

UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

## Trabalho 2 - Análise Sintática da Linguagem PL/0

Carlos Henrique Hannas de Carvalho	nº USP: 11965988
Henrique Carobolante Parro	nº USP: 11917987
Pedro Antonio Bruno Grando	nº USP: 12547166
Weverton Samuel Alves	nº USP: 11917326

Prof. Thiago A. S. Pardo

SCC0605 - Teoria da Computação e Compiladores

23 de junho de 2024

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Alterações do Trabalho 1</b>	<b>1</b>
<b>3</b>	<b>Decisões de Projeto</b>	<b>3</b>
3.1	Primeiro e Seguidores . . . . .	3
3.2	Alerações na gramática . . . . .	6
3.3	Modelagem dos Grafos Sintáticos . . . . .	6
3.3.1	Grafo de Programa . . . . .	6
3.3.2	Grafo de Constante . . . . .	7
3.3.3	Grafo de Variável . . . . .	7
3.3.4	Grafo de Procedimento . . . . .	7
3.3.5	Grafo de Primeiro Comando . . . . .	7
3.3.6	Grafo de Comando . . . . .	8
3.3.7	Grafo de Expressão . . . . .	8
3.3.8	Grafo de Condição . . . . .	9
<b>4</b>	<b>Código</b>	<b>9</b>
4.1	Compilação do Código . . . . .	10
<b>5</b>	<b>Exemplo de Execução</b>	<b>10</b>
<b>6</b>	<b>Conclusão</b>	<b>11</b>

## Lista de Figuras

1	Autômato de comentário da linguagem PL/0. . . . .	2
2	Autômato de número inteiro da linguagem PL/0. . . . .	2
3	Autômato completo da linguagem PL/0. . . . .	3
4	Grafo do não-terminal <programa>. . . . .	6
5	Grafo do não-terminal <constante>. . . . .	7
6	Grafo do não-terminal <variavel>. . . . .	7
7	Grafo do não-terminal <procedimento>. . . . .	7
8	Grafo do não-terminal <prim_comando>. . . . .	8
9	Grafo do não-terminal <comando>. . . . .	8
10	Grafo do não-terminal <expressao>. . . . .	9
11	Grafo do não-terminal <condicao>. . . . .	9
12	Execução do exemplo disponibilizado pelo docente e seu resultado. . . . .	10
13	Execução do código de teste dos alunos e seus resultados. . . . .	11

## Lista de Tabelas

1	Conjunto de primeiros e seguidores dos símbolos não-terminais da linguagem PL/0. . . . .	4
2	Conjunto de seguidores e seguidores do Pai para os símbolos de sincronização da linguagem PL/0. . . . .	5

# 1 Introdução

A principal etapa de um compilador, durante a execução de um programa-fonte, é a análise sintática - que será estudada ao longo deste projeto. O analisador sintático gerencia todo processo de compilação, coordenando etapas anteriores e posteriores, além de verificar a combinação de *tokens*, após a análise léxica. Ou seja, o período de sintaxe verifica a formação do programa e reconhece as cadeias pertencentes à linguagem, de acordo com a gramática.

O objetivo deste projeto é desenvolver a análise sintática da linguagem de programação PL/0. O analisador sintático será projetado, a partir do léxico (Trabalho 1), para processar um código em PL/0 e detectar eventuais erros, léxicos e sintáticos, que o programa-fonte apresenta. A gramática da linguagem PL/0 e demais informações do Trabalho 1 estão disponíveis para visualização em [1] e [2], respectivamente.

A fim de alcançar o objetivo, desenvolve-se e implementa-se, em linguagem C, um analisador sintático descendente preditivo recursivo para PL/0. Além desse desenvolvimento, faz-se algumas correções em relação ao analisador léxico (Trabalho 1). O resultado final será um analisador sintático funcional: etapa principal de um compilador.

Este relatório inclui as decisões de projeto, tais como lógica de implementação e síntese entre analisadores léxico e sintático, além de fornecer instruções detalhadas para compilação e execução do código. Por fim, também será apresentado um exemplo de execução, demonstrando o analisador sintático desenvolvido.

## 2 Alterações do Trabalho 1

Identificou-se algumas alterações necessárias no analisador léxico, para dar continuidade à análise sintática. Elas são listadas em tópicos, a seguir, e foram atualizadas nos códigos:

- Tratamento para caracteres fora do padrão ASCII;
- limite de comentários para uma linha;
- tratamento de erro para números reais;
- tratamento para parênteses.

Em um primeiro instante, o analisador léxico estava ausente de tratamento para caracteres fora do padrão ASCII. Um exemplo realizado durante a apresentação continha caracteres *ã* e *ã* - eles são tratados como *chars* negativos e, naturalmente, não fazem parte da linguagem PL/0. Dessa forma, adicionou-se um tratamento no código para eles serem consumidos pelo removedor de comentários.

Os comentários aceitos pela linguagem são aqueles pertencentes a uma única linha e entre-chaves. Dessa forma, para o Trabalho 2, adicionou-se uma transição e um novo estado, no autômato, que limitasse o comentário a uma linha. Abaixo segue um exemplo de cadeias válida e inválida, respectivamente, por PL/0:

- Cadeia válida: {*comentario*}
- Cadeia inválida: {  
                  *comentario*}

A figura 1 mostra o autômato de comentário atualizado:

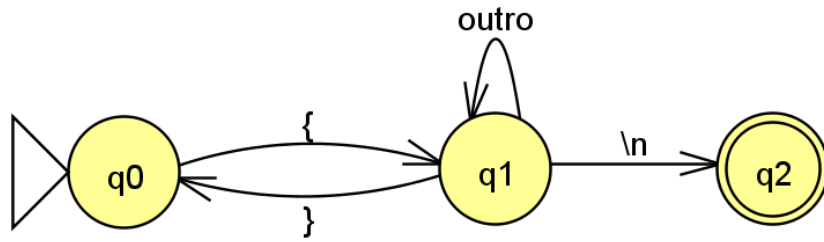


Figura 1: Autômato de comentário da linguagem PL/0.

Finalmente, a última alteração foi em relação à inserção de um tratamento de erro para números reais. Os números aceitos em PL/0 são apenas inteiros e o Trabalho 1 não tratava, como erro, as entradas de números reais ou números mal-formatados. Para isso, atualizou-se o autômato de números inteiros para tratamento de erros. Abaixo, a figura 2 apresenta o autômato de números inteiros atualizado:

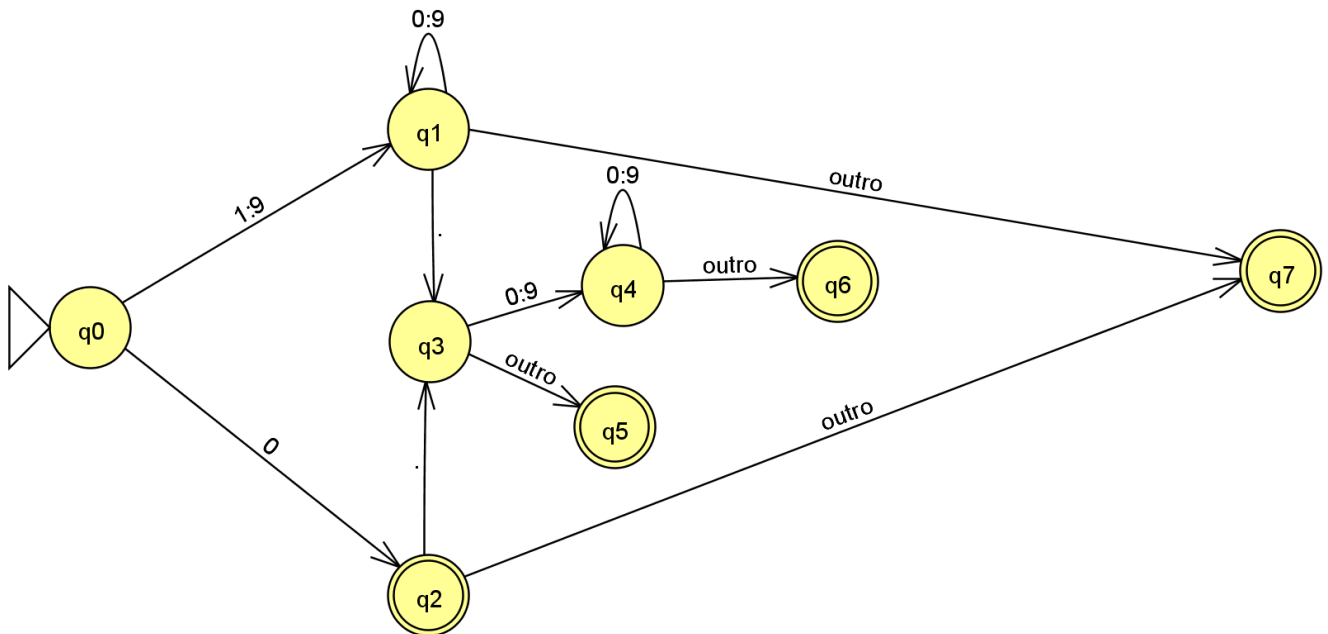


Figura 2: Autômato de número inteiro da linguagem PL/0.

Por fim, não foi considerada na primeira parte, por desatenção dos alunos, a inclusão de estados para tratar os parênteses. As alterações nos autômatos de comentário e números inteiros, vistos nas figuras 1 e 2, respectivamente, implicam em uma atualização do autômato final. A imagem 3 apresenta o autômato completo da linguagem PL/0:



Mapeou-se os primeiros e seguidores de todos não-terminais e projetou-se os grafos sintáticos, para a linguagem PL/0 - as seções 3.1 e 3.3 apresentam, respectivamente, o mapeamento de primeiros/seguidores e os grafos. Além disso, desenvolveu-se, em linguagem C, um código que implementa o analisador sintático do programa-fonte - presente na seção 4 do relatório.

A tabela 1 apresenta os primeiros e seguidores dos termos não-terminais da linguagem PL/0. Ela foi montada baseada na gramática presente em [1]:

Não Terminal	Primeiros	Seguidores
<programa>	{ CONST, VAR, PROCEDURE, BEGIN, CALL, IF WHILE, ident, . , $\lambda$ }	{ $\lambda$ }
<bloco>	{ CONST, VAR, PROCEDURE, BEGIN, CALL, IF WHILE, ident, $\lambda$ }	{ ; , . }
<declaracao>	{ CONST, VAR, PROCEDURE, $\lambda$ }	{ ident, CALL, BEGIN, IF, WHILE, . , ; }
<constante>	{ CONST, $\lambda$ }	{ VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, . , ; }
<mais_const>	{ , , $\lambda$ }	{ ; }
<variavel>	{ VAR, $\lambda$ }	{ PROCEDURE, ident, CALL, BEGIN, IF, WHILE, . , ; }
<mais_var>	{ , , $\lambda$ }	{ ; }
<procedimento>	{ PROCEDURE, $\lambda$ }	{ ident, CALL, BEGIN, IF, WHILE, . , ; }
<comando>	{ ident, CALL, BEGIN, IF, WHILE, $\lambda$ }	{ END, ; , . }
<mais_cmd>	{ ; , $\lambda$ }	{ END }
<expressao>	{ -, +, ident, numero, ( }	{ =, <>, <, ≤, >, ≥, ), END, ;, ., THEN, DO }
<operador_unario>	{ -, +, $\lambda$ }	{ ident, numero, ( }
<termo>	{ ident, numero, ( }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ;, ., THEN, DO }
<mais_termos>	{ +, -, $\lambda$ }	{ =, <>, <, ≤, >, ≥, ), END, ;, ., THEN, DO }
<fator>	{ ident, numero, ( }	{ *, / , -, +, =, <>, <, ≤, >, ≥, ), END, ;, ., THEN, DO }
<mais_fatores>	{ *, / , $\lambda$ }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ;, ., THEN, DO }
<condicao>	{ ODD, -, +, ident, numero, ( }	{ THEN, DO }
<relacional>	{ =, <>, <, ≤, >, ≥ }	{ -, +, ident, numero, ( }

Tabela 1: Conjunto de primeiros e seguidores dos símbolos não-terminais da linguagem PL/0.

Além disso, a tabela 2 apresenta os seguidores e seguidores do Pai, de cada não-terminal da gramática PL/0, para mostrar os símbolos de sincronização:

Não Terminal	Seguidores	Seguidores do Pai	Símbolos adicionais	Símbolos de Sincronização
<programa>	{ $\lambda$ }	{ $\lambda$ }	{ ; }	{ ; }
<bloco>	{ ; , . }	{ ident, CALL, BEGIN, IF, WHILE, ., ; , $\lambda$ }	{ $\lambda$ }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }
<declaracao>	{ ident, CALL, BEGIN, IF, WHILE, ., ; }	{ ; , . }	{ $\lambda$ }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }
<constante>	{ VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }	{ $\lambda$ }	{ VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }
<mais_const>	{ ; }	{ VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }	{ $\lambda$ }	{ VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }
<variavel>	{ PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }	{ $\lambda$ }	{ PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }
<mais_var>	{ ; }	{ PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }	{ $\lambda$ }	{ PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ., ; }
<procedimento>	{ ident, CALL, BEGIN, IF, WHILE, ., ; }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }	{ $\lambda$ }	{ ident, CALL, BEGIN, IF, WHILE, ., ; }
<comando>	{ END, ; , . }	{ ; , ., END }	{ $\lambda$ }	{ END, ; , . }
<mais_cmd>	{ END }	{ END, ; , . }	{ $\lambda$ }	{ END, ; , . }
<expressao>	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ END, ; , ., *, /, -, +, =, <>, <, ≤, >, ≥, ), THEN, DO }	{ $\lambda$ }	{ END, ; , ., *, /, -, +, =, <>, <, ≤, >, ≥, ), THEN, DO }
<operador_unario>	{ ident, numero, ( }	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ $\lambda$ }	{ ident, numero, (, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }
<termo>	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ $\lambda$ }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }
<mais_termos>	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ $\lambda$ }	{ =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }
<fator>	{ *, /, -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ $\lambda$ }	{ *, /, -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }
<mais_fatores>	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }	{ $\lambda$ }	{ -, +, =, <>, <, ≤, >, ≥, ), END, ; , ., THEN, DO }
<condicao>	{ THEN, DO }	{ END, ; , . }	{ $\lambda$ }	{ THEN, DO, END, ; , . }
<relacional>	{ -, +, ident, numero, ( }	{ THEN, DO }	{ ; }	{ -, +, ident, numero, (, ; , THEN, DO }

Tabela 2: Conjunto de seguidores e seguidores do Pai para os símbolos de sincronização da linguagem PL/0.

## 3.2 Alterações na gramática

Tomou-se liberdade de fazer algumas alterações substanciais na gramática, pois considerou-se que algumas estruturas possíveis nela eram ambíguas ou contra-produtivas para o programador e o projetista do compilador.

Primeiramente, adotou-se uma padronização que torna obrigatório o uso de ponto-e-vírgula (;) após uma chamada **CALL** ou após uma atribuição na seção de comandos, além de se retirar a exigência de um ponto-e-vírgula no início da regra **<mais\_cmd>**. As formas finais das regras aletradas são mostradas abaixo:

```
<comando> ::= CALL ident;  
<comando> ::= ident := <expressao>;  
<mais_cmd> ::= <comando><mais_cmd>
```

Ainda, observou-se que era possível fazer códigos que possuíam como corpo de comandos apenas uma única atribuição, sem a delimitação **BEGIN** e **END**. Portanto, criou-se uma regra que exige a utilização dessa estrutura para a declaração de um primeiro comando, e obrigou-se a utilizar-se essa regra sempre que o usuário desejar escrever um comando. A nova regra, bem como alterações em regras precedentes são mostradas abaixo:

```
<prim_comando> ::= BEGIN <comando><mais_comando>END |  $\lambda$   
<bloco> ::= <declaracao><prim_comando>
```

É importante notar que na execução do código, é possível passar pela regra **<prim\_comando>** não se digitando um bloco de comando. Contudo, se qualquer outro comando for digitado, que não as diretivas **BEGIN** e **END**, essa regra resultará em um erro que avisará ao usuário as diretivas necessárias para iniciar os comandos. Considerou-se que essa padronização torna a linguagem mais compreensível e reduz a complexidade do analisador sintático.

## 3.3 Modelagem dos Grafos Sintáticos

Os símbolos não-terminais da gramática PL/0 foram agrupados, de forma a formar 8 grafos principais: programa, constante, variável, procedimento, primeiro comando (**<prim\_comando>**), comando, expressão e condição. A ideia em agrupá-los é apresentar, de forma mais sucinta, as regras gramaticais da linguagem-fonte. Adotou-se as alterações mencionadas na seção 3.2 e os seguintes critérios, para o projeto dos grafos sintáticos:

- Símbolos não-terminais geram outros símbolos (terminais ou não-terminais);
- Símbolos não-terminais, gerados por um Pai, estão presentes em retângulos;
- Símbolos terminais, gerados por um Pai, estão presentes em elipses;
- Símbolo Pai não está presente em retângulo e nem em elipse.

### 3.3.1 Grafo de Programa

A figura 4 apresenta o grafo principal da linguagem - trata-se do nó raiz da árvore sintática de PL/0:



Figura 4: Grafo do não-terminal **<programa>**.

Expandiu-se dois não-terminais da gramática original (**<bloco>** e **<declaracao>**): o não-terminal **<bloco>** originalmente expande-se em **<declaracao>** e **<comando>**; por sua vez, o não-terminal **<declaracao>** expande-se em **<constante>** e **<variavel>**. Essas expansões resultam no diagrama da figura 4.



### 3.3.2 Grafo de Constante

O grafo de <constante> expande-se em  $\lambda$  ou em símbolos terminais seguido de um não-terminal <mais\_const>. O termo <mais\_const> foi agrupado, para facilitar a visualização da regra gramatical, conforme a figura 5:

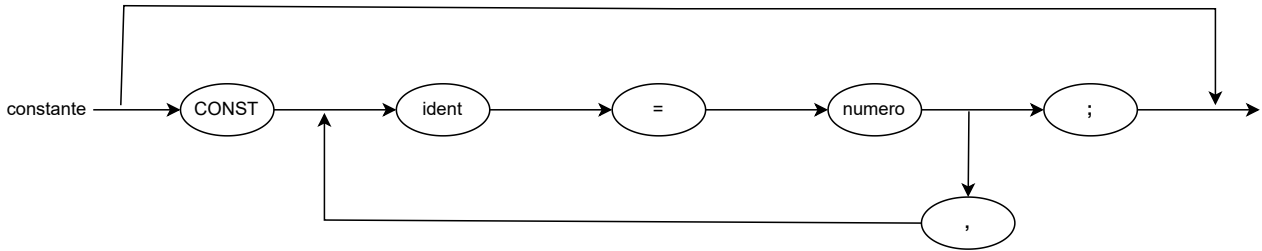


Figura 5: Grafo do não-terminal <constante>.

O termo  $\lambda$  é representado no grafo pela seta superior, que não apresenta símbolos terminais ou não-terminais.

### 3.3.3 Grafo de Variável

O grafo de <variavel> expande-se em  $\lambda$  ou em símbolos terminais seguido de um não-terminal <mais\_var>. O termo <mais\_var> foi agrupado, para facilitar a visualização da regra gramatical, conforme a figura 6:

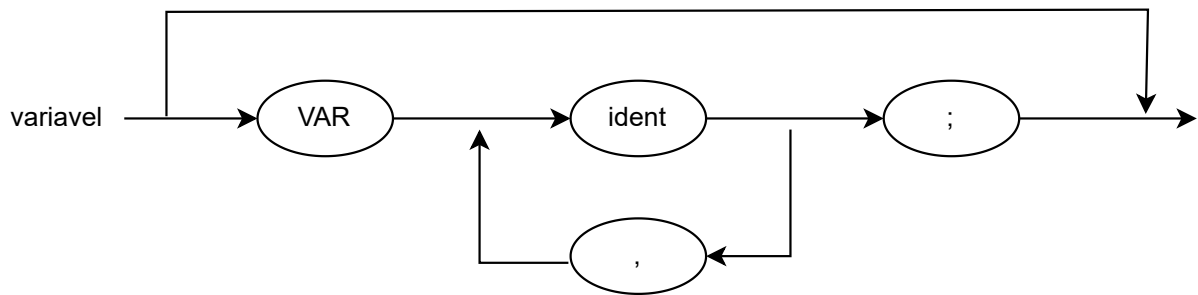


Figura 6: Grafo do não-terminal <variavel>.

O termo  $\lambda$  é representado no grafo pela seta superior, que não apresenta símbolos terminais ou não-terminais.

### 3.3.4 Grafo de Procedimento

A figura 7 apresenta o grafo de <procedimento>:

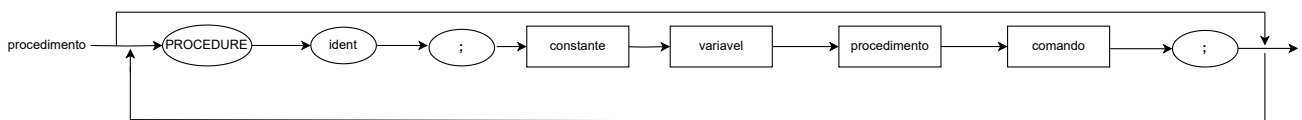


Figura 7: Grafo do não-terminal <procedimento>.

O nó Pai da figura 7 gera  $\lambda$ , representado pela seta superior, que não apresenta símbolos terminais ou não-terminais; ou gera símbolos terminais, seguido de um não-terminal <bloco>, expandido semelhantemente à seção 3.3.1, seguido de um terminal ; e outro não-terminal (procedimento) - esse último é representado pela realimentação da seta inferior, que não apresenta símbolos terminais ou não-terminais.

### 3.3.5 Grafo de Primeiro Comando

A regra gramatical permite que o grafo <prim\_comando> gere os símbolos, BEGIN, <comando>, <mais\_cmd> e END, respectivamente, conforme a figura 8:

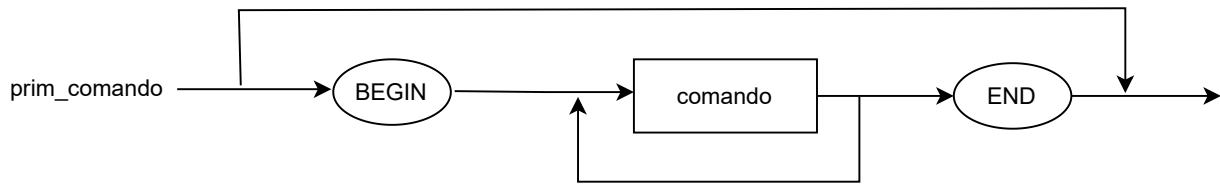


Figura 8: Grafo do não-terminal <prim\_comando>.

O nó Pai, <prim\_comando>, é capaz de gerar o termo  $\lambda$ , que é representado no grafo pela seta superior, que não apresenta símbolos terminais ou não-terminais. Vale ressaltar que a expressão apresenta um ciclo em <comando>: esse ciclo representa o não-terminal <mais\_cmd> da gramática original.

### 3.3.6 Grafo de Comando

A regra gramatical permite que o grafo <comando> gere diferentes símbolos terminais, como um identificador, CALL, BEGIN, IF ou WHILE - cada um desses terminais são seguidos de outros terminais ou não-terminais, conforme a figura 9:

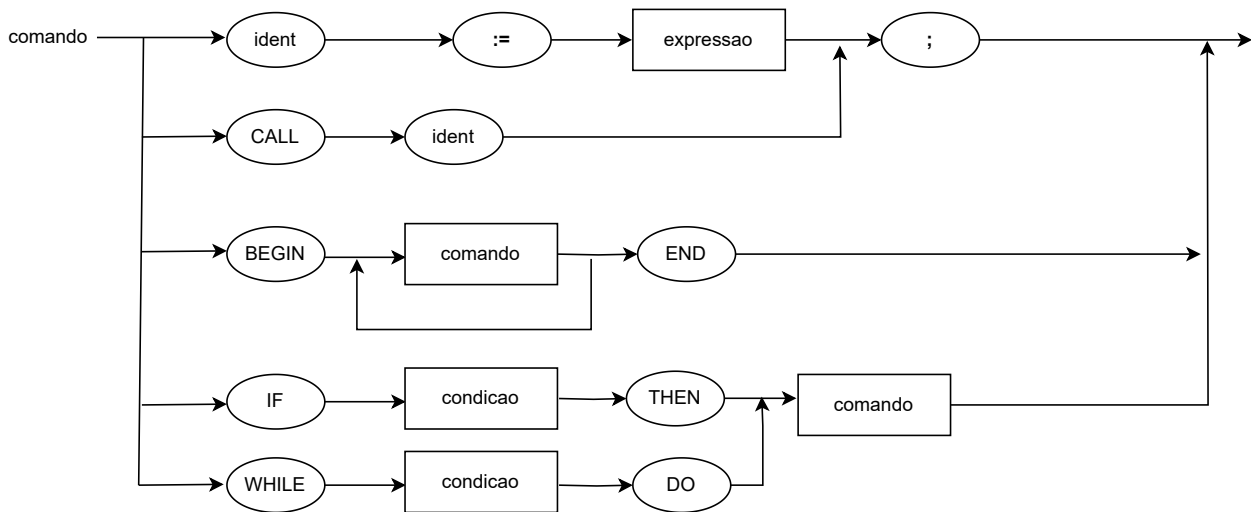


Figura 9: Grafo do não-terminal <comando>.

O nó Pai, <comando>, é capaz de gerar o termo  $\lambda$ , que é representado no grafo pela seta superior, que não apresenta símbolos terminais ou não-terminais. Vale ressaltar que a expressão do terminal BEGIN apresenta um ciclo em <comando>: esse ciclo representa o não-terminal <mais\_cmd> da gramática original.

### 3.3.7 Grafo de Expressão

A figura 10 representa o grafo de <expressao>:

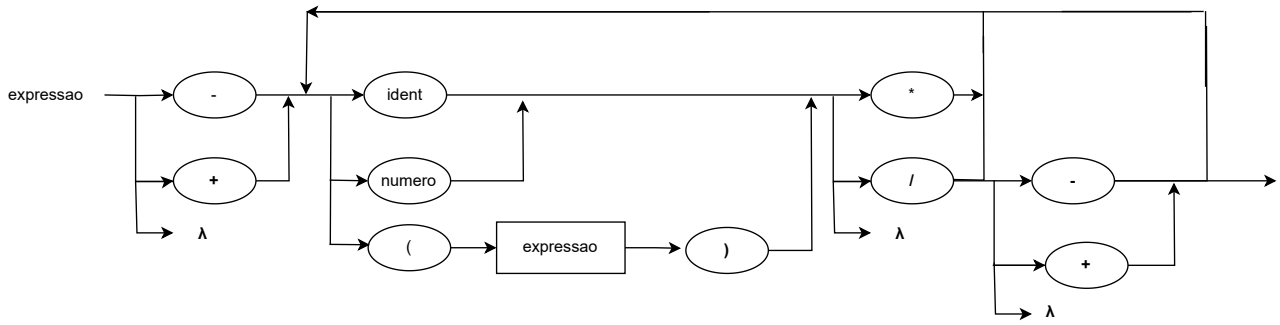


Figura 10: Grafo do não-terminal `<expressao>`.

Pela regra gramatical, [1], `<expressao>` gera outros 3 símbolos não-terminais em sequência: `<operador_unario>`, `<termo>` e `<mais_termos>`, respectivamente. Na figura 10 esses símbolos (não-terminais) foram expandidos, conforme a regra de cada um deles.

### 3.3.8 Grafo de Condição

O símbolo não-terminal `<condicao>` é capaz de gerar duas regras: terminal ODD seguido de não-terminal `<expressao>` ou não-terminais `<expressao>`, `<relacional>` e `<expressao>`, nessa ordem, respectivamente. A figura 11 mostra isso:

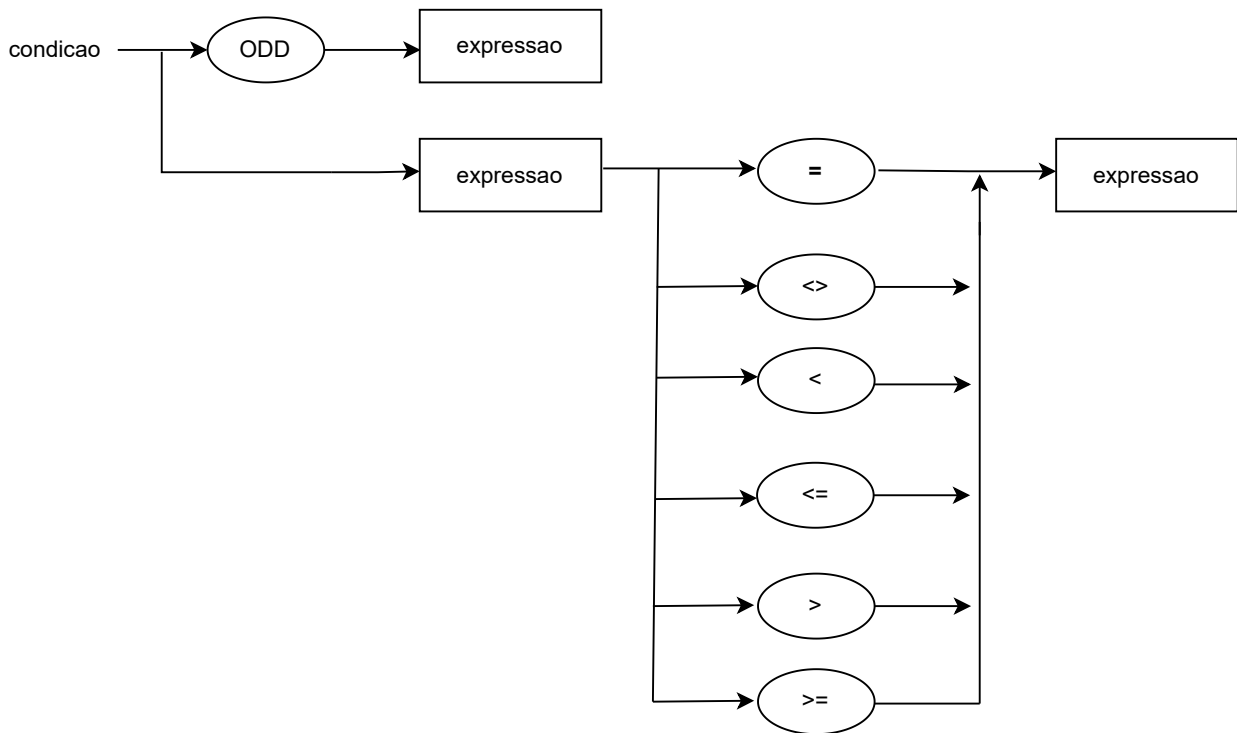


Figura 11: Grafo do não-terminal `<condicao>`.

A segunda regra, com não-terminais `<expressao>`, `<relacional>` e `<expressao>`, foi estendida: o símbolo `<relacional>` foi expandido nos terminais que ele gera, conforme a gramática. Os terminais são: `=`, `<>`, `<`, `<=`, `>` e `>=`.

## 4 Código

Do código, adicionou-se apenas um novo arquivo, `parser.c`, em relação ao Trabalho 1, no qual estão implementadas todas as rotinas relativas aos não-terminais da gramática. No tocante a isso, convém mencionar que foram excluídos alguns não-terminais como `<bloco>` e `<fatores>`, uma vez que eles poderiam ser substituídos pelas rotinas

que a chamam, a fim de otimizar o funcionamento do código.

Implementou-se também, nesse mesmo arquivo, a rotina de tratamento de erros, que analisa de forma hierarquizada se um símbolo de sincronização é encontrado e se ele pertence a um seguidor do símbolo faltante ou a um seguidor do Pai (regra geradora). Isso permite que tenha menos perdas de código e que cada execução possa extrair mais informações do código.

Fica inviável para um relatório expor todo o código e comentar parte por parte, portanto, reserva-se o direito de deixar a cargo do leitor acessar o código e ler os comentários nele descrevendo, detalhadamente, o funcionamento do código.

## 4.1 Compilação do Código

Sobre a compilação do código, mantém-se a mesma estrutura passada, bastando utilizar o comando `make`, uma vez que há um `Makefile` na pasta. Existem alguns arquivos pré-disponíveis para teste na pasta `test`. Para executar o código, basta digitar o comando.

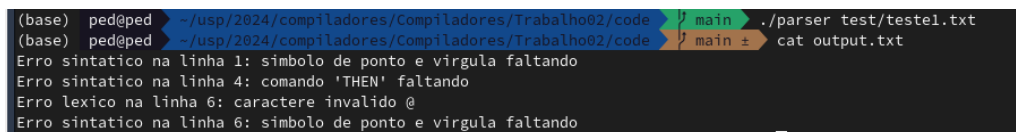
```
./parser <arquivo.txt>
```

## 5 Exemplo de Execução

Partindo para a exemplificação da execução do código, primeiramente, apresentar-se-á a execução do exemplo disponibilizado pelo docente, mostrado abaixo.

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

O resultado da execução é mostrado na figura 12. É possível ver que o resultado foi conforme o esperado pelo professor, com a exceção da alteração que foi feita e que passou a exigir a presença de ponto-e-vírgula em todas as atribuições.



```
(base) ped@ped ~ -/usp/2024/compiladores/Compiladores/Trabalho02/code main ./parser test/testel.txt
(base) ped@ped ~ -/usp/2024/compiladores/Compiladores/Trabalho02/code main cat output.txt
Erro sintatico na linha 1: simbolo de ponto e virgula faltando
Erro sintatico na linha 4: comando 'THEN' faltando
Erro lexico na linha 6: caractere invalido @
Erro sintatico na linha 6: simbolo de ponto e virgula faltando
```

Figura 12: Execução do exemplo disponibilizado pelo docente e seu resultado.

Já para um exemplo mais complexo, tem-se o caso do código abaixo.

```
1 CONST alfa,beta=5;
2
3 VAR a,b,c
4
5 {COMENTÁRIO COM ACENTUAÇÃO}
6 PROCEDURE meuproc;
7 VAR x,y
8 BEGIN
```

```

9      x:=5.5;
10     WHILE x>0
11         y:=y+alfa*4+beta;
12         x:=x-1
13 END;
14
15 BEGIN
16     IF claudinho <> (bochecha) THEN
17         cleitinho:=2*banana
18     CALL meuproc
19 END.

```

O resultado da análise desse código é mostrada na figura 13. É possível ver que o compilador é capaz de lidar com a maioria dos problemas de pequena ordem sem grandes complicações, identificando corretamente a maior parte dos erros. Contudo, desperta atenção o fato de o código não ter identificado a diretiva **THEN**, que se deve ao fato de “>” ser um seguidor da regra de fator, que origina a estrutura “( *expressao* )”. Preferiu-se manter o funcionamento dessa forma pois acusou-se a falta de um “)” e não haveria comprometimento da estrutura do código, além do que qualquer usuário com uma certa familiaridade com programação entenderia que ao não fechar os parênteses corretamente, ele atrapalharia o reconhecimento das cadeias subsequentes.

```

(base) ped@ped ~:/usp/2024/compiladores/Compiladores/Trabalho02/code main ± ./parser test/teste_a.txt
(base) ped@ped ~:/usp/2024/compiladores/Compiladores/Trabalho02/code main ± ls
a.out headers Makefile output.txt parser src test
(base) ped@ped ~:/usp/2024/compiladores/Compiladores/Trabalho02/code main ± cat output.txt
Erro sintatico na linha 1: simbolo de atribuicao faltando
Erro sintatico na linha 1: inteiro faltando em atribuicao
Erro sintatico na linha 3: simbolo de ponto e virgula faltando
Erro sintatico na linha 7: simbolo de ponto e virgula faltando
Erro lexico na linha 9: numeros reais 5.5 nao sao aceitos pela linguagem
Erro sintatico na linha 9: expressao esperada faltando (inteiro, identificador, '(expressao)')
Erro sintatico na linha 10: comando 'DO' esperado, mas encontrou y
Erro sintatico na linha 12: simbolo de ponto e virgula faltando
Erro sintatico na linha 16: ')' esperado mas encontrou >
Erro sintatico na linha 16: comando 'THEN' esperado mas encontrou >
Erro sintatico na linha 17: simbolo de ponto e virgula faltando
Erro sintatico na linha 18: simbolo de ponto e virgula faltando

```

Figura 13: Execução do código de teste dos alunos e seus resultados.

É interessante notar também que o código passou a ser capaz de tratar caracteres fora do padrão ASCII, bem como passou a identificar e tratar números reais.

## 6 Conclusão

A análise da linguagem PL/0 destaca a importância do analisador sintático durante a etapa de compilação. Ao longo do projeto, alterou-se sucintas regras gramaticais, a fim de eliminar certas ambiguidades e tornar a linguagem mais clara e eficiente.

Além disso, a modelagem dos grafos sintáticos, para os diferentes elementos da linguagem, como programa, constante, variável, procedimento, primeiro comando, comando, expressão e condição, demonstrou o cuidado e a precisão na representação da estrutura sintática da linguagem, para facilitar a compreensão e implementação dessa etapa do compilador. As decisões de projeto tomadas, como a definição dos primeiros e seguidores, as alterações na gramática e a modelagem dos grafos sintáticos, refletem o comprometimento em aprimorar o processo de compilação e garantir a correta interpretação dos programas escritos em PL/0.

Em suma, o trabalho evidencia a importância da análise sintática na construção de compiladores eficientes e confiáveis, destacando a relevância de decisões bem fundamentadas e de uma modelagem precisa para o sucesso do projeto.

## Referências

- [1] Linguagem PL/0. Disponível em: [https://edisciplinas.usp.br/pluginfile.php/8350382/mod\\_resource/content/0/Gram%C3%A1tica%20de%20PL0.pdf](https://edisciplinas.usp.br/pluginfile.php/8350382/mod_resource/content/0/Gram%C3%A1tica%20de%20PL0.pdf)
- [2] Trabalho 1 - Analisador Léxico. Disponível em: <https://github.com/carloshenriquehannas/Compiladores/tree/main/Trabalho01>