



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	2
2	Correções realizadas a partir do Trabalho 1	2
3	Decisões de Projeto	2
3.1	Estrutura e Modularidade	3
3.2	Analizador Sintático Descendente Preditivo Recursivo	3
3.3	Tratamento de Erros pelo Modo Pânico	3
3.3.1	Conjuntos de Sincronização (First e Follow)	4
3.4	Integração Léxico-Sintático	7
4	Instruções para Compilar e Executar	7
4.1	Requisitos	7
4.2	Execução (Modo Recomendado: Interface Gráfica)	7
4.3	Execução (Modo Manual: Linha de Comando)	8
5	Exemplo de Execução	8
6	Conclusão	9

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. 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

ocone qnd $p(\langle \text{declarac\~ao} \rangle) = \lambda$

$\varphi(\langle \text{programa} \rangle) = \varphi(\langle \text{bloco} \rangle) = p(\langle \text{declarac\~ao} \rangle) \cup p(\langle \text{comando} \rangle) = \{ \text{CONST},$
 $p(\langle \text{bloco} \rangle) = p(\langle \text{declarac\~ao} \rangle) \cup p(\langle \text{comando} \rangle) = \xrightarrow{\quad} \text{VAR, PROCEDURE,}$
 $p(\langle \text{declarac\~ao} \rangle) = p(\langle \text{constante} \rangle) = \{ \text{CONST, VAR, PROCEDURE, } \lambda \}$
 $p(\langle \text{constante} \rangle) = \{ \text{CONST, } \lambda \}$
 $p(\langle \text{mais-const} \rangle) = \{ , \lambda \}$
 $p(\langle \text{variavel} \rangle) = \{ \text{VAR, } \lambda \}$
 $p(\langle \text{mais-var} \rangle) = \{ , \lambda \}$
 $p(\langle \text{procedimento} \rangle) = \{ \text{PROCEDURE, } \lambda \}$
 $p(\langle \text{comando} \rangle) = \{ \text{ident, CALL, BEGIN, IF, WHILE, } \lambda \}$
 $p(\langle \text{mais-cmd} \rangle) = \{ , \lambda \}$
 $\checkmark \text{ i) } \varphi(\langle \text{express\~ao} \rangle) = p(\langle \text{operador-unario} \rangle) = \{ -, +, \text{ident, numero, } (\}$
 $p(\langle \text{operador-unario} \rangle) = \{ -, +, \lambda \}$
 $p(\langle \text{termo} \rangle) = p(\langle \text{fator} \rangle) = \{ \text{ident, numero, } (\}$
 $p(\langle \text{mais-termos} \rangle) = \{ -, +, \lambda \}$
 $p(\langle \text{fator} \rangle) = \{ \text{ident, numero, } (\}$
 $p(\langle \text{mais-fatores} \rangle) = \{ *, /, \lambda \}$
 $p(\langle \text{condicao} \rangle) = \{ \text{ODD} \} \cup p(\langle \text{express\~ao} \rangle) = \{ \text{ODD, -, +, ident, numero, } (\}$
 $p(\langle \text{relacional} \rangle) = \{ =, <, >, \leq, \geq, \}$

i) Se $\varphi(\langle \text{op-unario} \rangle) = \lambda \Rightarrow \varphi(\langle \text{express\~ao} \rangle) = p(\langle \text{termo} \rangle) = \{ \text{ident, numero, } (\}$

Calcule o seguidor para os não terminais do PLO

$$\mathcal{S}(\langle \text{programa} \rangle) = \{ \lambda \}$$

$$\mathcal{S}(\langle \text{bloco} \rangle) = \{ \cdot \} \cup \{ ; \} = \{ \cdot, ; \}$$

$$\mathcal{S}(\langle \text{declarações} \rangle) = \mathcal{P}(\langle \text{comando} \rangle) = \{ \text{ident}, \text{CALL}, \text{BEGIN}, \text{IF}, \text{WHILE}, \lambda \}$$

$$\begin{aligned} \mathcal{S}(\langle \text{constante} \rangle) &= \mathcal{P}(\langle \text{variável} \rangle) \cup \mathcal{P}(\langle \text{procedimento} \rangle) \cup \mathcal{P}(\langle \text{declaração} \rangle) \\ &= \{ \text{VAR}, \text{PROCEDURE} \} \end{aligned}$$

$$\mathcal{S}(\langle \text{mais-const} \rangle) = \mathcal{S}(\langle \text{constante} \rangle) \cup \mathcal{S}(\langle \text{mais-const} \rangle) = \{ ; \}$$

$$\mathcal{S}(\langle \text{variável} \rangle) = \mathcal{P}(\langle \text{procedimento} \rangle) = \{ \text{PROCEDURE}, \lambda \}$$

$$\mathcal{S}(\langle \text{mais-var} \rangle) = \{ ; \}$$

$$\mathcal{S}(\langle \text{procedimento} \rangle) = \mathcal{S}(\langle \text{declarações} \rangle) \cup \mathcal{S}(\langle \text{procedimento} \rangle) = \{ \text{ident}, \text{CALL}, \text{BEGIN}, \text{IF}, \text{WHILE}, \lambda, \cdot, ; \}$$

$$\mathcal{S}(\langle \text{comando} \rangle) = \mathcal{S}(\langle \text{bloco} \rangle) \cup \mathcal{P}(\langle \text{mais-cmd} \rangle) = \{ \cdot, ;, \cdot, \lambda \}$$

$$\mathcal{S}(\langle \text{mais-cmd} \rangle) = \{ \text{END} \} \cup \mathcal{S}(\langle \text{mais-cmd} \rangle) = \{ \text{END} \}$$

$$\begin{aligned} \mathcal{S}(\langle \text{expressão} \rangle) &= \mathcal{S}(\langle \text{comando} \rangle) \cup \{ \cdot \} \cup \mathcal{S}(\langle \text{condições} \rangle) \cup \mathcal{P}(\langle \text{relacional} \rangle) \\ &= \{ \cdot, ;, \cdot, \lambda, \cdot, \text{THEN}, \text{DO}, =, <, > \} \end{aligned}$$

$$\mathcal{S}(\langle \text{operador unário} \rangle) = \mathcal{P}(\langle \text{termo} \rangle) = \{ \text{ident}, \text{numero}, (\}$$

$$\mathcal{S}(\langle \text{termo} \rangle) = \mathcal{P}(\langle \text{mais-termos} \rangle) = \{ -, +, \lambda \}$$

$$\mathcal{S}(\langle \text{mais-termos} \rangle) = \mathcal{S}(\langle \text{expressão} \rangle) \cup \mathcal{S}(\langle \text{mais-termos} \rangle) = \{ \cdot, ;, \cdot, \lambda, \cdot, \text{THEN}, \text{DO}, =, <, > \}$$

$$\mathcal{S}(\langle \text{fator} \rangle) = \mathcal{P}(\langle \text{mais-fatores} \rangle) = \{ *, /, \lambda \}$$

$$\mathcal{S}(\langle \text{mais-fatores} \rangle) = \mathcal{S}(\langle \text{termo} \rangle) \cup \mathcal{S}(\langle \text{mais-fatores} \rangle) = \{ -, +, \lambda \}$$

$$\mathcal{S}(\langle \text{condições} \rangle) = \{ \text{THEN}, \text{DO} \}$$

$$\mathcal{S}(\langle \text{relacional} \rangle) = \mathcal{P}(\langle \text{expressão} \rangle) = \{ -, +, \text{ident}, \text{numero}, (\}$$

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.