Instituto de Ciências Matemáticas e de Computação

# SCC0605 - Teoria da Computação e Compiladores

## Relatório Trabalho 2


**Alunos:**

| | |
|---|---|
| Leonardo Gueno Risseto | 13676482 |
| Lucas Lima Romero | 13676325 |
| Luciano Gonçalves Lopes Filho | 13676520 |
| Marco Antonio Gaspar Garcia | 11833581 |
| Thiago Kashivagui Gonçalves | 13676579 |

**Docente:**
Prof. Thiago A. S. Pardo


São Carlos, 2025

# Sumário

# 1 Introdução

Este relatório descreve o desenvolvimento de um analisador sintático para a linguagem de programação PL/0, como parte do Trabalho 2 da disciplina de Teoria da Computação e Compiladores (SCC0605). O projeto, implementado na linguagem C, visa integrar-se ao analisador léxico desenvolvido no trabalho anterior, culminando em um sistema capaz de validar a estrutura de um programa PL/0 de acordo com sua gramática formal.

Os principais objetivos do trabalho foram:

1. Implementar um **analisador sintático descendente preditivo recursivo**, onde cada não-terminal da gramática é representado por uma função.

2. Desenvolver um mecanismo robusto de **tratamento de erros sintáticos pelo modo pânico**, permitindo que a análise continue após a detecção de um erro para encontrar múltiplos problemas em uma única execução.

3. Consolidar a integração entre os analisadores léxico e sintático, garantindo que a saída do primeiro alimente corretamente o segundo.

4. Produzir um relatório de saída claro, listando todos os erros léxicos e sintáticos encontrados ou uma mensagem de sucesso caso o programa esteja correto.

O sistema resultante é uma ferramenta funcional para a validação de código PL/0, com ênfase na modularidade do código, na usabilidade e na clareza das mensagens de erro.

# 2 Correções realizadas a partir do Trabalho 1

- **Detecção de erro em comentários:**
  - Corrigido o analisador léxico para detectar corretamente erros em comentários, conforme a especificação da linguagem PL/0, que permite apenas comentários de uma linha. Antes, o analisador permitia comentários em múltiplas linhas, o que não estava de acordo com a gramática. Agora, qualquer comentário não fechado corretamente na mesma linha é identificado como erro léxico.
- **Detecção de números mal formados:**
  - Corrigido um problema na detecção de números mal formados no analisador léxico, onde um caractere a mais era pulado após a identificação do erro. Agora, o analisador consome corretamente apenas os caracteres pertencentes ao número mal formado, garantindo precisão na análise e na mensagem de erro.

# 3 Decisões de Projeto

A implementação do analisador seguiu as diretrizes da especificação, resultando em um sistema modular e eficiente. As principais decisões de projeto são detalhadas a seguir.

## 3.1 Estrutura e Modularidade

O código-fonte foi organizado em módulos distintos para promover a separação de responsabilidades e facilitar a manutenção:

- `hash_table.*`: Implementa tabelas hash para o armazenamento e busca eficiente de palavras reservadas e símbolos da linguagem. Esta abordagem evita buscas lineares e otimiza o desempenho do analisador léxico.
- `lexico.*`: Responsável pela análise léxica. Transforma o código-fonte em uma sequência de tokens, identificando identificadores, números, palavras-chave e erros léxicos.
- `sintatico.*`: Contém a implementação do analisador sintático descendente recursivo e o tratamento de erros. É o coração do trabalho atual.
- `main.c`: Ponto de entrada do programa que orquestra a inicialização dos módulos, o gerenciamento de arquivos de entrada/saída e invoca o processo de análise.

Essa estrutura modular, conectada por meio de arquivos de cabeçalho (‘.h‘), garante um baixo acoplamento entre os componentes.

## 3.2 Analisador Sintático Descendente Preditivo Recursivo

Conforme especificado, foi implementado um parser descendente preditivo recursivo. Para cada produção da gramática da PL/0, uma função em C foi criada. A estrutura de cada regra foi previamente modelada através de grafos sintáticos, que serviram como um projeto visual para o código.

Os grafos completos estão disponíveis para consulta online em um arquivo Figma, acessível através deste link: https://www.figma.com/design/8k7DGFb1dH1s7qjPcApAil/Analisador-Sintático?node-id=0-1

Dessa forma, a lógica de todo o analisador sintático reflete a estrutura dos grafos, com cada função de parsing implementando as decisões e os caminhos de seu diagrama correspondente.

Para cada produção da gramática da PL/0, uma função em C foi criada. Por exemplo:

- A regra ‘programa ::= bloco.‘ é tratada pela função ‘void programa()‘.
- A regra ‘bloco ::= [declaracao] comando‘ é tratada pela função ‘void bloco()‘.
- E assim por diante para ‘declaracao()‘, ‘comando()‘, ‘expressao()‘, etc.

A análise é guiada por um token de *lookahead*, que representa o próximo token na entrada. As funções utilizam o tipo deste token para decidir qual regra de produção aplicar, consumindo-o com a função ‘advance()‘ ao fazer um casamento (match) bem-sucedido.

## 3.3 Tratamento de Erros pelo Modo Pânico

Uma das principais funcionalidades do analisador é seu robusto tratamento de erros, implementado através do **modo pânico**. Esta estratégia permite que o parser se recupere de um erro e continue a análise, possibilitando a detecção de múltiplos erros em uma única compilação.

1. **Detecção:** Quando uma função do parser encontra um token inesperado (um *mismatch*), ela invoca a função 'void erro(const char *msg)', que registra uma mensagem detalhada no arquivo de saída, incluindo a linha do erro e o token encontrado.

2. **Recuperação:** Após registrar o erro, a função 'void sincroniza(TokenTipo sincronizadores[], int tamanho)' é chamada. Esta função descarta tokens da entrada até encontrar um que pertença ao conjunto de *sincronização* (geralmente o conjunto FOLLOW do não-terminal atual). Isso permite que o parser retome a análise a partir de um ponto seguro.

Cada função de parsing principal (como 'constante()', 'variavel()', etc.) define seu próprio conjunto de tokens de sincronização, tornando a recuperação de erros específica para cada contexto da gramática.

### 3.3.1 Conjuntos de Sincronização (First e Follow)

A eficácia do modo pânico depende da escolha correta dos conjuntos de sincronização para cada regra da gramática. Estes conjuntos são derivados diretamente dos conjuntos **First** (Primeiro) e **Follow** (Seguidor) dos não-terminais da linguagem PL/0. O conjunto de sincronização de um não-terminal $A$ é tipicamente o seu conjunto $Follow(A)$, garantindo que o analisador possa saltar para o próximo token que pode legalmente seguir a construção que falhou.

A seguir, apresentamos os conjuntos calculados que guiaram a implementação dos vetores `sincronizadores[]` em cada função do parser.

Primeiros

ocorre qnd $p(\langle declaração \rangle) = \lambda$

$p(\langle programa \rangle) = p(\langle bloco \rangle) = p(\langle declarações \rangle) \cup p(\langle comando \rangle) = \{$ CONST, VAR, PROCEDURE, ident, CALL, BEGIN, IF, WHILE, $\lambda\}$

$p(\langle bloco \rangle) = p(\langle declarações \rangle) \cup p(\langle comando \rangle) =$ → mesmo que acima

$p(\langle declaracao \rangle) = p(\langle constante \rangle) = \{$ CONST, VAR, PROCEDURE, $\lambda\}$ → mesmo que acima

$p(\langle constante \rangle) = \{$ CONST, $\lambda\}$

$p(\langle mais\_const \rangle) = \{$ , , $\lambda\}$

$p(\langle variavel \rangle) = \{$ VAR, $\lambda\}$

$p(\langle mais\_var \rangle) = \{$ , , $\lambda\}$

$p(\langle procedimento \rangle) = \{$ PROCEDURE, $\lambda\}$

$p(\langle comando \rangle) = \{$ ident, CALL, BEGIN, IF, WHILE, $\lambda\}$

$p(\langle mais\_cmd \rangle) = \{$ ; , $\lambda\}$

se $p(\langle op\_unario \rangle) = \lambda$

✓ i) $p(\langle expressao \rangle) = p(\langle operador\_unario \rangle) = \{-, +,$ ident, numero, $(\}$

$p(\langle operador\_unario \rangle) = \{-, +, \lambda\}$

$p(\langle termo \rangle) = p(\langle fator \rangle) = \{$ ident, numero, $(\}$

$p(\langle mais\_termos \rangle) = \{-, +, \lambda\}$

$p(\langle fator \rangle) = \{$ ident, numero, $(\}$

$p(\langle mais\_fatores \rangle) = \{*, /, \lambda\}$

$p(\langle condicao \rangle) = \{$ ODD $\} \cup p(\langle expressão \rangle) = \{$ ODD, -, +, ident, numero, $(\}$

$p(\langle relacional \rangle) = \{=, <>, <, <=, >, >=\}$

i) Se $p(\langle op\_unario \rangle) = \lambda \Rightarrow p(\langle expressao \rangle) = p(\langle termo \rangle) = \{$ ident, numero, $(\}$

Calcule o seguidor para os não terminais da PLO

$s_2(\langle programa \rangle) = \{\lambda\}$

$s(\langle bloco \rangle) = \{.\} \cup \{;\} = \{., ;\}$

$s(\langle declaração \rangle) = p(\langle comando \rangle) = \{ident, CALL, BEGIN, IF, WHILE, \lambda\}$

$s(\langle constante \rangle) = p(\langle variável \rangle) \cup p(\langle procedimento \rangle) \cup p(declaracoes)$
$= (VAR, PROCEDURE$

$s(\langle mais-const \rangle) = s(\langle constante \rangle) \cup s(\langle mais-const \rangle) = \{;\}$

$s(\langle variável \rangle) = p(\langle procedimento \rangle) = \{PROCEDURE, \lambda\}$

$s(\langle mais\_var \rangle) = \{;\}$

$s(\langle procedimento \rangle) = s(\langle declaracao \rangle) \cup s(\langle procedimento \rangle) = \{ident, CALL, BEGIN, IF, WHILE, \lambda\}$

$s(\langle comando \rangle) = s(\langle bloco \rangle) \cup p(\langle mais-cmd \rangle) = \{., ;, ;, \lambda\}$
$\{.,;\}$ $\{;,\lambda\}$

$s(\langle mais-cmd \rangle) = \{END\} \cup s(\langle mais-cmd \rangle) = \{END\}$

$s(\langle expressao \rangle) = s(\langle comando \rangle) \cup \{)\} \cup s(\langle condicao \rangle) \cup p(\langle relacional \rangle)$
$= \{., ;, ;, \lambda, ), THEN, DO, =, <, >\}$

$s(\langle operador\_unario \rangle) = p(\langle termo \rangle) = \{ident, numero, (\}$
$s(\langle termo \rangle) = p(\langle mais\_termos \rangle) = \{-, +, \lambda\}$
$s(\langle mais\_termos \rangle) = s(\langle expressao \rangle) \cup s(\langle mais-termos \rangle) = \{., ;, ;, \lambda, ), THEN, DO, =, <, >\}$
$s(\langle fator \rangle) = p(\langle mais-fatores \rangle) = \{*, /, \lambda\}$
$s(\langle mais-fatores \rangle) = s(\langle termo \rangle) \cup s(\langle mais-fatores \rangle) = \{-, +, \lambda\}$
$s(\langle condicao \rangle) = \{THEN, DO\}$
$s(\langle relacional \rangle) = p(\langle expressão \rangle) = \{-, +, ident, numero, (\}$

### 3.4 Integração Léxico-Sintático

A comunicação entre o analisador léxico e o sintático é feita pela função `void advance()`. Esta função, ao ser chamada pelo parser, solicita o próximo token ao analisador léxico através de `obter_token()`.

Uma decisão importante foi centralizar o tratamento de erros léxicos dentro de `advance()`. Se `obter_token()` retorna um `TOKEN_ERRO`, a função `advance()` se encarrega de:

1. Imprimir a mensagem de erro léxico no arquivo de saída.
2. Tentar uma recuperação simples (ex: tratar um "Número Mal Formado"como `TOKEN_NUMERO`) para permitir que a análise sintática prossiga.
3. Solicitar o próximo token até que um válido seja encontrado, continuando o loop.

Essa abordagem desacopla o parser dos detalhes dos erros léxicos, permitindo que ele se concentre apenas na estrutura da linguagem.

## 4 Instruções para Compilar e Executar

### 4.1 Requisitos

- Um compilador C (o projeto foi desenvolvido e testado com 'gcc').
- Utilitário 'make'.
- Python 3 e a biblioteca 'tkinter' para a interface gráfica (opcional, mas recomendado).

O projeto foi testado em ambiente Linux, mas deve funcionar em outros sistemas operacionais com as ferramentas mencionadas.

### 4.2 Execução (Modo Recomendado: Interface Gráfica)

A maneira mais simples de usar o analisador é através da interface gráfica em Python.

1. Navegue até a raiz do projeto (a pasta que contém `interface_tester.py`).
2. Execute o seguinte comando no terminal:

```
python3 interface_tester.py
```

3. Uma janela será aberta. Cole ou escreva seu código PL/0 na área de texto superior.
4. Clique no botão **Analisar**. A interface irá automaticamente compilar e executar o analisador.
5. O resultado (erros ou mensagem de sucesso) será exibido na área de texto inferior. As linhas com erro no código-fonte serão destacadas em vermelho.

### 4.3 Execução (Modo Manual: Linha de Comando)

Para compilar e executar o projeto manualmente:

1. Crie um arquivo de entrada com seu código PL/0 em `Código/input/codigo.pl0`.
2. Abra um terminal e navegue até o diretório `Código/`.
3. Execute o comando:

```
make run
```

4. O comando irá compilar os arquivos-fonte (gerando um executável 'main') e executá-lo.
5. A saída do analisador será gerada no arquivo `Código/output/saida_sintatico.txt`.

Para limpar os arquivos gerados pela compilação, use `make clean`.

## 5 Exemplo de Execução

### Exemplo sem erro:

Código PL/0

```
CONST a = 10, b = 20;
VAR x, y;
BEGIN
   x := a + b;
   y := x * 2
END.
```

Saída

```
Nenhum erro encontrado.
```

### Exemplo com erro:

Código PL/0

```
VAR n, fat;
BEGIN
        n:=4;
        fat:=1;
        WHILE n > 1 DO
                BEGIN
                fat:=fat n;
                n:=n-1;
                END
END.
```

Saída

```
Erro sintático na linha 7: Esperado operador aritmético (+, -, * ou /). Token atual: n
```

# 6 Conclusão

O desenvolvimento do Trabalho 2 permitiu a aplicação prática de conceitos fundamentais da construção de compiladores, como a análise sintática e o tratamento de erros. A implementação de um analisador descendente preditivo recursivo para a linguagem PL/0, integrado a um analisador léxico e equipado com um robusto mecanismo de recuperação de erros em modo pânico, foi concluída com sucesso.

O sistema atende a todos os requisitos da especificação, oferecendo uma análise detalhada do código-fonte e reportando múltiplos erros de forma clara e precisa. A modularidade do projeto e a interface de usuário amigável são pontos fortes que facilitam o uso e a manutenção.

Este trabalho consolida uma base sólida para as próximas etapas do desenvolvimento de um compilador, como a análise semântica e a geração de código.