Universidade do Estado do Rio de Janeiro
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Compiladores

Relatório Técnico do Analisador Sintático

Aluno:
Marcio Sirimarco de Souza Junior - 201710076811
Mateus Henrique Freitas Maciel - 202220460111

Professor(a):
Lis Custódio

9 de novembro de 2025

1 Descrição Teórica do Programa

O analisador sintático construído é um **analisador descendente recursivo preditivo**. Esta abordagem é caracterizada por ter uma função em C para cada não-terminal da gramática (por exemplo, `programa()`, `comandos()`, `expr()`). A análise inicia na função do símbolo inicial (`programa()`) e as funções se chamam recursivamente, "descendo"pela árvore de derivação à medida que consomem tokens da entrada.

Verificação LL(1): A gramática original não era LL(1) devido à **recursão à esquerda** nas regras de expressão (ex: `<expr> ::= <expr> + <term>`). Para que a análise descendente funcione (evitando loops infinitos), a gramática foi transformada:

```
<expr> ::= <term> <expr_linha>
<expr_linha> ::= + <term> <expr_linha> | - <term> <expr_linha> | ε

<term> ::= <factor> <term_linha>
<term_linha> ::= * <factor> <term_linha> | / <factor> <term_linha> | ε
```

Essa transformação elimina a recursão direta e torna a gramática LL(1), permitindo que o analisador seja "preditivo". Ele usa os conjuntos **FIRST** (definidos no `.h`) para prever qual produção deve ser escolhida. Por exemplo, a função `comando()` usa o **FIRST** para decidir se deve chamar `if_stmt()` (se o token for `TOKEN_IF`) ou `while_stmt()` (se for `TOKEN_WHILE`). A linguagem em LL(1) se encontra no documento: “linguagem para LL(1)”.

Tratamento de Erros (Modo Pânico): O analisador implementa a recuperação de erro via **Modo Pânico**. Esta técnica baseia-se nos conjuntos **FOLLOW**, que são pré-calculados e definidos no `.h`. Ao encontrar um erro (um token inesperado), o analisador:

- Imprime uma mensagem de erro, incluindo a linha e a coluna.
- Ativa o modo pânico, que descarta tokens da entrada.
- Ele para de descartar tokens assim que encontrar um "símbolo de sincronização"— um token que pertence ao conjunto **FOLLOW** da regra que estava sendo analisada.
- Isso permite que o analisador continue a análise a partir desse ponto seguro, possibilitando a detecção de múltiplos erros em uma única compilação.

2 Descrição da Estrutura do Programa

O analisador está estruturado em quatro arquivos principais:

- `compilador.c`: O programa principal (`main`) que serve como "driver". Ele lê o arquivo fonte, chama a função principal de análise (`analisar()`) e reporta o resultado final (sucesso ou falha).
- `analisador_lexico.c` / `analisador_lexico.h`: O scanner, responsável por fornecer o fluxo de tokens (`get_token()`) para o analisador sintático.
- `analisador_sintatico.h`: O cabeçalho que define a interface (`analisar()`) e, crucialmente, todos os conjuntos **FIRST** e **FOLLOW** como macros C.
- `analisador_sintatico.c`: Suas principais funções são:
  - `analisar()`: Ponto de entrada que inicializa o léxico, obtém o primeiro token (`token_atual`) e inicia a análise chamando `programa()`.
  - `programa()`, `decls()`, `comando()`, `if_stmt()`, etc.: Um conjunto de funções estáticas, cada uma implementando uma regra não-terminal da gramática.

- `consumir(int token_esperado, ...)`: Função que verifica se o `token_atual` é o esperado. Se sim, avança para o próximo token chamando `get_token()`. Se não, chama `erro_sintatico()`.
- `erro_sintatico(const char* mensagem, ...)`: Implementa o Modo Pânico. Foi corrigida para consumir o token do erro antes de sincronizar, evitando loops infinitos (o "core dumped"que encontramos).
- `in(int token, ...)`: Função utilitária para verificar se um token pertence a um **FIRST** ou **FOLLOW** set.

3 Descrição do Funcionamento

O fluxo de execução é comandado pelo analisador sintático:

- **Inicialização:** `main()` em `compilador.c` chama `analisar()`. Esta função zera a flag `houve_erro_sintatico`, inicializa o léxico e obtém o primeiro `token_atual`.
- **Análise (Sucesso):** A função `programa()` é chamada. Ela chama `consumir(TOKEN_INICIO)`. Se for bem-sucedido, o `consumir` pega o próximo token e `programa()` chama `decls()`. O processo se repete, com as funções se chamando recursivamente (`comandos()` -> `comando()` -> `if_stmt()` -> `expr()`). Se a função `programa()` terminar e o `token_atual` for `TOKEN_EOF`, `analisar()` retorna `true`, e `main()` imprime "[SUCESSO]".
- **Análise (Erro):** Suponha que `while_stmt()` espere um `)` mas `token_atual` é `{`.
  - A função `consumir(TOKEN_FECHA_PARENTESES, ...)` falha.
  - `consumir()` chama `erro_sintatico()`, passando o `FOLLOW(while_stmt)` como conjunto de sincronização.
  - `erro_sintatico()` imprime o erro (linha/coluna) e marca `houve_erro_sintatico = true`.
  - **Correção do Loop:** `erro_sintatico()` consome o token `{` (o token do erro) para garantir que a análise avance.
  - **Modo Pânico:** A função entra em loop, mas como `{` está no `FOLLOW(while_stmt)` (pois `{` inicia um bloco, que é um comando), ela sincroniza imediatamente.
  - `erro_sintatico()` retorna. A análise continua, tratando o `{` como o início de um novo comando.
  - A análise continua até o fim. Como a flag `houve_erro_sintatico` é `true`, `analisar()` retorna `false` e ao final de toda a análise sintática a `main()` imprime "[FALHA]".

4 Descrição dos Testes Realizados e Saídas Obtidas

Foram realizados quatro testes principais para validar a corretude e a robustez do analisador. Os testes seguem anexados junto com código na pasta “Analisador Sintático”.

Testes de Sucesso

Teste 1: `teste_sucesso1.txt` **Objetivo:** Validar a capacidade do analisador de aceitar programas sintaticamente corretos, testando a estrutura completa `inicio/fim`, múltiplas declarações, `read`, `print`, `if/else` e um loop `while` com um bloco `{}`.

Código-Fonte de Entrada:

```
inicio
  -- Arquivo de teste de sucesso

  int a;
  float b;
  string msg;

  read(a);
  b = (a * 10.5) / 2;

  if (a) {
    msg = "Contador e maior que zero";
  } else {
    msg = "Contador e zero ou negativo";
  }

  while (a) {
    print(a);
    a = a - 1;
    print(b);
  }

  print("Fim do loop");
  print(msg);

fim
```

Teste 2: `teste_sucesso2.txt` **Objetivo:** Focar em casos mais específicos, como a produção `factor` negativo (`x = -10;`), expressões aninhadas (`(2 + -5)`) e a produção `else_opt` vazia (um `if` sem `else`).

Código-Fonte de Entrada:

```
inicio
  -- Teste de sucesso 2
  -- Foco em expressoes negativas e if sem else

  int x;
  int y;

  x = -10;  -- Testa o factor: - <factor>
  y = x * (2 + -5); -- Testa expressao aninhada com negativo

  if (y)
    print("y nao e zero");

  -- Testa chamada de funcao com args vazio
  -- (Sintaticamente valido, embora semanticamente possa nao ser)
  x();

fim
```

Saída Gerada (para ambos os testes de sucesso):

```
[SUCESSO] Analise Sintatica concluida.
```

Análise: A saída confirma que o analisador processou todos os tokens de ambos os arquivos e terminou corretamente no `TOKEN_EOF`, provando que a gramática LL(1) e o analisador descendente estão funcionando corretamente.

Testes de Erro

Teste 3: `teste_erro1.txt` **Objetivo:** Testar a capacidade de recuperação de múltiplos erros, incluindo omissão de `;`, `)` e erros em cascata.

Código-Fonte de Entrada:

```
inicio
  -- Arquivo de teste com erros sintaticos

  int contador -- Erro 1: Falta ;

  float total;

  -- Erro 2: Falta ')' no 'while'
  while (contador > 0 {
    total = total + contador;
    contador = contador - 1;
  }

  -- Erro 3: 'print' sem ';' no final
  print(total)
fim
```

Saída Gerada:

```
Erro linha 6, coluna 3:
  >   float total;
      ^
ERRO SINTATICO: Esperava token 59.
  Token recebido: 269 (Inesperado)
  Iniciando modo panico... Sincronizando com { 268 269 267 257 265 266 262 264 123 261 }
  Sincronizacao encontrada. Continuando analise no token 257.
Erro linha 6, coluna 9:
  >   float total;
            ^
ERRO SINTATICO: Depois do ID esperava '=' ou '('.
  Token recebido: 257 (Inesperado)
  Iniciando modo panico... Sincronizando com { 257 265 266 262 264 123 261 125 }
  Sincronizacao encontrada. Continuando analise no token 264.
Erro linha 9, coluna 19:
  >   while (contador > 0 {
                        ^
ERRO SINTATICO: Esperava token 41.
  Token recebido: 271 (Inesperado)
  Iniciando modo panico... Sincronizando com { 257 265 266 262 264 123 261 125 }
  Sincronizacao encontrada. Continuando analise no token 123.
Erro linha 17, coluna 1:
  > fim
    ^
ERRO SINTATICO: Esperava token 59.
  Token recebido: 261 (Inesperado)
  Iniciando modo panico... Sincronizando com { 257 265 266 262 264 123 261 125 }
Erro linha 17, coluna 4:
  > fim
     ^
ERRO SINTATICO: Esperado um comando ou 'fim'.
  Token recebido: 270 (Inesperado)
  Iniciando modo pnico... Sincronizando com { 261 125 }
Erro linha 17, coluna 4:
  > fim
     ^
ERRO SINTATICO: Esperava token 261.
  Token recebido: 270 (Inesperado)
  Iniciando modo panico... Sincronizando com { 270 }
[FALHA] Analise Sintatica encontrou erros.
```

Análise: O analisador detectou e se recuperou de múltiplos erros. **Falta de `;` (Linha 6):** O analisador esperava um `;` após `contador` e encontrou `float`. O pânico sincronizou no `TOKEN_IDENTIFICADOR` (`total`). **Falta de `)` (Linha 9):** O `while` esperava `)` mas encontrou `{`. O pânico sincronizou no `{`, pois ele está no `FOLLOW(while_stmt)`. **Falta de `;` no `print` (Linha 17):** O `print` esperava `;` mas encontrou `fim`. O pânico sincronizou em `fim`. A recuperação levou a erros em cascata até o `EOF`, e a análise terminou com falha.

Teste 4: `teste_erro2.txt` **Objetivo:** Testar erros em expressões (`atribuicao`) e a recuperação de erros que anteriormente causavam o loop infinito ("core dumped").

Código-Fonte de Entrada:

```
inicio
  -- Teste de erros sintaticos (atribuicao, ID, bloco)

  int a;
  int b;

  a = ;  -- Erro 1: Expressao faltando apos a atribuicao

  b 10;  -- Erro 2: Esperava '=' ou '(' depois do ID

  {
    print("bloco incompleto");

  -- Erro 3: Falta '}' para fechar o bloco

fim
```

Saída Gerada:

```
Erro linha 7, coluna 7:
  >   a = ;  -- Erro 1: Expressao faltando apos a atribuicao
        ^
ERRO SINTATICO: Esperado expressao.
  Token recebido: 59 (Inesperado)
  Iniciando modo panico... Sincronizando com { 41 44 59 }
  Sincronizacao encontrada. Continuando analise no token 59.
Erro linha 16, coluna 1:
  > fim
    ^
ERRO SINTATICO: Esperava token 125.
  Token recebido: 261 (Inesperado)
  Iniciando modo panico... Sincronizando com { 257 265 266 262 264 123 261 125 }
Erro linha 16, coluna 4:
  > fim
     ^
ERRO SINTATICO: Esperado um comando ou 'fim'.
  Token recebido: 270 (Inesperado)
  Iniciando modo panico... Sincronizando com { 261 125 }
Erro linha 16, coluna 4:
  > fim
     ^
ERRO SINTATICO: Esperava token 261.
  Token recebido: 270 (Inesperado)
  Iniciando modo panico... Sincronizando com { 270 }
[FALHA] Analise Sintatica encontrou erros.
```

Análise: Expressão Faltando (Linha 7): Em `a = ;`, a função `expr()` esperava um fator mas encontrou `;`. O pânico sincronizou no próprio `;`, pois `TOKEN_PONTO_VIRGULA` pertence ao `FOLLOW(expr)`. **Erro `b 10;` (Linha 9):** Este erro não aparece no log. Isso demonstra o funcionamento do modo pânico: após o erro da linha 7, o analisador descartou tokens (`b`, `10`, `;`) até encontrar um símbolo de sincronização (`{` na linha 11). **Falta de `}` (Linha 16):** O bloco esperava `}` mas encontrou `fim`. O pânico foi ativado e sincronizou em `fim`, que está no `FOLLOW(bloco)`, gerando erros em cascata até o `EOF`.

Conclusão dos Testes

Os testes demonstram que o analisador sintático não apenas valida programas corretos, mas também é robusto o suficiente para lidar com programas incorretos, reportando múltiplos erros e finalizando a análise graças à implementação correta do modo pânico.