e retrocede	(token, token) ou (token, identificador)
S1	-	Nulo
S2	-	(token, simbolo_virgula)
S3	-	(token, simbolo_ponto_virgula)
S4	-	(token, simbolo_abre_parenteses)
S5	-	(token, simbolo_fecha_parenteses)
S6	-	(token, simbolo_mais)
S7	-	(token, simbolo_menos)
S8	-	(token, simbolo_multiplicacao)
S9	-	(token, simbolo_ponto)
S10	-	(token, simbolo_divisao)
S11	-	(token, simbolo_igualdade)
S12	-	(token, simbolo_atribuicao)
S13	retrocede	(token, numero)
S14	-	(token, simbolo_menor_igual)
S15	-	(token, simbolo_diferente)
S16	retrocede	(token, simbolo_menor)
S17	-	(token, simbolo_maior_igual)
S18	retrocede	(token, simbolo_maior)
S19	-	Nulo
E1	retrocede	(token, <erro_lexico>)
E2	-	(token, <erro_lexico>)

No caso da função de saída do identificador ou de palavras reservadas, além de retroceder, ele também precisa realizar a verificação para diferenciar a saída do tipo-identificador e o tipo-palavra-reservada. Essa função de saída, referente ao estado S_0 , é mostrada no código 1.

```
char * funcaoSaida0(char * palavra){
    //Ignora o ultimo caractere lido: retrocede.
    int tam = strlen(palavra);
    palavra[tam-1] = '\0';

    char * saida = (char*) malloc(MAXLINHA);

    //Verifica se eh palavra reservada.
    int palavraReservada = verificaSePalavraReservada(palavra);

    //Se eh reservada, identifica com o proprio nome.
    if(palavraReservada){
        strcpy(saida, palavra);
        strcat(saida, ", ");
        strcat(saida, palavra);
    }
    //Se ela nao eh reservada, identifica como identificador.
    else{
        strcpy(saida, palavra);
        strcat(saida, ", ");
        strcat(saida, IDENT);
    }
}
```

```

    }
    return saida;
}

```

Código 1: função de saída de S_0

Há ramos do autômato que devem receber símbolos de apenas um caractere e, portanto, não precisam o *retroceder*. A sua função de saída apenas retorna a identificação do token. Esse tipo de função de saída também é usado para reconhecer erros léxicos. O código 2 mostra um exemplo de função de saída de S_4 , que lê '('.

```

char * funcaoSaida4(char * palavra){
    //Aloca memoria para a saida.
    char * saida = (char*) malloc(MAXLINHA);

    //Concatena o token 'palavra' com seu tipo.
    strcpy(saida, palavra);
    strcat(saida, ", ");
    strcat(saida, SIMB_ABRE_PARENTESE);

    //Retorna o par (token, tipo).
    return saida;
}

```

Código 2: função de saída de S_4

Para ler símbolos com mais de um caractere, que pode receber um outro caractere que não faz parte do símbolo, é preciso retrocedê-lo para a fita. Dessa forma, a função de saída precisa ignorar o último caractere para imprimir na saída. O código 3 mostra a função de saída do estado S_{13} , que os tokens dos dígitos.

```

char * funcaoSaida13(char * palavra){
    //Ignora o ultimo caractere lido: retrocede.
    int tam = strlen(palavra);
    palavra[tam-1] = '\0';

    //Aloca memoria para a saida.
    char * saida = (char*) malloc(MAXLINHA);

    //Concatena o token 'palavra' com seu tipo.
    strcpy(saida, palavra);
    strcat(saida, ", ");
    strcat(saida, SIMB_NUMERO);
    return saida;
}

```

Código 3: função de saída de S_{13}

3 Instruções de compilação

Antes de compilar, é preciso ter o código-fonte escrito em PL/0. Todos os códigos-fonte devem estar no diretório *Entradas*, na pasta principal do projeto. Nela, há um código padrão chamado *in.txt*, que será executado se nenhum outro for indicado. O código 4 apresenta o código-fonte padrão.

```

1  VAR a, b, c;
2  BEGIN
3      a:=2;
4      b:=3;
5      c:=a+b
6  END.

```

Código 4: arquivo *txt* escrito em *PL/0*.

Para executar o programa do analisador léxico, na pasta principal do projeto, deve-se inserir no terminal:

```
$ make IN="Entradas/seu_codigo_pl.txt"
```

onde *seu_codigo_pl.txt* é o arquivo do código-fonte escolhido. Caso não se coloque a cláusula *IN*, o arquivo padrão executado será "*Entradas/in.txt*".

O comando *make* limpará os arquivos antigos, compilará e executará. Após esse passo, o programa gerará duas saídas:

- *Logs/logs.txt*: arquivo que registra a saída do terminal, com mensagens de debug;
- *Logs/out.txt*: arquivo com apenas os pares *(token, tipo)* do arquivo *in.txt*.

O program foi desenvolvido e testado no sistema operacional *Ubuntu 22.04.3 LTS*. Para acesso ao repositório do GitHub, contendo o autômato, os códigos e mais informações, confira a referência [2].

4 Exemplo de execução

Para o teste do autômato o código-fonte 5, localizado em *Logs/teste_simbolos.txt*, foi utilizado.

```

1  {Bom dia, pr^'%&of. Th+-=7:84[]iago Pardo}
2  VAR n, sum, i, j;
3  ^~,;
4  PROCEDURE Sum^~'UpToN;
5  VAR temp;
6  BEGIN
7      sum := 0;
8      WHILE i := 1^~'%'26 DO
9          sum := sum + i;
10     END;
11 END;
12
13 PROCEDURE 'MultUpToN;
14 VAR temp;
15 BEG~IN
16     te^mp := 1;
17     WHILE i := 2 DO
18         temp := temp * i;
19     END;
20 END;
21
22 BEGIN

```

```

23     n:=10;
24     j:=0;
25     WHILE j <= n DO
26     BEGIN
27         CALL SumUpToN;
28         j:= j + 1;
29     END
30 END.

```

Código 5: arquivo de teste de todos os símbolos.

O código 6 apresenta o arquivo de saída gerado, em *Logs/out.txt*

```

1  VAR, VAR
2  n, identificador
3  ,, simbolo_virgula
4  sum, identificador
5  ,, simbolo_virgula
6  i, identificador
7  ,, simbolo_virgula
8  j, identificador
9  ;; simbolo_ponto_virgula
10 ^, <erro_lexico>
11 ~, <erro_lexico>
12 ', <erro_lexico>
13 ', <erro_lexico>
14 PROCEDURE, PROCEDURE
15 Sum, identificador
16 ^, <erro_lexico>
17 ~, <erro_lexico>
18 ', <erro_lexico>
19 ', <erro_lexico>
20 UpToN, identificador
21 ;; simbolo_ponto_virgula
22 VAR, VAR
23 tem, identificador
24 /, simbolo_divisao
25 p, identificador
26 ;; simbolo_ponto_virgula
27 sum, identificador
28 :=, simbolo_atribuicao
29 0, numero
30 ;; simbolo_ponto_virgula
31 WHILE, WHILE
32 i, identificador
33 :=, simbolo_atribuicao
34 1, numero
35 ^, <erro_lexico>
36 ~, <erro_lexico>
37 ', <erro_lexico>
38 ', <erro_lexico>
39 %, <erro_lexico>
40 26, numero
41 sum, identificador
42 :, <erro_lexico>
43 ~, <erro_lexico>
44 =, simbolo_igualdade
45 sum, identificador
46 +, simbolo_mais
47 i, identificador
48 ;; simbolo_ponto_virgula
49 END, END
50 ;; simbolo_ponto_virgula
51 END, END
52 ;; simbolo_ponto_virgula
53 PROCEDURE, PROCEDURE
54 ', <erro_lexico>

```

```

55 MultUpToN, identificador
56 ;; simbolo_ponto_virgula
57 VAR, VAR
58 temp, identificador
59 ;; simbolo_ponto_virgula
60 BEG, identificador
61 ^, <erro_lexico>
62 te, identificador
63 ^, <erro_lexico>
64 mp, identificador
65 :=, simbolo_atribuicao
66 1, numero
67 ;; simbolo_ponto_virgula
68 WHILE, WHILE
69 i, identificador
70 :=, simbolo_atribuicao
71 2, numero
72 temp, identificador
73 :=, simbolo_atribuicao
74 temp, identificador
75 *, simbolo_multiplicacao
76 i, identificador
77 ;; simbolo_ponto_virgula
78 END, END
79 ;; simbolo_ponto_virgula
80 END, END
81 ;; simbolo_ponto_virgula
82 n, identificador
83 :, <erro_lexico>
84 =, simbolo_igualdade
85 10, numero
86 ;; simbolo_ponto_virgula
87 j, identificador
88 :=, simbolo_atribuicao
89 0, numero
90 ;; simbolo_ponto_virgula
91 WHIL, identificador
92 @, <erro_lexico>
93 E, identificador
94 j, identificador
95 <, simbolo_menor
96 ^, <erro_lexico>
97 =, simbolo_igualdade
98 n, identificador
99 CALL, CALL
100 SumUpToN, identificador
101 ;; simbolo_ponto_virgula
102 j, identificador
103 :=, simbolo_atribuicao
104 j, identificador
105 +, simbolo_mais
106 1, numero
107 ;; simbolo_ponto_virgula
108 END, END
109 ., simbolo_ponto
110 END, END
111 ., simbolo_ponto

```

Código 6: arquivo de saída do teste.

Referências Bibliográficas

- [1] JFLAP - java formal languages and automata package. <https://www.jflap.org/>. Acesso em 09 de maio de 2024.
- [2] Hugo Nakamura, Nicholas Bragança, Isaac Soares, and Beatriz Lomes. MyFirst-Compiler. <https://github.com/ikuyorih9/MyFirstCompiler>, 2024.