



**Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação**

**SCC0217 - Linguagens de Programação e
Compiladores**

Docente: Diego Raphael Amancio
Estagiário PAE: Davi Alves Bezerra

ANALISADOR LÉXICO

Guilherme de Abreu Barreto, 12543033
Laura Fernandes Camargos, 13692334
Sandy da Costa Dutra, 12544570

São Carlos, June 24, 2025

1 Introdução

Este relatório descreve o desenvolvimento de um analisador léxico para a linguagem LALG. O objetivo principal é transformar um programa fonte LALG em uma sequência de tokens (pares `<cadeia, token>`), identificando também possíveis erros léxicos.

2 Decisões de Projeto e Justificativas

2.1 Ferramenta Escolhida

O `flex` é uma ferramenta padrão e altamente eficiente para a geração de analisadores léxicos, amplamente utilizada na construção de compiladores. Sua integração com C facilita a manipulação de strings e a entrada/saída de dados, além de gerar um código otimizado.

2.2 Formato de Saída dos Tokens

Os tokens são impressos no formato `<lexema> - <tipo_do_token>` ou `<lexema>` para palavras reservadas, conforme a exemplificação fornecida no documento do trabalho.

Esta escolha facilita a compreensão da saída para fins de depuração e verificação, evitando o uso de códigos numéricos, uma exigência explícita do trabalho para facilitar o entendimento.

2.3 Tratamento de Comentários

Comentários de linha única, delimitados por chaves `{` e `}`, são reconhecidos e ignorados pelo analisador léxico. O comportamento padrão de um analisador léxico é descartar elementos que não contribuem para a estrutura sintática do programa, como comentários. A implementação utiliza um estado exclusivo (`%x COMMENT`) no `flex` para consumir o conteúdo entre as chaves.

2.4 Tabela de Palavras Reservadas

A implementação da tabela de palavras reservadas é feita implicitamente pela ordem das regras no arquivo `.l`. As regras para palavras reservadas são definidas antes da regra geral para identificadores. O `flex` prioriza a regra

que aparece primeiro no arquivo em caso de padrões ambíguos ou sobrepostos. Ao colocar as palavras reservadas antes do padrão de identificadores, garantimos que termos como "program" sejam sempre reconhecidos como a palavra reservada "program" e não como um identificador. Esta abordagem é inerentemente eficiente, pois o `flex` gera um autômato otimizado.

2.5 Tratamento de Erros Léxicos

Implementação de tratamento de erros léxicos específicos para cenários como "número real mal formado", "identificador muito grande", "símbolo não pertencente à linguagem" e "comentário não terminado". As mensagens de erro seguem o formato `<lexema> - erro - <mensagem_específica>`. O tratamento específico facilita a identificação e correção de problemas no código-fonte LALG.

3 Especificação do Analisador Léxico (Flex)

O analisador léxico é especificado em um arquivo `.l` para o `flex`.

3.1 Seção de Definições (código C e opções Flex)

- `%option noyywrap`: Desabilita a necessidade de uma função `yywrap()` para indicar o fim da entrada.
- `%option yylineno`: Habilita a variável global `yylineno` para rastreamento do número da linha.
- Declaração de variáveis globais como `current_line` e `current_column` para controle de posição.
- Definição de `MAX_ID_LENGTH` para verificar o tamanho de identificadores.
- Funções auxiliares `print_token(lexeme, token.type)` para formatar a saída e `report_lexical_error(lexeme, error_message)` para mensagens de erro específicas.
- Definição de estado `%x COMMENT` para o tratamento de comentários.

3.2 Seção de Regras (Padrões e Ações)

As regras são definidas por expressões regulares e as ações \mathbb{C} associadas quando um padrão é casado. A ordem é crucial.

3.2.1 Palavras Reservadas:

- Padrões exatos para cada palavra reservada LALG ("program", "var", "integer", "begin", "end", "read", "write", "while", "do", "if", "then", "else", "const", "procedure", "for", "to").
- Ação: Chama `print_token(yytext, "palavra_reservada")`, onde "palavra_reservada" é o próprio lexema.

3.2.2 Símbolos Especiais:

- Padrões exatos para operadores e delimitadores (e.g., ":", "(", ")", " ", " ", " ", " ", ":", "=", "<>", ">=", "<=", "!", "!", "+", "-", "*", "/").
- Ação: Chama `print_token(yytext, "simb_nome_do_simbolo")`.

3.2.3 Números:

- **Erro de Número Real Mal Formado:** $[0-9]+\{a-zA-Z\}[_a-zA-Z0-9]^*$ (ex: 1.a23). Esta regra vem primeiro para capturar especificamente o erro.
 - Ação: `report_lexical_error(yytext, "numero real mal formado").`
- **Número Real Válido:** $[0-9]+\{0-9\}+.$
 - Ação: `print_token(yytext, "num_real").`
- **Número Inteiro:** $[0-9]^+.$
 - Ação: `print_token(yytext, "num_int").`

3.2.4 Identificadores:

`[a-zA-Z] [_a-zA-Z0-9]*`.

- Ação: Verifica `yyleng` (tamanho do lexema). Se `yyleng > MAX_ID_LENGTH`, chama `report_lexical_error(yytext, "identificador muito grande")`. Caso contrário, chama `print_token(yytext, "id")`.

3.2.5 Comentários:

- `"{"`: Inicia o estado `COMMENT`.
- `<COMMENT>[^{]}^*`: Ignora qualquer caractere que não seja `}` dentro do comentário.
- `<COMMENT>"`: Finaliza o estado `COMMENT`.
- `<COMMENT><<EOF>>`: Captura o erro de comentário não terminado.
 - Ação: `report_lexical_error("{", "comentario nao terminado")`.

3.2.6 Espaços em Branco e Quebras de Linha:

- `[\t]+`: Ignora espaços e tabulações.
- `"\n"`: Ignora quebras de linha e reinicia `current_column`, incrementando `current_line`.

3.2.7 Caracteres Não Reconhecidos (Erro Genérico):

- `.`: Captura qualquer caractere que não foi casado por nenhuma regra anterior.
- Ação: `report_lexical_error(yytext, "simbolo nao pertencente a linguagem")`.

3.3 Seção de Código C Adicional

- Função `main`:
 - Responsável por abrir o arquivo de entrada (se fornecido como argumento) ou ler do `stdin`.

- Chama `yylex()` para iniciar a análise léxica.
- Fecha o arquivo de entrada, se aplicável.

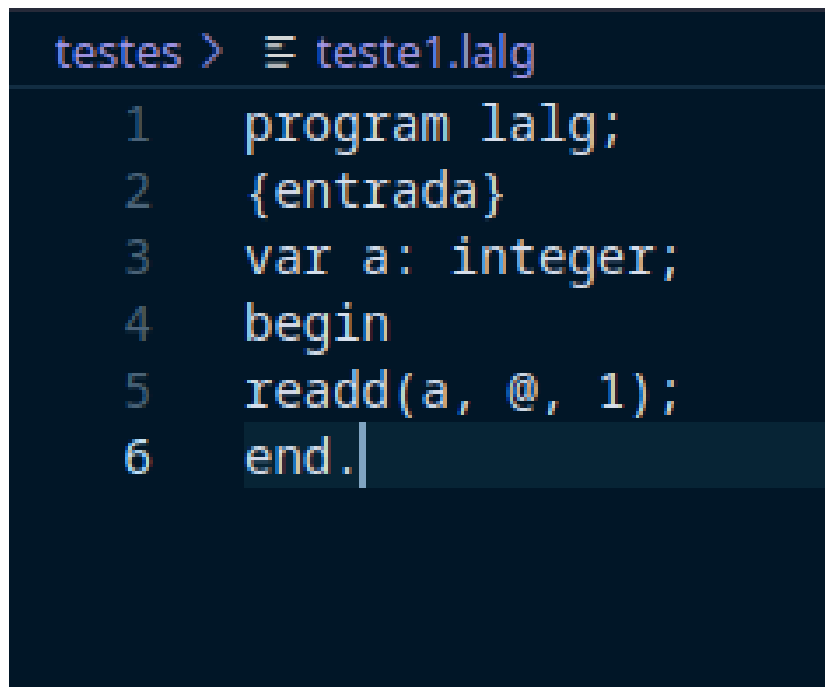
4 Compilação e Execução

Para compilar: `make`

Para testar: `make run`

Os `.out` gerados na pasta `testes` possui as respectivas saídas esperadas.

5 Exemplos de Execução



```
testes > ≡ teste1.lalg
1  program lalg;
2  {entrada}
3  var a: integer;
4  begin
5  readd(a, @, 1);
6  end.
```

Figure 1: Exemplo de Teste 1

```
testes > ≡ teste2.lalg
1  program lalg;
2  {teste}
3  var a: integer;
4  begin
5  readd(a,@,1);
6  end.
```

Figure 2: Exemplo de Teste 2

```
testes > ≡ teste3.lalg
1  program lalg;
2  {outro teste}
3  var a, minha_variavel: integer;
4  begin
5  readd(a,@,1.a23);
6  end.
```

Figure 3: Exemplo de Teste 3