



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

**SCC0605 Teoria da Computação e Compiladores**

## **Trabalho 2: Análise Sintática**

**Pedro Arthur Franoso - 12547301**  
**Erik Melges - 12547399**  
**Michel Hecker Faria - 12609690**  
**Fernando Clarindo Cristovo - 12547573**

**Docente responsvel:** Prof. Thiago A. S Pardo

So Carlos  
1º semestre / 2024

# SUMÁRIO

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Código e Abordagem</b>	<b>1</b>
2.1	Modificações no léxico . . . . .	1
2.2	Analizador Sintático . . . . .	5
2.3	Gerenciamento de Erros . . . . .	6
<b>3</b>	<b>Instruções para Compilar e Executar</b>	<b>9</b>
3.1	Compilação e Execução Padrão . . . . .	9
3.2	Execução com Arquivo de Entrada Específico . . . . .	9
<b>4</b>	<b>Exemplo de execução</b>	<b>9</b>
4.1	Alteração na gramática . . . . .	10
4.2	Acentuação . . . . .	11
4.3	Palavras reservadas escritas com letras minúsculas . . . . .	11
<b>5</b>	<b>Conclusão</b>	<b>11</b>

## LISTA DE FIGURAS

- 1 Autômato, onde “OUTROS” representa tudo que for diferente dos caminhos possíveis para o estado e “SÍMBOLOS” são todos os símbolos do alfabeto de PL/0 (=, +, -, \*, /, ,, ;, :, !, >, <, {, }) . . . . . 1

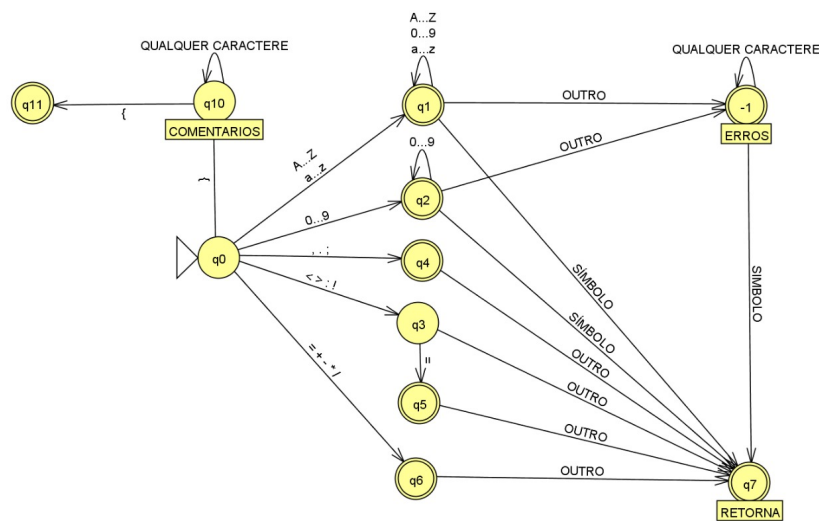
## 1. INTRODUÇÃO

Este relatório descreve o desenvolvimento de um analisador sintático para a linguagem PL/0, que é uma continuação do trabalho realizado anteriormente para o analisador léxico. Foi implementado um analisador sintático descendente preditivo recursivo, juntamente com o tratamento de erro do modo pânico. Os códigos estão disponíveis no repositório: <https://github.com/Amigao/compiladores>.

## 2. CÓDIGO E ABORDAGEM

Nesta seção serão descritas as abordagens e os códigos implementados para estruturar o trabalho.

### 2.1. Modificações no léxico



**Figura 1:** Autômato, onde “OUTROS” representa tudo que for diferente dos caminhos possíveis para o estado e “SÍMBOLOS” são todos os símbolos do alfabeto de PL/0 (=, +, -, \*, /, ,, ;, :, !, >, <, {, })

Não houveram mudanças substanciais nas transições do analisador léxico, ou seja, o autômato permanece basicamente o mesmo, representado na Figura 1. Na estrutura do código *lexical\_analyzer.c*, uma função chamada *getNextToken* foi criada, corrigindo alguns erros de projeto do primeiro trabalho, de forma que ela retorne somente um único token, ao invés de consumir o arquivo de texto todo, quando solicitado pelo analisador sintático, e lida com tudo que for diferente disso, como comentários e quebras de linha (*/n*). A nova função pode ser vista abaixo:

```
// Função para obter o próximo token da entrada
TokenInfo getNextToken(CompilingInfo *comp_info) {
    TokenInfo token_info;
    memset(&token_info, 0, sizeof(TokenInfo));

    // Estado inicial do autômato
    int current_state = INITIAL_STATE;

    // Variáveis para percorrer o arquivo e armazenar token/classe
    char character;
    char buffer[MAX_BUF_SIZE];
    int buffer_index = 0;
    buffer[0] = '\0';
```

---

```

// Enquanto não for o fim do arquivo
while ((character = fgetc(comp_info->input_file)) != EOF) {
    // Contador de linhas
    if (character == '\n') {
        comp_info->current_line++;
    }

    // Se espaço ou nova linha indica fim do token
    if (character == ' ' || character == '\n') {
        if (token_info.state == 10) {
            if (character == ' ') {
                buffer[buffer_index] = character;
                buffer[buffer_index + 1] = '\0';
                current_state = token_info.state;
                buffer_index++;
                continue;
            } else {
                insert_error(comp_info, ERRO_COMENTARIO_NAO_FECHADO, token_info.token);
                // Reseta o estado e as variáveis
                current_state = INITIAL_STATE;
                token_info.state = INITIAL_STATE;
                buffer_index = 0;
                buffer[buffer_index] = '\0';
                token_info.final = false;
                continue;
            }
        }
    }

    if (token_info.final) {
        if (token_info.state == -1) {
            insert_error(comp_info, ERRO_LEXICO, token_info.token);
            token_info.token_enum = IDENT;
        }
        // Reseta o estado e as variáveis
        current_state = INITIAL_STATE;
        buffer_index = 0;
        buffer[buffer_index] = '\0';
        token_info.final = false;
        return token_info;
    }
} else {

    // Chama o analisador léxico para cada caractere
    token_info = lexical_analyzer(character, buffer, &comp_info->reservedTable, current_state);

    // Se entrou no estado de comentário
    if (token_info.state == 11) {
        buffer[buffer_index] = character;
        buffer[buffer_index + 1] = '\0';
        // Reseta o estado e as variáveis
        current_state = INITIAL_STATE;
        buffer_index = 0;
        buffer[buffer_index] = '\0';
    }
}

```

---

```

        token_info.final = false;
    }
    // Se entrou no estado de retorno
    else if (token_info.state == RETURN_STATE) {
        if (current_state == -1) {
            insert_error(comp_info, ERRO_LEXICO, token_info.token);
            token_info.token_enum = IDENT;
        }
        // Devolve o caractere para a cadeia de entrada
        ungetc(character, comp_info->input_file);
        // Reseta o estado e as variáveis
        current_state = INITIAL_STATE;
        buffer_index = 0;
        buffer[buffer_index] = '\0';
        return token_info;
    } else { // Caso contrário, continua lendo e adicionando no buffer
        buffer[buffer_index] = character;
        buffer[buffer_index + 1] = '\0';
        current_state = token_info.state;
        buffer_index++;
    }
}

// Se o buffer estiver vazio, retorna token EOF
if (buffer_index == 0) {
    strncpy(token_info.token, "EOF", sizeof(token_info.token) - 1);
    token_info.token[sizeof(token_info.token) - 1] = '\0';
    token_info.token_enum = ENDOFFILE;
}

return token_info;
}

```

Apesar das transições serem as mesmas, um novo estado de retorno foi criado e é identificado fora das transições e retorna o final do arquivo (EOF), que será útil para indicar quando o gerenciamento de erro pelo modo pânico deve parar por definitivo (consumiu até o final do arquivo). Essa verificação é feita através do contador de caracteres do buffer de entrada, que caso esteja vazio, indica que o arquivo de entrada chegou ao final sem adicionar nada ao buffer, ou seja, não há mais tokens a serem lidos pelo léxico e interpretados pelo sintático. Pode ser vista abaixo:

```

...
// Se o buffer estiver vazio, retorna token EOF
if (buffer_index == 0) {
    strncpy(token_info.token, "EOF", sizeof(token_info.token) - 1);
    token_info.token[sizeof(token_info.token) - 1] = '\0';
    token_info.token_enum = ENDOFFILE;
}
...

```

Em relação a estrutura geral do código, algumas outras adições foram feitas para facilitar a passagem de informações entre as funções e para aumentar o tempo de comparação entre variáveis e reduzir o custo de armazenamento através. Entre elas, destacam-se o enum com todos as variáveis finais e

---

a struct de informações do estado de compilação atual: semelhante àquela que carrega os dados do token que está sendo analisado no momento, ela carrega dados importantes e comuns para o analisador sintático e léxico.

```
typedef enum {
// Terminais
CONST,
IDENT,
NUMERO,
VIRGULA,
PONTO_E_VIRGULA,
IGUAL,
ATRIBUICAO,
VAR,
PROCEDURE,
CALL,
BEGIN,
END,
IF,
THEN,
WHILE,
DO,
SUBTRACAO,
SOMA,
MULTIPLICACAO,
DIVISAO,
PARENTESE_ESQUERDA,
PARENTESE_DIREITA,
ODD,
DIFERENTE,
MENOR,
MENOR_IGUAL,
MAIOR,
MAIOR_IGUAL,
PONTO,
ENDOFFILE,
TOKEN_COUNT, //contador do número de tokens
} TokenType;
```

```
typedef struct CompilingInfo {
    FILE* input_file;
    ErrorInfo **error_list;
    Table reservedTable;
    int current_line;
} CompilingInfo;
```

Essas adições facilitaram muito a legibilidade do código, a velocidade de programação e a redução das passagens de parâmetros desnecessárias. Além dessas novas estruturas, todas as outras que já estavam presentes e eram utilizadas por ambos analisadores foram inseridos em um novo arquivo *aux.structs.h*.

---

## 2.2. Analisador Sintático

No desenvolvimento do sintático, utilizamos os procedimentos vistos em aula do analisador descendente preditivo recursivo com base na gramática do PL/0 fornecida, em que a entrada é processada da esquerda para a direita, gerando uma árvore de derivação a partir do nó raiz, no nosso caso o *<programa>*, fazendo chamadas recursivas para cada regra gramatical.

Como a estrutura da árvore é basicamente a mesma para todas regras, vamos apenas mostrar a implementação de algumas partes do código baseado na gramática.

A princípio, temos a função principal que começa chamando as regras e inicializando parâmetros importantes.

```
void sintatic_analyzer(FILE *input_file, FILE *output_file) {
    CompilingInfo comp_info;
    comp_info.input_file = input_file;
    comp_info.error_list = &error_list;
    comp_info.current_line = 1;

    // Constroi tabela reservada
    build_reserved_table(&comp_info.reservedTable);

    // NÓ RAÍZ
    programa(&comp_info);

    // Imprime os erros encontrados
    printErrors(error_list, output_file);

    // Libera as tabelas e listas utilizadas
    free_error_list(error_list);
    free_table(&reservedTable);
}
```

Nessa primeira função, inicializamos as tabelas e listas utilizadas (lista de erro e tabela de símbolos reservados), e inicializamos a derivação da árvore a partir do *<programa>*, seguido de:

```
void programa(CompilingInfo *comp_info) {
    token_info = getNextToken(comp_info);
    bloco(comp_info);
    if (token_info.token_enum != PONTO) {
        printf("Erro: '.' esperado no final do programa.\n");
        insert_error(comp_info, ERRO_SINTATICO, "'.' esperado no final do programa.");
        TokenType sync[] = {PONTO, ENDOFFILE, BEGIN, END};
        panic_mode(comp_info, sync, sizeof(sync)/sizeof(sync[0]));
        return;
    }
}

void bloco(CompilingInfo *comp_info) {
    declaracao(comp_info);
    comando(comp_info);
}

void declaracao(CompilingInfo *comp_info) {
    constante(comp_info);
    variavel(comp_info);
    procedimento(comp_info);
}
```



---

```
}
```

Veja que essa parte segue a estrutura inicial da gramática:

```
<programa> ::= <bloco> .
<bloco> ::= <declaracao> <comando>
<declaracao> ::= <constante> <variavel> <procedimento>
```

De modo que *<programa>* chama *<bloco>*, o *<bloco>* chama *<declaracao>* e *<comando>*, e assim por diante. Veja que a *<declaracao>* começa com uma *<constante>*, seguindo a estrutura:

```
void constante(CompilingInfo *comp_info) {
    if (token_info.token_enum == CONST) {
        token_info = getNextToken(comp_info);
        if (token_info.token_enum != IDENT) {
            printf("Erro: Identificador esperado apos 'CONST'.\n");
            insert_error(comp_info, ERRO_SINTATICO, "Identificador esperado apos 'CONST'.");
            TokenType sync[] = {PONTO_E_VIRGULA, VAR, PROCEDURE, IDENT, CALL, BEGIN, IF,
                               WHILE, END, ENDOFFILE};
            panic_mode(comp_info, sync, sizeof(sync)/sizeof(sync[0]));
            return;
        }

        token_info = getNextToken(comp_info);
        if (token_info.token_enum != IGUAL) {
            printf("Erro: '=' esperado apos identificador.\n");
            TokenType sync[] = {PONTO_E_VIRGULA, VAR, PROCEDURE, IDENT, CALL, BEGIN, IF,
                               WHILE, END, ENDOFFILE};
            panic_mode(comp_info, sync, sizeof(sync)/sizeof(sync[0]));
            return;
        }
    }
    ...
}
```

Aqui vemos que a estrutura possui terminais e não terminais esperados, como dentro da regra *<constante>* espera-se que haja um terminal **CONST** a ser lido, seguido de um **ident** e os demais símbolos terminais e não terminais, como mostrado a seguir (leia-se  $\lambda$  em /lambda):

```
<constante> ::= CONST ident = numero <mais_const> ; | /lambda
```

Seguindo essa estrutura para toda a gramática desenvolvemos a base do analisador sintático.

## 2.3. Gerenciamento de Erros

Perceba que dentro das chamadas do sintático mostrada, temos algumas estruturas do tratamento de erro presentes dentro da função sempre que o terminal esperado não é encontrado. Como pedido, utilizamos o tratamento pelo modo de pânico, que consiste em, ao encontrar algum token inesperado dentro de uma regra, consumir palavras até encontrar um caractere de sincronização para voltar a compilação do programa. A função principal do modo pânico é a seguinte:

```
void panic_mode(CompilingInfo* aux, TokenType sync[], int sync_count) {
    int i;
    while (1) {
```

```

// Verifica se o token atual é um token de sincronização
for (i = 0; i < sync_count; i++) {
    if (tok.token_enum == (int)sync[i] || tok.token_enum == ENDOFFILE) {
        if (tok.token_enum == ENDOFFILE) {
            printf("\n\nO modo de pânico consumiu até o final do arquivo :( !!!!\n\n");
            exit(-1);
        }
        return; // Encontrou um símbolo de sincronização, sai da função
    }
}

tok = getNextToken(aux); // Continua consumindo tokens
até encontrar um símbolo de sincronização
}
}

```

Ao ser chamada, ela recebe uma lista com os tokens de sincronização de cada não terminal, e começa a procurar por cada um deles dentro do código, consumindo caracteres até encontrar, conferindo se não chegou até o final do arquivo. Para cada símbolo não terminal, sendo A o símbolo a ser consumido, foram escolhidos símbolos de sincronização baseados nas seguintes regras:

- Seguidores(A)
- Seguidores do pai de A
- Símbolos de sincronização extra (escolhidos manualmente)

Representamos os símbolos de sincronização de cada não terminal na Tabela 1 a seguir:

**Tabela 1:** Tabela de Símbolos de sincronização

Não Terminal	Seguidores	Seguidores do Pai	Símbolos Extras	Símbolos de Sincronização
<programa>	.	N/A	BEGIN, END	., BEGIN, END
<bloco>	.	.	BEGIN, END	., BEGIN, END
<declaracao>	ident, CALL, BEGIN, IF, WHILE, END, .	.	;	ident, CALL, BEGIN, IF, WHILE, END, ;; .
<constante>	VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .	ident, CALL, BEGIN, IF, WHILE, END, .	;	VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .
<mais_const>	;	VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .	;	;; VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, .
<variavel>	PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .	ident, CALL, BEGIN, IF, WHILE, END, .	;	PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .

Não Terminal	Seguidores	Seguidores do Pai	Símbolos Extras	Símbolos de Sincronização
<mais_var>	;	PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, ;; .	;	;;, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, END, .
<procedimento>	ident, CALL, BEGIN, IF, WHILE, END, ;; .	ident, CALL, BEGIN, IF, WHILE, END, .	;	ident, CALL, BEGIN, IF, WHILE, END, ;; .
<comando>	;;, END, .	END, .	;	;;, END, .
<mais_cmd>	END	END	;	END, ;
<expressao>	;;, ), THEN, DO, END	;;, ), THEN, DO, END	;	;;, ), THEN, DO, END
<operador_unario>	ident, numero, (	ident, numero, (	;	ident, numero, (, ;
<termo>	;;, ), THEN, DO, +, -, END	;;, ), THEN, DO, +, -, END	;	;;, ), THEN, DO, +, -, END
<mais_termos>	;;, ), THEN, DO, END	;;, ), THEN, DO, END	;	;;, ), THEN, DO, END
<fator>	*, /, +, -, ;;, ), THEN, DO, END	*, /, +, -, ;;, ), THEN, DO, END	;	*, /, +, -, ;;, ), THEN, DO, END
<mais_fatores>	;;, ), THEN, DO, +, -, END	;;, ), THEN, DO, +, -, END	;	;;, ), THEN, DO, +, -, END
<condicao>	THEN, DO	THEN, DO	;	THEN, DO, ;
<relacional>	ident, numero, (	ident, numero, (	;	ident, numero, (, ;

Baseado na tabela acima adequamos os caracteres de sincronização para cada não terminal.

Uma única e pequena alteração que optamos por fazer na gramática para que uma maior variação de erros fosse detectada foi feita no bloco <mais\_comandos>, assumindo a seguinte forma:

```

void mais_comando(CompilingInfo *comp_info) {
switch (token_info.token_enum)
{
case IDENT:
case CALL:
case BEGIN:
case WHILE:
case IF:
printf("Erro: ';' esperado apos comando.\n");
insert_error(comp_info, ERRO_SINTATICO, "';' esperado apos comando.");
comando(comp_info);
mais_comando(comp_info);
break;
case PONTO_E_VIRGULA:
token_info = getNextToken(comp_info);
comando(comp_info);
mais_comando(comp_info);
break;
}
}

```

Diferente da original:

```

void mais_comando(CompilingInfo *comp_info) {
if (token_info.token_enum == PONTO_E_VIRGULA){

```

```

        token_info = getNextToken(comp_info);
        comando(comp_info);
        mais_comando(comp_info);
    }
}

```

Essa alteração permite que a ausência de ponto-vírgula ao longo de um bloco de comando seja informada e evita que todo o bloco em questão seja consumido pelo modo pânico. Fazer isso através de verificações dentro do bloco *<comando>* estava gerando problemas devido à recursividade entre esses dois blocos, de modo que muitas vezes o token de sincronização lido no modo pânico era o **END** que finaliza o código, fazendo com que todo o código fosse consumido por causa de um pequeno erro de digitação do programador teórico. Contornar isso dentro das verificações até foi possível, mas com uma legibilidade muito pior e um grau de complexidade desnecessário. Um exemplo dessa alteração vai ser apresentada nos exemplos de execução.

### 3. INSTRUÇÕES PARA COMPILAR E EXECUTAR

#### 3.1. Compilação e Execução Padrão

No diretório do projeto, execute o seguinte comando no terminal para compilar e rodar o analisador usando o arquivo de entrada padrão (*input.txt*):

```
make run
```

Este comando compila os módulos necessários e executa o analisador, lendo o arquivo *input.txt* localizado no diretório do projeto. O resultado da análise será escrito no arquivo *output.txt*, e os erros ou comentários processados serão exibidos no terminal.

#### 3.2. Execução com Arquivo de Entrada Específico

Caso deseje utilizar um arquivo de entrada diferente, o comando a ser utilizado é:

```
make run ARGS=<nome_do_arquivo>
```

Substitua *<nome\_do\_arquivo>* pelo nome do arquivo que deseja analisar. Por exemplo, para analisar o arquivo *input2.txt*, o comando seria:

```
make run ARGS=input2.txt
```

Este comando irá compilar, caso necessário, e executar o analisador usando o arquivo especificado, gerando a saída correspondente no arquivo *output.txt* e exibindo os erros e comentários no terminal.

### 4. EXEMPLO DE EXECUÇÃO

Utilizando o exemplo fornecido na especificação de trabalho, temos como arquivo de entrada:

```

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

```

E arquivo de saída *output.txt*:

ERRO SINTATICO: ';' esperado apos declaracao de variavel. Linha 1.  
ERRO SINTATICO: 'THEN' esperado. Linha 4.  
ERRO LEXICO: Termo "@" mal formado. Linha 6.

Vemos que foram detectados os erros, indicando se são erros sintáticos ou léxicos, além do tipo de erro encontrado.

A fim de fazer mais testes, vejamos a detecção de erros para o seguinte arquivo de entrada:

```
1 const
2   max = 10;
3
4 var
5   n, result
6 procedure factorial;
7 VAR
8   i;
9 BEGIN
10  result := @;
11  i := n;
12  WHILE i > 1 DO
13  BEGIN
14    result := result * i
15    i := i - 1
16  END
17 END;
18
19 {comentário com acentos e fechado corretamente ç â ô ! ^}
20 {comentário com acentos e fechado de forma errônea ç â ô ! ^
21
22 BEGIN
23   n := 5;
24   CALL factorial;
25 END.
```

Como arquivo de saída temos:

ERRO SINTATICO: ';' esperado apos declaracao de variavel. Linha 5.  
ERRO LEXICO: Termo "@" mal formado. Linha 10.  
ERRO SINTATICO: ';' esperado apos comando. Linha 14.  
ERRO: comentario nao fechado encontrado na linha 20: "{comentário com acentos e não fechado ç â ô ! ^"

Aqui, podemos ver o resultado da alteração na gramática supramencionada e a correção dos erros aparentes no primeiro trabalho.

#### 4.1. Alteração na gramática

Na linha 14 (*result := result \* i*), a gramática original indicaria um erro de ausência de **END**, já que o bloco *<mais\_comandos>* só seria chamado se houvesse um ponto e vírgula ao final da linha. Como não há, a gramática diz que o bloco de comando acabou ali e, por isso, deveria haver um terminal **END** ao invés do identificador *i* encontrado. No entanto, é evidente que o programador teórico adicionou um comando a mais (linha 15), o que corresponderia ao bloco *<mais\_comandos>*, mas esqueceu de inserir o ponto e vírgula. A gramática atual possui, portanto, mais formas de se entrar no bloco *<mais\_comandos>*, mas informa ao usuário que isso é um erro. Essa mudança facilita a correção do erro pelo programador teórico e também a execução do método de pânico.

---

## 4.2. Acentuação

No primeiro trabalho, um dos problemas observados foi que o código não conseguia lidar com acentos. O erro foi encontrado, de modo que devido a um estado que não era final estar sendo considerado final, palavras com acentos estavam sendo procuradas na tabela de símbolos reservados, o que quebrava o método de hashing, e foi corrigido, de modo que agora os comentários, como os das linhas 20 e 21, aceitam qualquer caractere, seja um acento, uma letra acentuada, etc, e informa também o erro de comentário não fechado ao usuário.

## 4.3. Palavras reservadas escritas com letras minúsculas

Como pode-se observar no arquivo de entrada, **const**, **var** e **procedure** são símbolos terminais que constituem a tabela de símbolos reservados e estão escritos em letras minúsculas, e agora são identificadas corretamente deixando de ser um problema do código.

# 5. CONCLUSÃO

Neste projeto, conseguimos avançar na construção do compilador para a linguagem PL/0, trabalhando em uma das etapas mais importantes da compilação, a análise sintática. Para começar a desenvolver o sintático, foi necessário fazer pequenas modificações no léxico apresentado no primeiro projeto, adequando-o para as novas condições e corrigindo erros cometidos. No desenvolvimento do sintático, projetamos um analisador sintático descendente preditivo recursivo, sendo um método de simples implementação e eficiente. Para isso derivamos cada conjunto de não terminais em procedimentos possivelmente recursivos seguindo a gramática do PL/0. Também foi abordado nesse trabalho técnicas de tratamento de erro para o analisador sintático, sendo implementado o tratamento pelo modo de pânico, sendo também de fácil implementação, simplesmente fazendo uma busca por tokens de sincronização sempre que um token esperado não é encontrado, utilizando dos conhecimentos de seguidor e seguidor do pai, além de símbolos extras escolhidos por nós para servirem como tokens de parada, tomando cuidado para não consumir o programa inteiro. O contato com o conceito de tratamento de erros nos deu a liberdade e segurança para fazer pequenas alterações na gramática de modo a facilitar ainda mais a identificação de erros do programador (teórico) que estaria usando a linguagem PL/0.