Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Engenharia de Computação - 2024.1

Trabalho 2: analisador léxico para a linguagem PL/0

SCC0605 - Teoria da Computação e Compiladores

Mateus Santos Messias - NºUSP: 12548000

Pedro Borges Gudin - NºUSP: 12547997

Resumo

Foi feita algumas mudanças no léxico conforme solicitado e alterações para melhorar seu desempenho. Foi criado um analisador sintático que utiliza o modo pânico para a linguagem PL/0, com o objetivo de exercitar o conteúdo aplicado nas aulas.

Introdução

Este trabalho descreve o desenvolvimento de um analisador sintático para a linguagem PL/0, como parte da disciplina SCC0605 da Universidade de São Paulo. Após a construção do analisador léxico, a gramática da linguagem PL/0 foi estudada para implementar as regras de produção e a estrutura sintática dos programas escritos nesta linguagem. Utilizando uma abordagem de análise descendente recursiva, foram definidas funções recursivas em C para cada regra da gramática, visando reconhecer a estrutura hierárquica do código fonte.

Durante o desenvolvimento, várias decisões de projeto foram tomadas, incluindo a escolha das estruturas de dados (árvore de derivação), o tratamento de erros sintáticos e a organização do código. O programa resultante lê um arquivo de texto contendo um programa em PL/0, utiliza o analisador léxico para identificar tokens, e aplica as regras sintáticas para validar a estrutura do programa. Erros sintáticos são identificados e reportados com mensagens de erro detalhadas. O relatório cobre desde a definição da gramática até a implementação e execução do analisador sintático, destacando as decisões de projeto e os resultados obtidos.

O Analisador Sintático

1. **Gramática da linguagem PL/0**

```
<programa> ::= <bloco> .
<bloco> ::= <declaracao> <comando>
<declaracao> ::= <constante> <variavel> <procedimento>
<constante> ::= CONST ident = numero <mais_const> ; | λ
<mais_const> ::= , ident = numero <mais_const> | λ
<variavel> ::= VAR ident <mais_var> ; | λ
<mais_var> ::= , ident <mais_var> | λ
<procedimento> ::= PROCEDURE ident ; <bloco> ; <procedimento> | λ
<comando> ::= ident := <expressao>
| CALL ident
| BEGIN <comando> <mais_cmd> END
| IF <condicao> THEN <comando>
| WHILE <condicao> DO <comando>
| λ
<mais_cmd> ::= ; <comando> <mais_cmd> | λ
<expressao> ::= <operador_unario> <termo> <mais_termos>
<operador_unario> ::= - | + | λ
<termo> ::= <fator> <mais_fatores>
<mais_termos> ::= - <termo> <mais_termos> | + <termo> <mais_termos> | λ
<fator> ::= ident | numero | ( <expressão> )
<mais_fatores> ::= * <fator> <mais_fatores> | / <fator> <mais_fatores>
| λ
<condicao> ::= ODD <expressao>
| <expressao> <relacional> <expressao>
<relacional> ::= = | <> | < | <= | > | >=
```

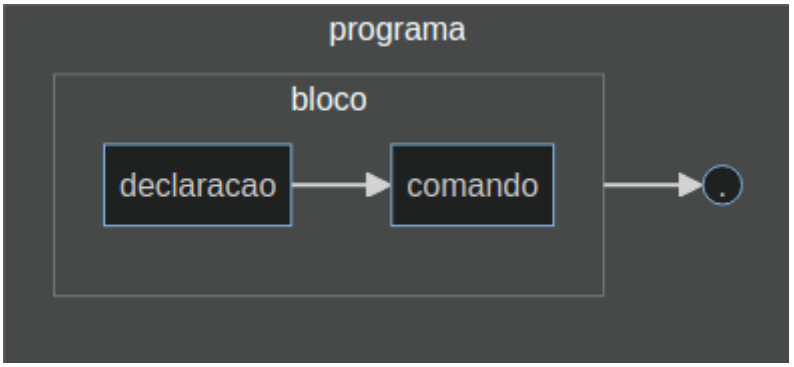
Além disso:

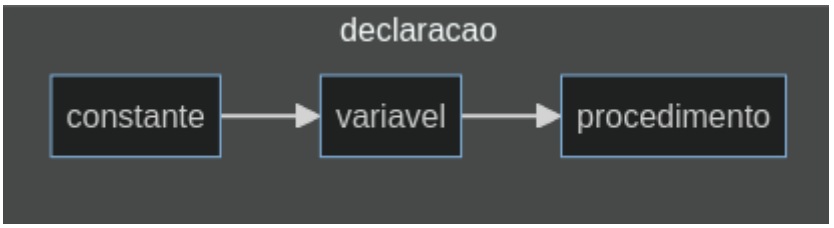
- comentários são de única linha, entre chaves { }
- identificadores são formados por letras e dígitos, começando por letra
- só há números inteiros, formados por um ou mais dígitos (entre 0 e 9)

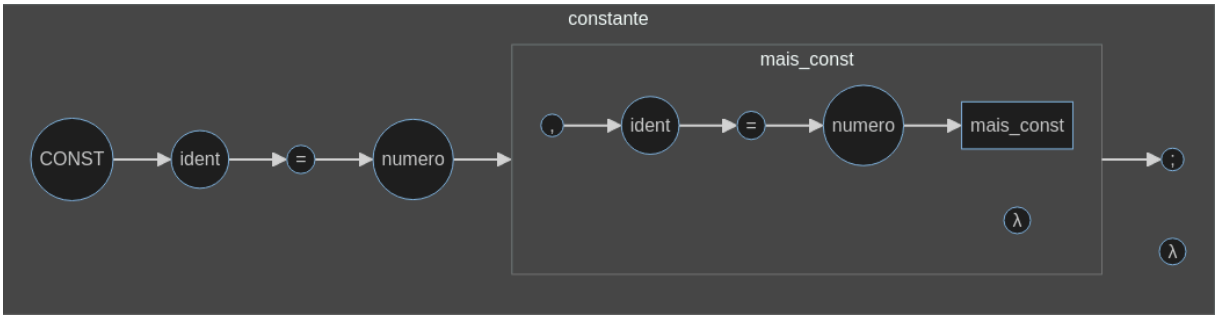
2. **Mudanças no Léxico**

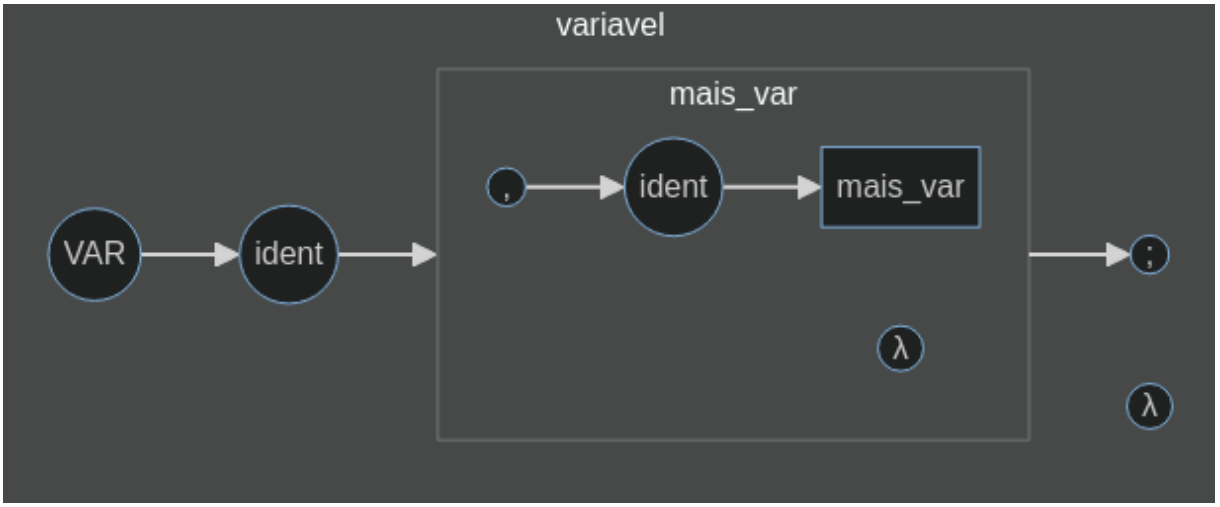
Mudou-se algumas coisas do analisador léxico dentre elas foi feito uma otimizacao deixando a tabela de transicao estatica o que melhorou consideravelmente a performance, além de adicionar a linha do token. É importante citar que o léxico nao retorna um token NULL como forma de marcar o fim do programa, agora ele retorna um token com os campos cadeia = "", tipo = final, line = line contendo a linha final do programa.

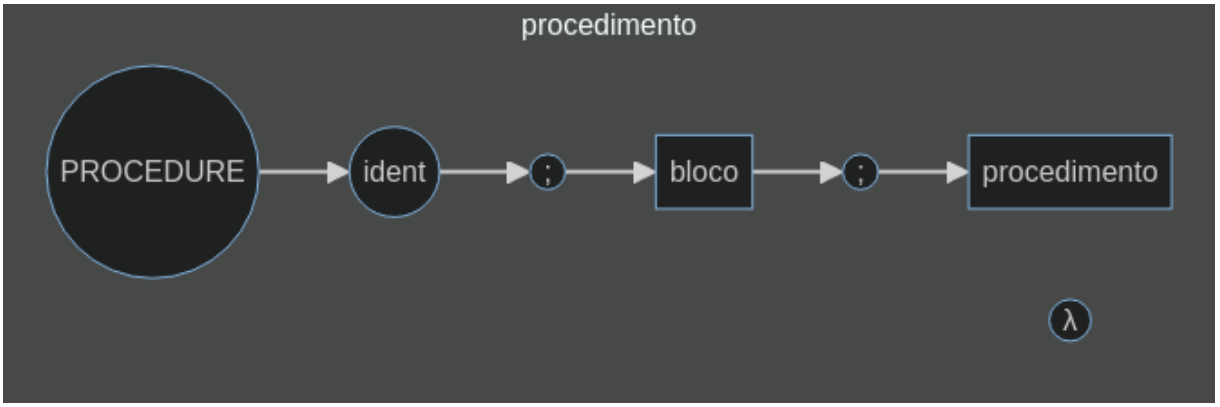
3. **Esquema da Gramática**

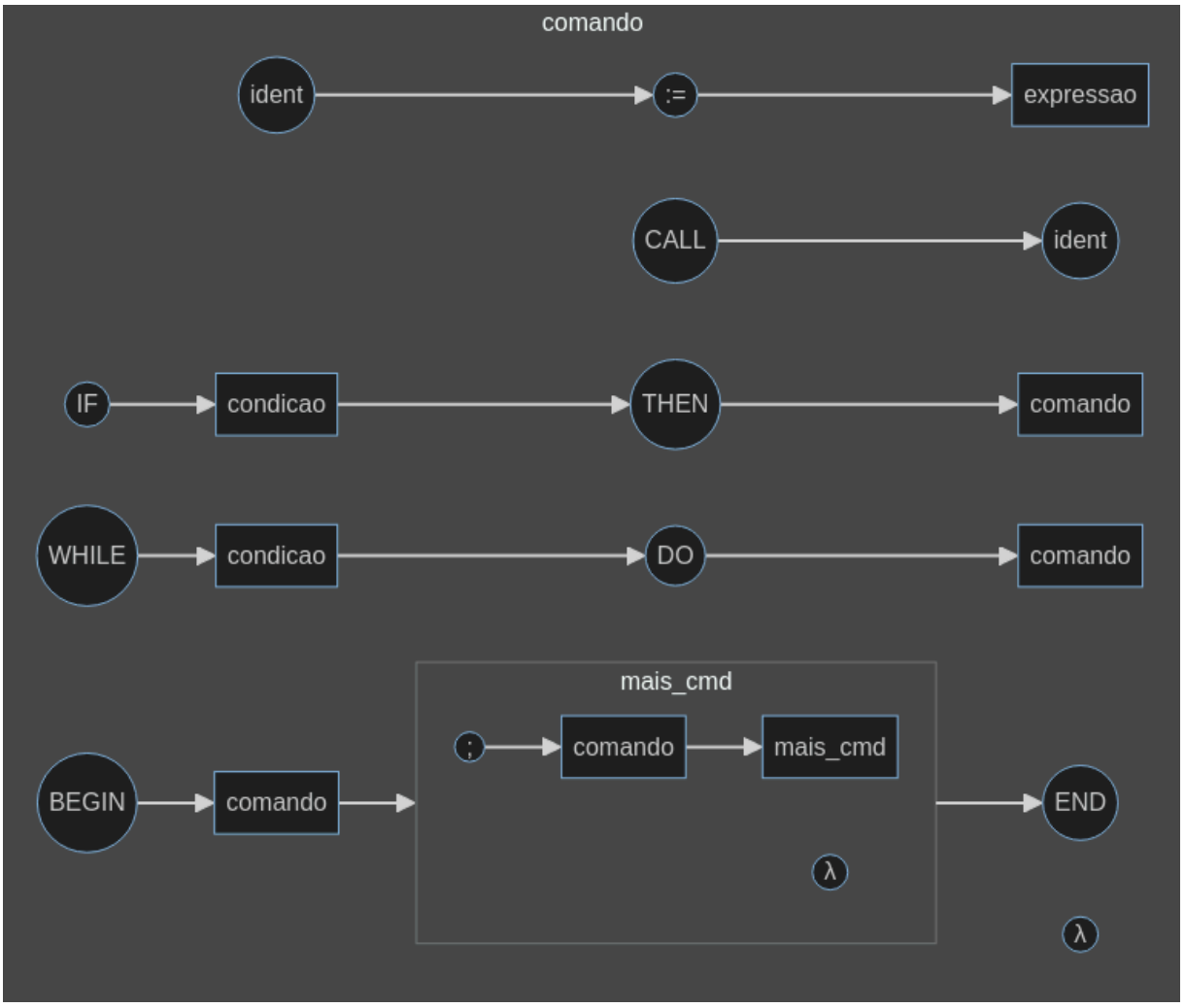


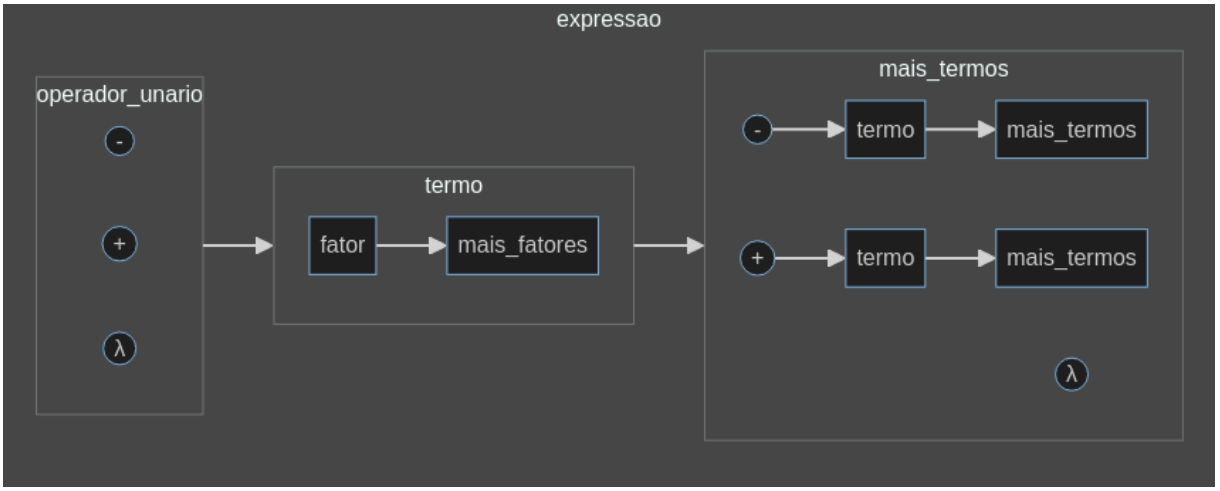


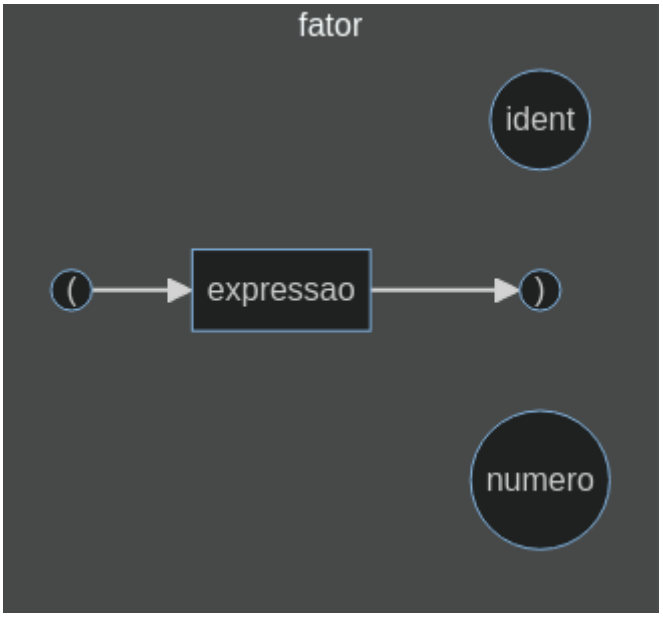


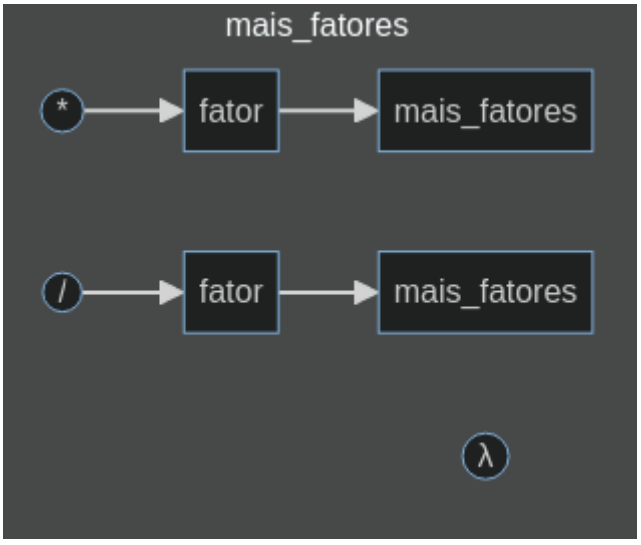


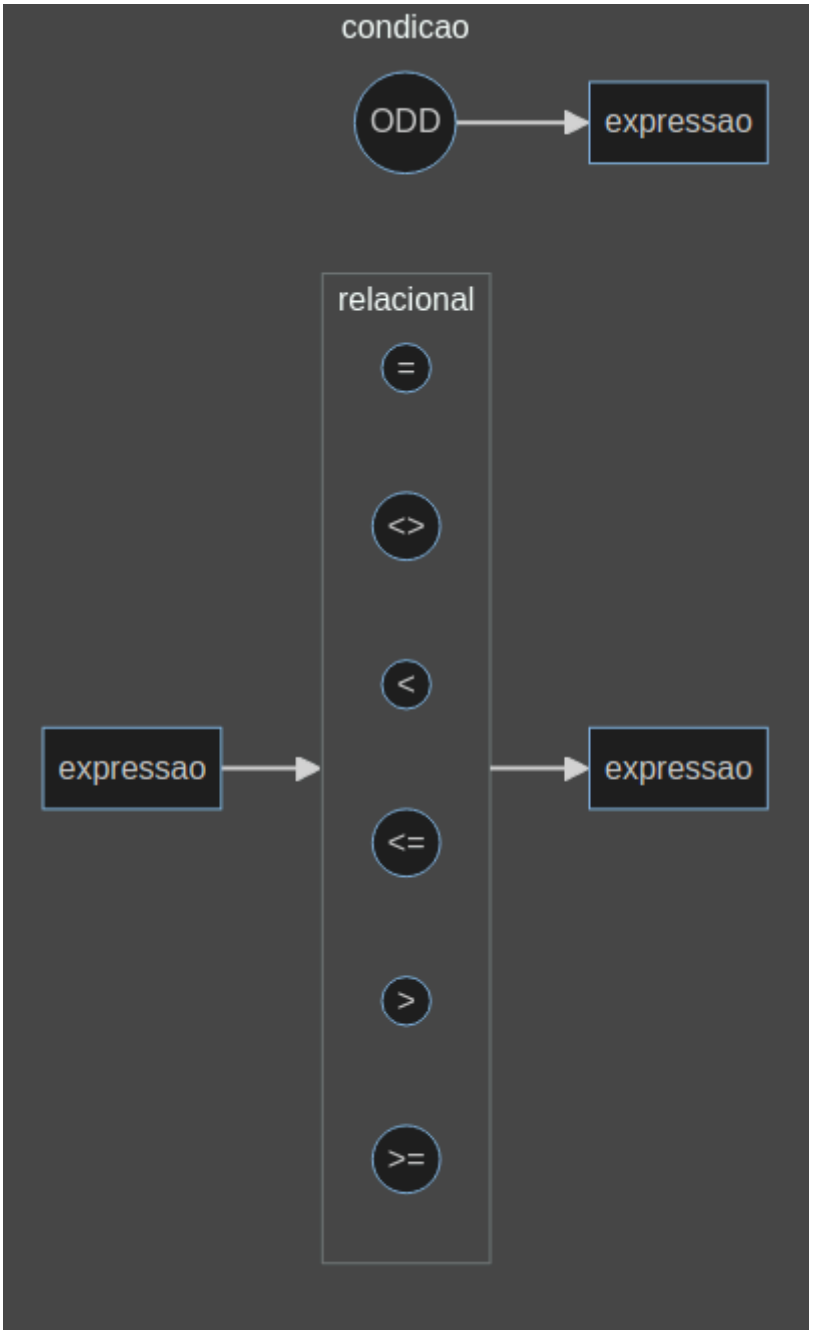

procedimento
PROCEDURE
ident
;
bloco
;
procedimento
λ


comando
ident
:=
expressao
CALL
ident
IF
condicao
THEN
comando
WHILE
condicao
DO
comando
mais_cmd
;
comando
mais_cmd
BEGIN
comando
END
λ
λ


expressao
operador_unario
-
+
λ
termo
fator
mais_fatores
mais_termos
-
termo
mais_termos
+
termo
mais_termos
λ


fator
ident
(
expressao
)
numero


mais_fatores
*
fator
mais_fatores
/
fator
mais_fatores
λ



2.2. **Tabela de Primeiro e Seguidor**

Não-Terminal	Primeiro	Seguidor
programa	{CONST, VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, "."}	{λ}
bloco	{CONST, VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, λ}	{".", ";"}
declaracao	{CONST, VAR, PROCEDURE, λ}	{ident, CALL, BEGIN, IF, WHILE, ".", ";"}

constante	{CONST, λ}	{VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ".", ";"}
mais_const	{",", λ}	{";"}
variavel	{VAR, λ}	{PROCEDURE, ident, CALL, BEGIN, IF, WHILE, ".", ";"}
mais_var	{",", λ}	{";"}
procedimento	{PROCEDURE, λ}	{ident, CALL, BEGIN, IF, WHILE, ".", ";"}
comando	{ident, CALL, BEGIN, IF, WHILE, λ}	{".", ";", END}
mais_cmd	{";", λ}	{END}
expressao	{-, +, ident, numero, "("}	{".", ";", END, THEN, DO, =, <>, <, <=, >, >=, ")"}
condicao	{ODD, -, +, ident, numero, "(")}	{THEN, DO}
operador_unario	{-, +, λ}	{ident, numero, "("}
termo	{ident, numero, "("}	{-, +, ".", ";", END, THEN, DO, =, <>, <, <=, >, >=, ")"}
mais_termos	{-, +, λ}	{".", ";", END, THEN, DO, =, <>, <, <=, >, >=, ")"}
fator	{ident, numero, "("}	{*, "/", -, +, ".", ";", END, THEN, DO, =, <>, <, <=, >, >=, ")"}
mais_fatores	{*, "/", λ}	{-, +, ".", ";", END, THEN, DO, =, <>, <, <=, >, >=, ")"}
relacional	{=, <>, <, <=, >, >=}	{-, +, ident, numero, "("}

Decisões de Projeto

Durante o desenvolvimento do analisador sintático, tomamos várias decisões de projeto para garantir a eficiência e a facilidade de manutenção do código:

1. **Forma que são coletados os erros**: Decidimos verificar os erros léxicos a cada novo token requisitado pelo sintático, para aí sim analisar os erros sintáticos.
2. **Erros sendo tratados**:
    a. **Falta de um terminal esperado:** Um bom exemplo seria a falta do terminal "CALL".
    b. **Final inesperado:** Algum token ou sua falta pode ter causado a busca de um token de sincronização que não foi encontrado ou o procedimento bloco teve um final inesperado.
    c. **Token inesperado:** Algum token ou sua falta causou um erro dentro de sua regra.
3. **Verificação de erros após o fim do programa**: Assim como consta na gramática PL/0, depois do não terminal bloco existe um terminal "." que indica o fim do programa, ou seja, não é permitida a existência de qualquer token após ele. Portanto, verifica-se a existência de erros, além de considerar um possível erro léxico de comentário não terminado, já que o programador pode ter esquecido de fechar um comentário, e este é ignorado pelos analisadores léxico e sintático.
4. **Automatização da Execução**: Utilizamos o utilitário `make` para automatizar a compilação e execução do programa.

Enfrentamos desafios, como garantir a correta implementação dos autômatos finitos e o tratamento eficiente de comentários e espaços em branco. Para superar esses desafios, realizamos testes extensivos com diversos programas escritos em PL/0 e ajustamos nossos autômatos conforme necessário.

Executando e Compilando o programa

1. **Requisitos de sistema**

- Sistema Operacional: Linux Ubuntu 22.04
- Compilador: GCC
- Ferramentas adicionais: make

2. **Como executar o programa**

Para compilar e executar o programa, siga os seguintes passos:

1. Navegue até o diretório onde o código-fonte está localizado.
2. Para compilar o programa digite `make` ou `make all` no terminal
3. Para rodar basta digitar `make run ARGS=<nome_do_programa_de_entrada.txt>`, ja existe um programa teste na pasta que utilizaremos, portanto o comando para rodar fica:
   `make run ARGS=teste.txt`.
4. O arquivo de entrada deve estar no formato de um programa PL/0.

2.1 **Saída do programa**

Ao executar o programa com um arquivo de entrada será gerado um arquivo de saída chamado `saida.txt` que contém o programa teste.

Exemplo de Entrada:

```
VAR a,b,a,c
BEGIN
  a:=2;
  IF a>2
      b:=3;
  c:=@+b
END.
```

Exemplo de saída:

```
Erro sintático na linha 2: ponto e virgula esperado
Erro léxico na linha 6: @
Erro sintático na linha 7: THEN esperado

Cada erro é uma chance de aprender. Não desista!
```