Compiladores

Gabarito Comentado: Avaliação Front-End

Respostas esperadas e análise de dificuldade

Prof. Aléssio Miranda Júnior
CEFET-MG - Campus Timóteo

Fevereiro de 2026

Legenda de Dificuldade:

Fácil – Aplicação direta da teoria | Médio – Exige raciocínio e construção | Difícil – Exige domínio integrado

Questão 1: Análise Sintática LL(1) vs LR (SLR) [12,0 pts]

a) Eliminação de Recursão à Esquerda [4,0 pts]

Dificuldade: Média. O aluno precisa identificar a recursão à esquerda e aplicar o algoritmo corretamente. Erros comuns: esquecer de tratar $E \to T$ ou confundir ε.
Referência: Contemplado na Questão 11 da Lista de Exercícios.

Resposta esperada:

A gramática **não** pode ser usada diretamente em um parser descendente recursivo porque a produção $E \to E + T$ possui **recursão à esquerda direta**, o que causaria loop infinito em `parseExpression()`.

Aplicando o algoritmo de eliminação:

- Produções originais de E: $E \to E + T \mid T$
- $\alpha = + T, \quad \beta = T$

Gramática transformada:

1) $E \to T\ E_{novo}$
2) $E_{novo} \to +\ T\ E_{novo}$
3) $E_{novo} \to \varepsilon$
4) $T \to \textbf{id}$

(Nota: Utilizamos E_{novo} para não confundir com o E' da gramática aumentada do autômato.)

Critério de correção: 2,0 pts pela justificativa (recursão à esquerda → loop infinito); 2,0 pts pela gramática transformada correta com o novo não-terminal e ε.

b) Autômato de Itens LR(0) [4,0 pts]

Dificuldade: Difícil. É a questão mais trabalhosa da prova. Exige construção metódica de closure e goto para vários estados. Principal erro: esquecer itens de closure ou transições.
Referência: Contemplado nas Questões 14 e 15 da Lista de Exercícios.

Resposta esperada:
Estado I_0 (closure de $E' \to \bullet\ E$):

```
E' -> . E
E  -> . E + T
E  -> . T
T  -> . id
```

$I_0 \xrightarrow{E} I_1$:

```
E' -> E .              [ACEITAR]
E  -> E . + T
```

$I_0 \xrightarrow{T} I_2$:

```
E -> T .
```

$I_0 \xrightarrow{\mathbf{id}} I_3$:

```
T -> id .
```

$I_1 \xrightarrow{+} I_4$:

```
E -> E + . T
T -> . id
```

$I_4 \xrightarrow{T} I_5$:

```
E -> E + T .
```

$I_4 \xrightarrow{\mathbf{id}} I_3$ (estado já existente).
Total: 6 estados (I_0 a I_5).
Critério: 1,0 pt pelo I_0 correto (com o closure de E e T); 1,5 pt por I_1 (shift/reduce) e as transições no I_0; 1,5 pt pelos estados finais e fechamento do autômato.

c) FIRST, FOLLOW e Conflitos SLR(1) [4,0 pts]

Dificuldade: Média. Calcular FIRST/FOLLOW é mecânico se o aluno domina as regras. A análise de conflito exige interpretar o estado I_1.
Referência: Contemplado nas Questões 12 e 15 da Lista de Exercícios.

Resposta esperada:

Não-terminal	FIRST	FOLLOW
E	{**id**}	{\$, +}
T	{**id**}	{\$, +}

Análise de conflitos no estado I_1:
I_1 contém: $E' \to E \bullet$ (reduce por regra 0) e $E \to E\ \bullet\ +\ T$ (shift no +).

- **Ação de reduce** por $E' \to E$: deve ocorrer nos tokens de FOLLOW$(E') = \{\$\}$.
- **Ação de shift** no token $+$.
- Como $\$ \neq +$, **não há conflito Shift/Reduce**.
- Não há dois reduces no mesmo estado, logo **não há conflito Reduce/Reduce**.

$\Rightarrow$ A gramática é **SLR(1)**.

Critério: 2,0 pts pela tabela FIRST/FOLLOW correta; 2,0 pts pela análise de conflito bem justificada.

Questão 2: Decisões de Projeto (Léxico + AST) [8,0 pts]

a) Tokenização [2,0 pts]

Dificuldade: Fácil. Aplicação direta do conteúdo da Etapa 2. Bastava percorrer a string e categorizar.

Referência: Contemplado na Questão 9 da Lista de Exercícios.

Resposta esperada para `x := 10 + 5 * y;`:

#	Lexema	Categoria
1	`x`	IDENTIFIER
2	`:=`	ASSIGN
3	`10`	NUM_LITERAL
4	`+`	OP_ADD
5	`5`	NUM_LITERAL
6	`*`	OP_MUL
7	`y`	IDENTIFIER
8	`;`	SEMICOLON

Critério: 0,25 pt por token correto (lexema + categoria).

b) Precedência no Código [3,0 pts]

Dificuldade: Média. O aluno precisa articular a relação entre hierarquia de chamadas e precedência. Exige compreensão do próprio código.

Referência: Baseado no Trabalho Prático. Tópico de Parser Descendente abordado na Questão 5 da Lista.

Resposta esperada:

A precedência é garantida pela **hierarquia de chamadas recursivas**:

1. `parseExpression()` trata operadores de **menor** precedência ($+$, $-$). Internamente, chama `parseTerm()`.
2. `parseTerm()` trata operadores de **maior** precedência ($*$, `div`). Internamente, chama `parseFactor()`.
3. `parseFactor()` trata os **operandos atômicos**: literais, identificadores e expressões entre parênteses.

Como `parseTerm()` é chamado *dentro* de `parseExpression()`, os nós de multiplicação ficam **mais profundos** na AST, sendo avaliados **primeiro**. Isso garante que `5 * y` vire um subgrupo antes de ser somado com `10`.

Critério: 1,0 pt pela hierarquia correta; 1,0 pt por explicar que profundidade = precedência; 1,0 pt por conectar com a AST gerada.

c) Desenho da AST [3,0 pts]

Dificuldade: Média. Exige visualização da árvore. Erro comum: inverter a precedência ou colocar a multiplicação acima da adição.

Referência: Baseado no Trabalho Prático. Aplicações de AST abordadas nas Questões 2 e 3 da Lista.

Resposta esperada (forma hierárquica):

```
AssignmentCommand
|-- lhs: Identifier("x")
\-- rhs: BinaryExpression("+")
         |-- left:  Literal(10)
         \-- right: BinaryExpression("*")
                    |-- left:  Literal(5)
                    \-- right: Identifier("y")
```

Critério: 1,0 pt pelo nó raiz `AssignmentCommand` correto; 1,0 pt pela adição como operação externa e multiplicação como interna (precedência); 1,0 pt pelos nós folha corretos (`Identifier` vs `Literal`).

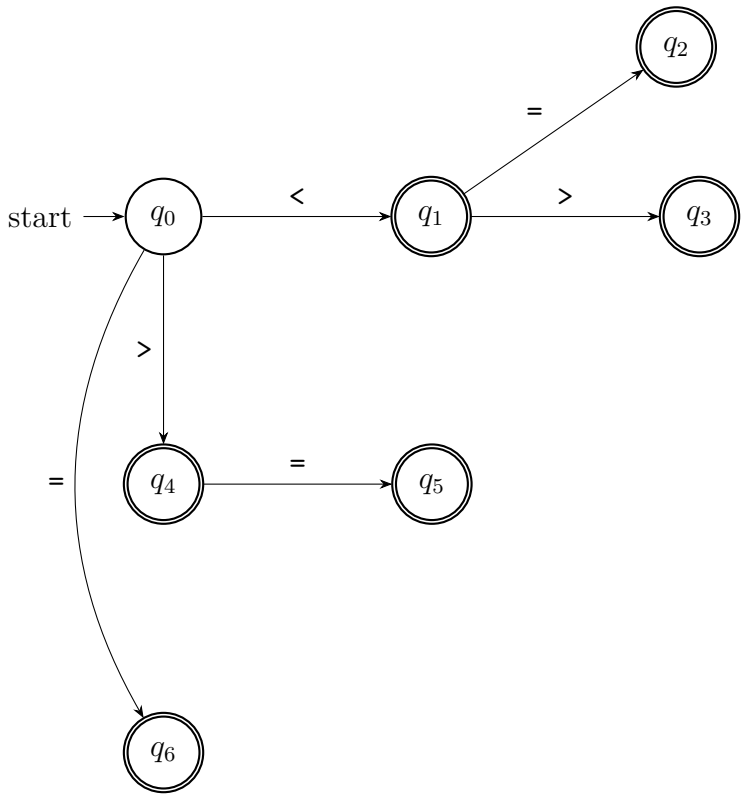
Questão 3: O Analisador Léxico e o Lookahead [5,0 pts]

a) AFD dos Operadores Relacionais [3,0 pts]

Dificuldade: Média. O desenho do AFD é direto, mas exige atenção aos estados de aceitação e transições "otherwise" (retract).

Referência: Construção de autômatos contemplada na Questão 10 da Lista de Exercícios.

Resposta esperada:



- q_1: aceita `<` (se próximo char $\neq$ `=` e $\neq$ `>`, faz *retract*)
- q_2: aceita `<=`
- q_3: aceita `<>`
- q_4: aceita `>` (se próximo char $\neq$ `=`, faz *retract*)
- q_5: aceita `>=`
- q_6: aceita `=`

Critério: 1,0 pt pela estrutura geral; 1,0 pt por todos os estados de aceitação corretos; 1,0 pt pelas transições e menção ao retract.

b) Lookahead / peek() [2,0 pts]

Dificuldade: Fácil. O aluno implementou isso diretamente na Etapa 2. Basta descrever o mecanismo.
Referência: Tópico restrito à implementação do Trabalho Prático.

Resposta esperada:
Quando `nextToken()` lê o caractere `<`, ele não pode decidir imediatamente se é `<`, `<=` ou `<>`. Então:

1 Chama `peek()` (ou lê o próximo caractere sem consumir).

2 Se o próximo for `=`: consome e retorna token `LESS_EQUAL`.

3 Se o próximo for `>`: consome e retorna token `NOT_EQUAL`.

4 Caso contrário: **não** consome (faz "unread" / retract) e retorna token `LESS_THAN`.

O `peek()` age como o lookahead de 1 caractere, mantendo o ponteiro de leitura inalterado até a decisão ser tomada.

Critério: 1,0 pt por descrever o mecanismo de peek/lookahead; 1,0 pt por cobrir os 3 casos (`<`, `<=`, `<>`).

Questão 4: Recuperação de Erros (Panic Mode) [5,0 pts]

a) Atuação do Panic Mode [3,0 pts]

Dificuldade: Média. Exige raciocínio procedimental sobre a execução do parser. Alunos confundem "descartar tokens" com "ignorar a linha toda".
Referência: Contemplado diretamente na Questão 4 da Lista de Exercícios.

Resposta esperada:

Ao processar `x := 10`, o parser espera um `;` após a expressão. Porém, o próximo token é `y` (IDENTIFIER). Nesse momento:

1. O parser detecta o erro: **token esperado** = `SEMICOLON`; **token encontrado** = `IDENTIFIER "y"`.
2. Emite mensagem de erro: *"Erro na linha 2: esperado ';' antes de 'y'."*.
3. Entra em **Panic Mode**: começa a **descartar tokens** da entrada um a um (`y`, `:=`, `20`) até encontrar um **token de sincronização**.
4. Ao encontrar o `;` após `20`, o parser **se ressincroniza** e volta ao estado normal, continuando a análise a partir do próximo comando.
5. Assim, o `end.` final é processado normalmente.

Critério: 1,0 pt por identificar o ponto exato do erro; 1,0 pt por descrever o descarte de tokens; 1,0 pt por explicar a ressincronização.

b) Tokens de Sincronização [2,0 pts]

Dificuldade: Fácil. Lista direta dos delimitadores de bloco/comando em Pascal.
Referência: Contemplado diretamente na Questão 4 da Lista de Exercícios.

Resposta esperada:

Tokens de sincronização recomendados dentro de um bloco `begin...end`:

- `;` (ponto-e-vírgula) – delimitador de comandos
- `end` – fim de bloco
- `begin` – início de sub-bloco
- `if`, `while`, `for`, `repeat` – início de comandos estruturados
- `.` (ponto final de programa)

A ideia é escolher tokens que marquem "fronteiras" seguras de comandos para que o parser possa retomar de um ponto confiável.

Critério: 1,0 pt por listar `;` e `end`; 1,0 pt por justificar a escolha.

Resumo de Dificuldade por Item

Questão	Pts	Dificuldade	Observação
1a	4,0	Média	Eliminação de recursão à esquerda
1b	4,0	Difícil	Construção completa do autômato LR(0)
1c	4,0	Média	FIRST/FOLLOW + análise de conflito
2a	2,0	Fácil	Listar tokens (mecânico)
2b	3,0	Média	Explicar hierarquia de precedência
2c	3,0	Média	Desenhar a AST corretamente
3a	3,0	Média	Desenho de AFD
3b	2,0	Fácil	Descrever peek/lookahead
4a	3,0	Média	Raciocínio procedimental
4b	2,0	Fácil	Listar tokens de sincronização
Distribuição: 4 Fáceis (8 pts) / 5 Médias (17 pts) / 1 Difícil (4 pts) ≈ **Prova equilibrada**			