UNIVERSIDADE DE SÃO PAULO – USP
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO

# SCC0605 - Teoria da Computação e Compiladores

## Trabalho 2 - Análise Léxica e Sintática

**Docente responsável:** Prof. Dr. Thiago Alexandre Salgueiro Pardo

**Alunas:**
Amália Vitória de Melo — Nº USP: 13692417
Bárbara Fernandes Madera — Nº USP: 11915032
Letícia Crepaldi da Cunha — Nº USP: 11800879

São Carlos 2025

# Sumário

# 1 Introdução

Este relatório apresenta o desenvolvimento de um analisador léxico e sintático para a linguagem PL/0, realizado na disciplina SCC0605. O projeto foi dividido em duas etapas: implementação do analisador léxico, responsável pela identificação dos tokens, e do analisador sintático, que valida a estrutura dos programas segundo a gramática da linguagem.

O compilador desenvolvido reconhece tokens, detecta erros léxicos e sintáticos e estabelece uma boa base para fases futuras, como análise semântica.

# 2 Analisador Léxico

O analisador léxico segmenta o código-fonte em unidades chamadas tokens, como identificadores, operadores, números e palavras reservadas.

## 2.1 Tokens Reconhecidos

- **Identificadores**: letra seguida de letras e/ou dígitos. Verificados contra a tabela de palavras reservadas.
- **Números**: números inteiros.
- **Comentários**: delimitados por {...}. Comentários não fechados geram erro léxico.
- **Símbolos**: operadores (`:=`, `<>`, `<=`, `>=`) e delimitadores (`+`, `-`, `*`, `(`, `)`, `;`, `.`).

## 2.2 Erros Identificados e Melhorias

Foram observados erros no Trabalho 1, como:

- Falha no reconhecimento de comentários.
- Remoção incorreta de caracteres '1' e '0'.
- Inserções indesejadas de vírgulas na saída.
- Mensagens de erro pouco informativas.

## 2.3 Alterações no Analisador Léxico

Durante a evolução do projeto, foram realizadas alterações significativas no código do analisador léxico (`lexico.c`), resultando em uma versão aprimorada. As principais modificações foram focadas na melhoria tanto do tratamento de erros quanto melhoria do controle de fluxo da análise.

A mudança mais relevante foi a substituição de comandos `continue` por `return` nas situações em que ocorrem erros léxicos, como identificadores muito longos, números excedendo o tamanho permitido ou caracteres inválidos. Na versão anterior, o uso de `continue` fazia com que, ao encontrar um erro, o analisador simplesmente pulasse para o próximo ciclo do loop, sem gerar um token de erro adequado. Isso comprometia a comunicação com o analisador sintático, dificultando a recuperação de erros e a sincronização.

Com a utilização de `return`, o analisador retorna imediatamente um token do tipo `TOKEN_ERROR` sempre que identifica um erro léxico. Dessa forma, o analisador sintático pode ser devidamente informado do erro, possibilitando a aplicação correta das estratégias de recuperação, como o modo pânico.

Além disso, foram realizadas mudanças para otimizar a organização geral do código, tornando-o mais limpo e previsível. Essas mudanças tornaram o analisador léxico mais confiável, estável e alinhado com as necessidades do analisador sintático, reduzindo a incidência de falhas e melhorando o tratamento de erros na análise dos programas de entrada.

# 3 Implementação do Analisador Sintático

Nesta seção, apresentamos a implementação detalhada do analisador sintático desenvolvido para a linguagem PL/0. Este analisador foi construído utilizando a técnica de **descendente preditivo recursivo** e tem como principal objetivo validar a estrutura gramatical dos programas escritos nesta linguagem. Além disso, incorpora um mecanismo de **tratamento de erros por modo pânico**, garantindo a continuidade da análise mesmo quando ocorrem erros sintáticos.

## 3.1 Funcionamento Geral

O funcionamento do analisador baseia-se em uma função para cada regra da gramática PL/0
O fluxo principal é o seguinte:

- A análise é iniciada pela função `iniciarAnaliseSintatica()`, que configura os arquivos de entrada e saída e inicializa os ponteiros de controle.
- A função `programa()` representa o ponto de entrada da análise, correspondente à produção `<programa> ::= <bloco> .`.
- As demais funções, como `bloco()`, `declaracao()`, `comando()`, `expressao()`, entre outras, refletem diretamente as produções da gramática.

## 3.2 Variáveis Globais

Foram utilizadas algumas variáveis globais para facilitar o compartilhamento de informações entre as funções:

- `global_TextFile`: ponteiro para o arquivo de entrada.
- `global_TextSaida`: ponteiro para o arquivo de saída (log).
- `global_boolErro`: ponteiro para a variável que sinaliza a ocorrência de erros.
- `global_boolSpace`: controle para espaços (herdado do léxico).
- `panicMode`: flag que indica se o analisador está em modo de recuperação de erro.

## 3.3 Mecanismos de Controle e Tratamento de Erros

### 3.3.1 Função `match()`

Esta função tem como objetivo verificar se o token atual corresponde ao token esperado. Caso positivo, o token é consumido e a análise continua normalmente. Caso contrário, é gerada uma mensagem de erro sintático e o analisador entra em **modo pânico**, buscando tokens de sincronização para retomar a análise.

A função `match()` também aceita parâmetros adicionais que representam tokens de sincronização, além do token esperado no contexto.

### 3.3.2 Função `panic_mode_recover()`

Implementa o mecanismo de recuperação por modo pânico. Seu funcionamento consiste em consumir tokens até que um dos seguintes seja encontrado:

- O token esperado no contexto.
- Algum dos tokens passados como parâmetros.
- Tokens globais de sincronização: `BEGIN`, `CALL`, `IF`, `WHILE`, `END`, `EOF`.

Uma vez encontrado um ponto seguro, a análise prossegue normalmente.

## 3.4 Tratamento de Erros

O analisador implementa a seguinte estratégia de tratamento de erros:

- Ao detectar um erro, uma mensagem detalhada é registrada no arquivo de saída, indicando:
  - O token esperado.
  - O token encontrado.
- Utiliza o modo pânico para evitar a paralisação da análise, consumindo tokens até encontrar um ponto seguro de sincronização.
- Permite que múltiplos erros sejam identificados em uma única execução.

## 3.5 Saída Gerada

A análise gera vários arquivos `saidaX.txt` contendo:

- A lista dos tokens reconhecidos.
- Mensagens de erros léxicos e sintáticos.
- Mensagem sobre o sucesso ou falha da compilação.

Exemplo de saída de erro:

```
Erro sintático: Esperava 'THEN' apos a condicao do IF.
Esperava 'THEN', encontrou 'b' ('ident').
```

# 4 Compilação e Execução do Projeto

Para compilar e executar o compilador desenvolvido para a linguagem PL/0, utilizamos um ambiente Linux com suporte a compilador `gcc` e utilitários de terminal. Abaixo estão os passos detalhados:

## 4.1 Pré-requisitos

- Sistema operacional Linux (ou WSL no Windows);
- Compilador C (`gcc`) instalado;
- Terminal com suporte a `make`;

## 4.2 Comandos Utilizados

```
make all
make run
```

- `make all`: Compila todos os arquivos necessários, gerando o executável `compilador`.
- `make run`: Executa o programa `./compilador`, que lê o arquivo de entrada (geralmente `entrada.txt`) contendo o código-fonte em PL/0

## 4.3 Exemplo de Execução Manual (sem Makefile)

Caso deseje compilar e executar sem utilizar o `Makefile`, execute os seguintes comandos:

```
gcc main.c lexico.c sintatico.c -o compilador
./compilador
```

## 4.4 Arquivo de Entrada

O programa espera até 5 arquivos de teste por vez, chamados `testeX.txt` contendo o código em PL/0 a ser analisado. Cada um gera um arquivo de saída referente a entrada fornecido, `saidaX.txt`, com os resultados da análise.

Caso necessário, é possivel alteral a quantidade dos arquivos de saida na função main em main.c, modificando no for a quantidade de arquivos de entrada.

## 4.5 Exemplo de Execução

Dado o seguinte código no arquivo `entrada.txt`:

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

A execução gera a seguinte saída em `saida.txt`:

```
VAR, VAR
a, ident
,, simbolo_virgula
b, ident
,, simbolo_virgula
c, ident
BEGIN, BEGIN
Erro sintático: Esperava ';' apos a declaracao de variavel. Esperava
'simbolo_ponto_e_virgula', encontrou 'BEGIN' ('BEGIN').
a, ident
:=, simbolo_atribuicao
2, numero
;, simbolo_ponto_e_virgula
```

```
IF, IF
a, ident
>, simbolo_maior
2, numero
b, ident
Erro sintático: Esperava 'THEN' apos a condicao do IF. Esperava 'THEN',
encontrou 'b' ('ident').
:=, simbolo_atribuicao
3, numero
c, ident
:=, simbolo_atribuicao
@, <ERRO_LEXICO> (Caractere inválido)
+, simbolo_mais
b, ident
END, END
., simbolo_ponto
Compilacao falhou com erros sintaticos ou lexicos.
```

Este exemplo demonstra tanto o reconhecimento correto dos tokens quanto a detecção de erros léxicos (caractere inválido '@') e sintáticos (ausência do THEN).

# 5 Conclusão

O desenvolvimento deste compilador permitiu aplicar, na prática, os conceitos de teoria da computação e construção de compiladores. A etapa léxica consolidou o entendimento sobre a definição e reconhecimento de tokens, além do tratamento de erros léxicos.

Na etapa sintática, a utilização da técnica de descida recursiva possibilitou traduzir a gramática formal diretamente em código. O mecanismo de recuperação por modo pânico permitiu a identificação de múltiplos erros em uma única execução.

Ambas as etapas — léxica e sintática — são essenciais para assegurar que o código-fonte esteja corretamente estruturado antes de avançar para fases posteriores, como análise semântica ou geração de código. A integração dessas fases resultou em um compilador funcional, capaz de fornecer diagnósticos claros e precisos sobre a estrutura dos programas em PL/0.