Universidade de São Paulo
Escola de Engenharia de São Carlos
Departamento de Ciência de Computação

SCC0605 Teoria da Computação e Compiladores

Trabalho 1: análise léxica

Pedro Arthur Françoso - 12547301
Erik Melges - 12547399
Michel Hecker Faria - 12609690
Fernando Clarindo Cristovão - 12547573

Docente responsável: Prof. Thiago A. S Pardo

São Carlos
1º semestre / 2024

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Neste trabalho desenvolveu-se a parte do analisador léxico com base na gramática PL/0, sendo está a primeira etapa de um compilador, crucial para o funcionamento das demais etapas. O léxico desenvolvido é capaz de fazer a identificação e classificação dos tokens, além de identificar erros léxicos no código, para isso utilizou-se conceitos abordados na disciplina, como autômato de estados finitos e tabela de palavras e símbolos reservados. Repositório do projeto: https://github.com/Amigao/compiladores

2. ABORDAGEM

Para abordar o problema, começamos a elaboração do projeto fazendo a representação do autômato necessário com base na gramática disponível, que será introduzido a frente; também foi implementado uma tabela hash com as palavras e símbolos reservados para facilitar a busca na hora de conferir e classificar um token, além de uma lista encadeada para o armazenamento dos tokens detectados como errados.

3. AUTÔMATOS

A utilização de autômatos é parte essencial do analisador léxico, permitindo transitar entre os estados, identificar e classificar os tokens da linguagem PL/0. O autômato implementado serviu como base para o desenvolvimento do código das transições necessário dentro do analisador léxico, representado de maneira simplificado pela Figura 1.

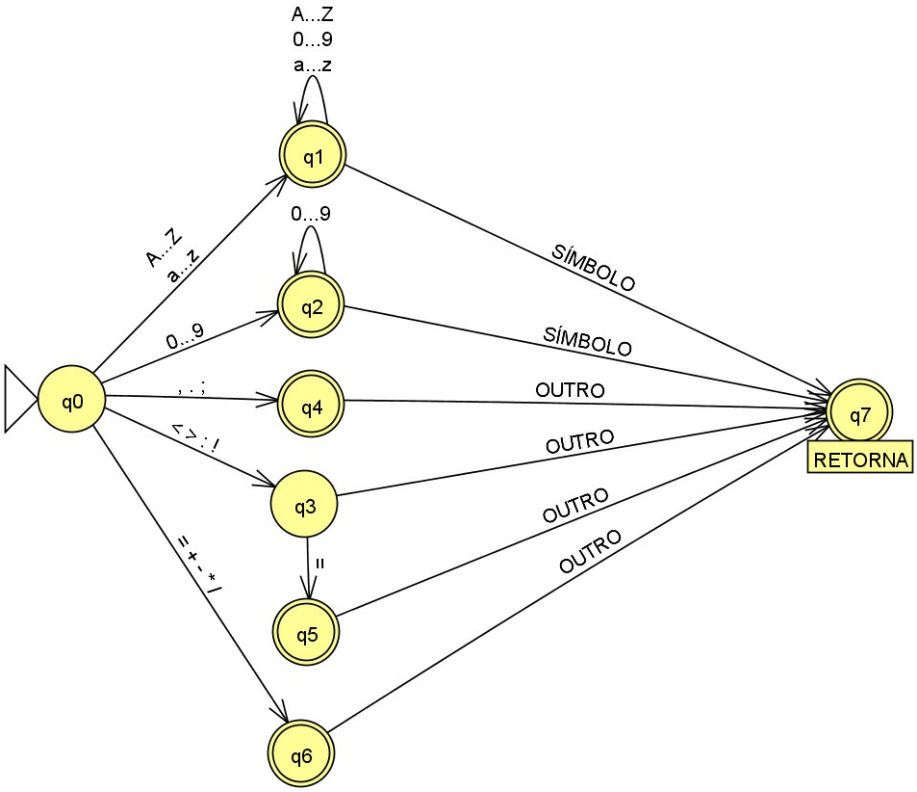

Figura 1: *Autômato inicial*

Então, um estado para interpretação de comentários foi adicionado, conforme mostrado pela Figura 2, não mostrado na Figura 1 para evitar poluição da figura.

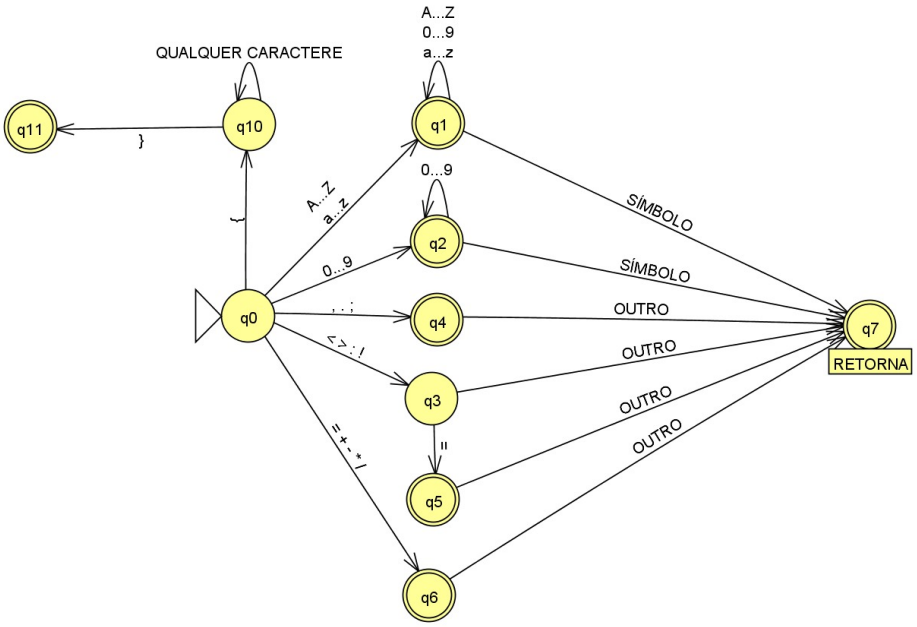

Figura 2: *Autômato com tratamento de comentários*

Cabe ressaltar que os tratamentos de erros são feitos através de um estado, aqui chamado de "-1". Portanto, o autômato real implementado, com todas as transições possíveis, pode ser observado pela Figura 3, cuja tabela de transição é a Tabela 1.

Qualquer outra transição além das mostradas no estado 0 também geram erro (estado -1), conforme a última linha da tabela de transições (Tabela 1) mostra e fica claro no *switch case* da função transição do código. Essa transição foi subtraída no desenho para evitar poluição da representação.

Além disso, o estado 7 é a posição onde o autômato reconhece que deve retroceder um caractere na cadeia de entrada e retornar o estado anterior, sendo fundamental no tratamento de cadeias comuns à linguagem PL/0, como quando mais de uma variável é declarada na mesma linha, usando vírgulas sem espaço. O bom funcionamento do código passou pela interpretação e implementação correta desse estado, sendo a maior dificuldade do projeto.

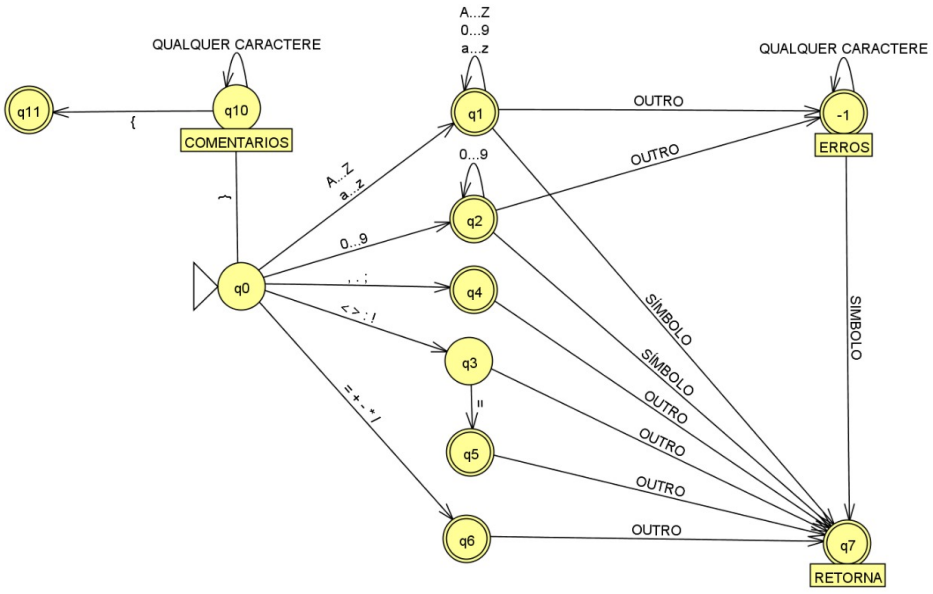

Figura 3: *Autômato completo*

Estado Atual	Condição de Entrada	Próximo Estado
0	isalpha(c)	1
0	isdigit(c)	2
0	is_first_double_operator(c)	3
0	is_delimiter(c)	4
0	is_single_operator(c)	6
0	c == '{'	10
0	c == '{'	10
1	isalpha(c) **ou** isdigit(c)	1
1	is_valid_symbol(c)	7
2	isalpha(c)	-1
2	isdigit(c)	2
2	is_valid_symbol(c)	7
3	is_second_double_operator(c)	5
3	qualquer	7
4	qualquer	7
5	qualquer	7
6	qualquer	7
10	c == '}'	11
10	qualquer	10
-1	is_valid_symbol(c)	7
qualquer outro	qualquer	-1

Tabela 1: *Tabela de Transição de Estados*

Função	Descrição
isalpha(c)	Verifica se 'c' é uma letra (A-Z, a-z)
isdigit(c)	Verifica se 'c' é um dígito (0-9)
is_first_double_operator(c)	Verifica se 'c' é o primeiro de um operador duplo (:, !, <, >)
is_second_double_operator(c)	Verifica se 'c' é o caractere '='
is_single_operator(c)	Verifica se 'c' é um operador de caractere único (=, +, -, *, /)
is_delimiter(c)	Verifica se 'c' é um delimitador (, ; .)
is_valid_symbol(c)	Verifica se 'c' é um símbolo válido (=, +, -, *, /, ,, ;, ., :, !, ¡, ¿, {)

Tabela 2: *Descrição das Funções Auxiliares*

4. CÓDIGOS

Esta seção descreve os módulos principais do analisador léxico desenvolvido em C, focando nas funções essenciais que compõem a base do compilador para a linguagem PL/0.

4.1. Gerenciamento de Erros

O módulo `errors_management.c` gerencia os erros léxicos encontrados durante a análise. Erros são registrados em uma lista encadeada, permitindo armazenar os erros juntamente com seu tipo (nesse caso, somente tipo 0, erro léxico) encontrados para um tratamento posterior.

Para caracteres inválidos no meio de palavras, foi escolhido tratar a cadeia toda como um erro, semelhante ao modo como o compilador *gcc* faz.

Principais Funções:

- `create_error_node`: Cria um novo nó para armazenar informações de erro.
- `insert_error`: Adiciona um erro detectado à lista ligada de erros.

- `printErrors`: Exibe todos os erros léxicos encontrados.
- `free_error_list`: Libera a memória alocada para a lista de erros.

```
// Definição da estrutura para um nó da lista ligada
typedef struct ErrorInfo{
    char *word;
    int line;
    int type;
    struct ErrorInfo *next;
} ErrorInfo;


// Função para liberar a memória alocada para a lista ligada
void printErrors(ErrorInfo *head) {
    printf("\n\n");
    ErrorInfo *current = head;
    while (current != NULL) {
        if (current->type == ERRO_LEXICO)
        printf("ERRO: erro LEXICO encontrado na linha %d.
        Termo \"%s\" mal formado.\n", current->line, current->word);
        current = current->next;
    }
    printf("\n\n");
}
```

Acima está representada a estrutura utilizada para armazenar os erros e a saída ao usuário.

4.2. Hashing

O arquivo `hashing.c` implementa funções de hash para otimizar a busca e inserção de tokens. A função de hash é projetada para minimizar conflitos usando características específicas dos tokens.

Funções de Hash:

- `hash_function`: Calcula o índice de hash baseado nos caracteres do token.
- `initialize_table`: Inicializa a tabela de hash.
- `insert_table`: Insere um token na tabela de hash.
- `search_table`: Busca um token na tabela de hash.
- `free_table`: Libera a memória utilizada pela tabela de hash.

A seguir estão algumas das partes do código desenvolvidas:

```
// Definição dos nós da table
typedef struct Node {
    char *word;
    char *token;
    struct Node *next;
} Node;

// Definição das tables (de simbolos e palavras reservadas e de simbolos e palavras)
typedef struct {
    Node *table[TABLE_SIZE];
} Table;
```

```
// Função de hash -> valores ASCII das primeiras duas letras da palavra e duas ultimas letras da pala
int hash_function(char *word) {
    int hash = 0;
    if(strlen(word) > 2){
        //Duas primeiras letras
        hash |= word[0];
        hash |= word[1] << 8;

        //Duas últimas letras
        int len = strlen(word);
        hash |= word[len - 2] << 16;
        hash |= word[len - 1] << 24;
    }    // Caso de strings de tamanho unitário ou de dois caracteres
    if(strlen(word) == 1) {
        // Apenas um caractere
        hash |= word[0];
    } else {
        // Dois caracteres
        hash |= word[0];
        hash |= word[1] << 8;
    }

    return hash % TABLE_SIZE;
}


// Função para inserir um par palavra-token na tabela hash
void insert_table(Table *table, char *word, char *token){
    int index = hash_function(word); // Calcula o índice usando a função de hash

    Node *new_node = malloc(sizeof(Node)); // Aloca memória para um novo nó
    if (new_node == NULL){ // Verifica se a alocação falhou
        fprintf(stderr, "Erro ao alocar memória para nó."); // Mensagem de erro se a alocação falhar
        exit(EXIT_FAILURE); // Sai do programa com falha
    }

    new_node->word = my_strdup(word); // Duplica a palavra e atribui ao novo nó
    new_node->token = my_strdup(token); // Duplica o token e atribui ao novo nó

    new_node->next = NULL; // Inicializa o ponteiro next do novo nó como NULL

    // Se a posição na tabela estiver vazia, insere o novo nó nessa posição
    if(table->table[index] == NULL){
        table->table[index] = new_node;
    } else { // Caso contrário, percorre a lista encadeada até o final e insere o novo nó lá
        Node *current = table->table[index];
        while(current->next != NULL){
            current = current->next;
        }
        current->next = new_node;
    }
```

```
}

// Função de busca
char *search_table(Table *table, char *word){
    int index = hash_function(word); // Calcula o índice usando a função de hash

    Node *current = table->table[index]; // Inicializa o ponteiro current com a cabeça da lista encad
    while (current != NULL) { // Percorre a lista encadeada
        if (strcmp(current->word, word) == 0) { // Compara a palavra atual com a palavra buscada
            return current->token; // Se encontrada, retorna o token associado
        }
        current = current->next; // Move para o próximo nó na lista
    }
    return NULL; // Se a palavra não for encontrada, retorna NULL
}
```

Nas funções apresentadas, vemos como foi definida a estrutura da tabela hash, além do método escolhido para inserir e buscar pelos elementos da tabela, sendo inserido um par para podemos identificar os símbolos e os identificadores atribuídos a eles. Outra parte importante é a função de busca na tabela, que em tempo constante consegue retornar o token associado ao buscado, caso encontrado.

O método de hashing escolhido [1] apresentou tempos de execução superiores a métodos de busca como árvore avl, lista encadeada e tabela hashing com método de hashing por multiplicação. Este método de hash calcula um valor baseado nos valores ASCII das duas primeiras e duas últimas letras de uma palavra. Para palavras com mais de dois caracteres, ele usa os valores ASCII das duas primeiras letras e os combina com os valores das duas últimas, aplicando deslocamentos de bits. Para palavras com um ou dois caracteres, ele usa os valores ASCII diretamente, com o segundo caractere sendo deslocado se houver.

Tabela 3: *Quantidade de palavras inseridas e buscadas por segundo*

Estrutura	Palavras Inseridas por Segundo	Palavras Buscadas por Segundo
AVL	115.4 mil	5.06 milhões
LISTA	111.1 mil	4.17 milhões
HASH POR MULTIPLICAÇÃO	113.2 mil	3.85 milhões
HASHING ADOTADO	**187.5 mil**	**7.27 milhões**

4.3. Analisador Léxico

O módulo `lexical_analyzer.c` é o coração do analisador léxico, processando a entrada para identificar tokens com base em um conjunto de regras definidas.

Operações Principais:

- Transições de estado para identificar caracteres e construir tokens.
- Uso de uma tabela de símbolos reservados para classificação de tokens.
- Gerenciamento de estados para determinar o fim da análise de um token.

```
int transition(int state, char c) {

    switch (state) {
        case 0:
            if (isalpha(c)) return 1;  // letras maiúsculas e minúsculas vão para o estado 1
```

[1] Inspirado em: https://www.youtube.com/watch?v=DMQ_HcNSOAI&t=429s

```
            if (isdigit(c)) return 2;  // números
            if (is_first_double_operator(c)) return 3;  // primeiro caractere de um operador duplo
            if (is_delimiter(c)) return 4;  // delimitadores
            if (is_single_operator(c)) return 6;  // operador com um caractere
            if (c == '{') return 10;  // comentario
            break;
        case 1:
            if (isalpha(c) || isdigit(c)) return 1;  // letras maiúsculas e minúsculas continuam no e
            if (is_valid_symbol(c)) return 7;
            break;
        case 2:
            if (isalpha(c)) return -1;
            if (isdigit(c)) return 2;
            if (is_valid_symbol(c)) return 7;
            break;
        case 3:
            if(is_second_double_operator(c)) return 5;
            else return 7;
            break;
        case 4:
        case 5:
        case 6:
            return 7;
            break;

        // Se entrar no estado de comentario, continua ate achar o simbolo de encerrar comentario
        case 10:
            if(c == '}') return 11;
            else return 10;
            break;


        // Estado de erro
        case -1:
            if(is_valid_symbol(c)) return 7;
            break;
    }
    return -1;  // qualquer outra transição leva a um estado de erro
}

// Funcao que sera chamada pelo sintatico
TokenInfo lexical_analyzer(char character, char *buffer, Table* reservedTable, int current_state){
    TokenInfo tok;
    tok.final = false;

    // Faz a transicao no automato baseado no caracter de entrada
    int new_state = transition(current_state, character);
    tok.state = new_state;
    tok.token = buffer;

    //se esta em um possivel estado final
    if (is_final_state(new_state)){
```

```
        int length = strlen(tok.token);
        // se for estado de retroceder, não adiciona
        if(new_state == RETURN_STATE){  o caracter da cadeia no token lido
            tok.token[length] = '\0';
        } else {
            tok.token[length] = character;
            tok.token[length + 1] = '\0';
        }

        tok.final = true;

        // Confere se eh um numero, erro ou se esta na tabela de palavras e simbolos reservados
        if (current_state == 2) tok.identifier = my_strdup("number");
        else if (current_state == -1) tok.identifier = my_strdup("ERRO LEXICO");
        else tok.identifier = check_reserved_table(reservedTable,tok.token);

    }

    // retorna o par token/classe
    return tok;
}
```

Acima estão representadas as partes principais do arquivo, nas quais as transições foram estabelecidas por um switch case, em que o autômato final será representado posteriormente. Temos que a função principal retorna uma struct que contem informações sobre o estado de entrada, o token e o seu identificador.

4.4. Programa Principal

O arquivo `main.c` somente inicia os arquivos de entrada e saída e chama a função de análise sintática (`sintatic analyzer`, por enquanto vazia) que coordena o processo de análise léxica, desde a leitura do arquivo de entrada até a geração do arquivo de saída com os tokens identificados.

Fluxo de Execução:

- Inicialização e configuração do ambiente de análise (tabelas de hash, listas de erros).
- Processamento contínuo do arquivo de entrada até que todo o texto seja analisado.
- Tratamento de erros e geração do arquivo de saída com os resultados da análise.

```
    // Enquanto nao acabar o arquivo
    while ((c = fgetc(input_file)) != EOF) {
        // contador de linhas
        if (c == '\n') {
            number_of_lines++;
        }

        // chegou no espaço ou \n indica, que acabou a token
        if (isspace(c) || c == '\n') {
            if(tok.state == 10 && isspace(c)){
                buffer[i] = c;
                buffer[i + 1] = '\0';
                current_state = tok.state;
                i++;
```

```
            continue;
        }

        if(tok.final){
            if (tok.state == -1){
                insert_error(&error_list, tok.token, number_of_lines, ERRO_LEXICO);
            }
            // imprime no arquivo de saida o par token/identificador
            fprintf(output_file, "%s, %s\n", tok.token, tok.identifier);
            //volta para o estado incial e reseta as Variaveis
            current_state = INITIAL_STATE;
            i = 0;
            buffer[i] = '\0';
            tok.final = false;
        }

    } else {

        // chama o lexico para cada caracter
        tok = lexical_analyzer(c, buffer, &reservedTable, current_state);

        // se entrar no estado de comentario
        if(tok.state == 11){
            buffer[i] = c;
            buffer[i+1] = '\0';
            // imprime o comentario que foi resetado
            printf("\nCOMENTARIO IGNORADO: %s\n", buffer);
            // reseta as variaveis
            current_state = INITIAL_STATE;
            i = 0;
            buffer[i] = '\0';
            tok.final = false;
        }

        // Se entrou no estado de retroceder
        else if (tok.state == RETURN_STATE) {
            if (current_state == -1){
                insert_error(&error_list, tok.token, number_of_lines, ERRO_LEXICO);
            }
            //adiciona ao arquivo de saida
            fprintf(output_file, "%s, %s\n", tok.token, tok.identifier);

            // devolve o caractere pra cadeia de entrada
            ungetc(c, input_file);
            // reseta as variaveis
            current_state = INITIAL_STATE;
            i = 0;
            buffer[i] = '\0';

        } else { // se nao, continua lendo e adicionando no buffer
            buffer[i] = c;
            buffer[i + 1] = '\0';
            current_state = tok.state;
```

```
                i++;
            }
        }
    }
```

Nesta etapa, é lido todo o arquivo de entrada e são interpretados os comandos retornados pelo léxico.

5. INSTRUÇÕES PARA COMPILAR E EXECUTAR

Dentro do diretório ou pasta do projeto, basta dar o comando

Make run

que o código será compilado e executado a partir do arquivo de texto padrão que está no mesmo diretório, *input.txt*, gerando o arquivo de saída *output.txt* e a saída de erros e exibição de comentários do código pelo terminal. O comando *Make run* compila e executa o código.

Caso o usuário queira usar outro arquivo de texto de entrada (outro código PL\0), deve rodar o programa com o comando

Make run ARGS=input2.txt

onde *input2.txt* é o arquivo que o usuário deseja "compilar" com o compilador PL\0.

Os comandos são os mesmos para ambientes Windows e Linux.

6. EXEMPLO DE EXECUÇÃO

Aqui iremos demonstrar como seria a saída do nosso programa para o seguinte arquivo de entrada:

input.tex

```
VAR MicHel123,b,c;
BEGIN
    a:=0;
    b:=3x;
    {primeiro comen}
    { comentario }
    c:=@+b;
    dlaspdla&spd;
END.
```

Ao rodar o programa será gerado um arquivo de saída:

output.tex

```
    VAR, <VAR>
MicHel123, ident
,, <VIRGULA>
b, ident
,, <VIRGULA>
c, ident
;, <PONTO_E_VIRGULA>
BEGIN, <BEGIN>
```

```
a, ident
:=, <SIMBOLO_ATRIBUICAO>
0, number
;, <PONTO_E_VIRGULA>
b, ident
:=, <SIMBOLO_ATRIBUICAO>
3x, ERRO LEXICO
;, <PONTO_E_VIRGULA>
c, ident
:=, <SIMBOLO_ATRIBUICAO>
@, ERRO LEXICO
+, <SIMBOLO_SOMA>
b, ident
;, <PONTO_E_VIRGULA>
dlaspdla&, ERRO LEXICO
spd, ident
;, <PONTO_E_VIRGULA>
END, <END>
., <PONTO>
```

além da seguinte saída no terminal:

Saída para o usuário

```
./programa input.txt

COMENTARIO IGNORADO: {primeiro comen}

COMENTARIO IGNORADO: { comentario }


ERRO: erro LEXICO encontrado na linha 4. Termo "3x" mal formado.
ERRO: erro LEXICO encontrado na linha 7. Termo "@" mal formado.
ERRO: erro LEXICO encontrado na linha 5. Termo "dlaspdla&spd" mal formado.
```

7. CONCLUSÃO

A título de conclusão, o desenvolvimento do projeto provou a necessidade de projetar o autômato com antecedência e a forma como fica facilitada a programação e desenvolvimento do projeto com esse autômato bem definido e corretamente construído.

Além disso, o projeto provou o caráter essencial das funções de *retroceder* e\ou marcador *lookahead*, já que sem o domínio desses métodos, seria praticamente impossível ler e interpretar um código da linguagem PL\0.

Por fim, o projeto deixou muito mais claro conceitos vistos em aula, até então sem aplicações práticas, e despertou interesse e participação por todos os membros do grupo, tendo cada um desenvolvido diferentes partes do projeto e contribuído com as decisões de projetos tomadas com diferentes interpretações a respeito de cada problema encontrado.