CENTRO UNIVERSITÁRIO FMU LEVEL UP

COMPILADORES

https://portal.fmu.br

Unidade 1: O Coração das Linguagens – Gramáticas e Árvores

I. Questões de Múltipla Escolha

1. **Resposta: B**
    - *Comentário:* No conjunto V (Não-Terminais), os símbolos representam estruturas sintáticas que ainda serão expandidas. Conforme Menezes (2011), eles são variáveis que "pedem" uma substituição baseada nas regras de produção.

2. **Resposta: B**
    - *Comentário:* A BNF é o padrão "de facto" para descrever a sintaxe. Foi criada para o ALGOL 60 e hoje define quase todas as linguagens modernas.

3. **Resposta: A**
    - *Comentário:* Enquanto Gramáticas Regulares lidam com padrões lineares (como CPFs ou e-mails), as GLCs conseguem "contar" e aninhar estruturas (como parênteses e blocos { }), algo fundamental para a recursividade.

4. **Resposta: B**
    - *Comentário:* As folhas são o final da linha: os tokens reais que o programador digitou (como if, 5, +).

5. **Resposta: C**
    - *Comentário:* A ambiguidade ocorre quando a gramática não define claramente a precedência, permitindo mais de uma interpretação para a mesma sentença.

6. **Resposta: C**
    - *Comentário:* O "L" em LL(k) significa *Left-to-right* (leitura) e *Leftmost derivation* (expansão a partir do símbolo mais à esquerda).

7. **Resposta: C**
    - *Comentário:* O símbolo inicial S é a "raiz" de tudo. Toda sentença válida na linguagem deve obrigatoriamente começar a partir de sua derivação.

8. **Resposta: B**
    - *Comentário:* Na derivação à direita, sempre expandimos o não-terminal mais próximo do final da sequência. É a base para os parsers LR (Bottom-Up).

9. **Resposta: B**
    - *Comentário:* Aho et al. destacam que as GLCs são poderosas o suficiente para as linguagens atuais e podem ser reconhecidas em tempo linear por Autômatos de Pilha.

10. **Resposta: B**
    - *Comentário:* O Front-End cuida da análise e validação. As regras de produção são o "molde" que diz se aquela construção de código é permitida.

II. Perguntas Reflexivas

1. **Ambiguidade (Humana vs. Programação):**

Na linguagem humana, o contexto e a entonação ajudam a resolver a ambiguidade. Na programação, o computador é uma máquina determinística. Se uma gramática for ambígua, o compilador pode interpretar pagar_imposto() de duas formas diferentes, gerando comportamentos imprevisíveis e falhas críticas de segurança. Por isso, em projeto de linguagens, a ambiguidade é um erro que deve ser eliminado.

2. **Influência da Árvore no Processador:**

A árvore define a ordem de execução. O processador segue a estrutura "Bottom-Up": ele precisa dos resultados dos nós inferiores para processar os superiores. Se a multiplicação está em um nível mais baixo que a soma, o processador carregará os valores nos registradores e executará a multiplicação primeiro, respeitando a lógica matemática inserida na gramática.

III. Exercícios Práticos

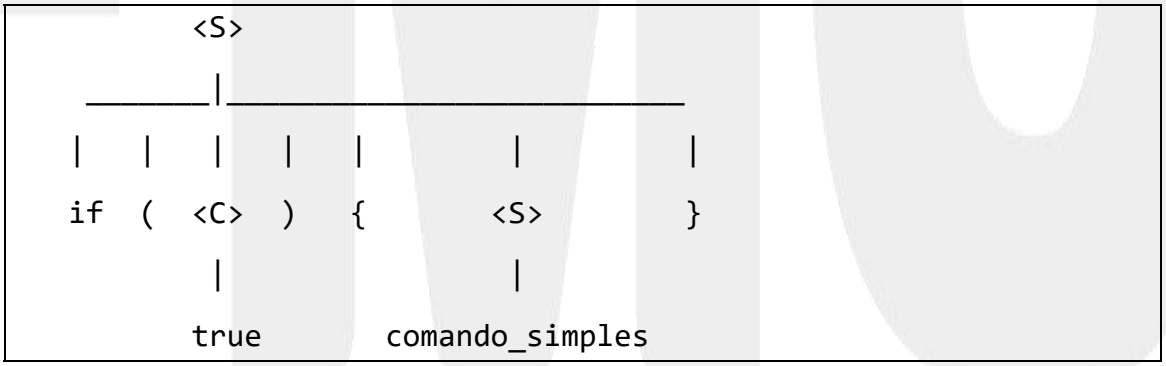
1. Modelagem (Números Pares de Dois Dígitos)

- Regras:
    1. <S> → <D1><D2>
    2. <D1> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    3. <D2> → 0 | 2 | 4 | 6 | 8

- Identificação:
    - **Não-Terminais:** {S, D1, D2} (as categorias).
    - **Terminais:** {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} (os dígitos reais).

2. Desenho Técnico: if (true) { comando_simples }

A árvore cresce hierarquicamente respeitando os parênteses e as chaves:



** Observe que cada terminal (if, (, true, etc) é uma folha, e o nó <C> e <S> interno são os não-terminais que deram origem a eles.

Unidade 2: Anatomia de um Compilador – O Front-End

I. Questões de Múltipla Escolha

1. **Resposta: B**
    - *Explicação:* A separação permite que você crie um Front-End para Python e use o mesmo Back-End (gerador de código) para Intel ou ARM, economizando anos de trabalho.

2. **Resposta: B**
    - *Explicação:* O Léxico não entende frases, apenas palavras. Essas "palavras" técnicas são os **Tokens**.

3. **Resposta: C**
    - *Explicação:* A Representação Intermediária (IR) é o produto final do Front-End; é um código "limpo" e independente de máquina que o Back-End usará para gerar o binário.

4. **Resposta: C**
    - *Explicação:* O erro é na ordem das peças. O "esqueleto" do comando IF exige um corpo (bloco). Se não existe, a gramática (sintaxe) foi violada.

5. **Resposta: C**
    - *Explicação:* A sintaxe aceita ID + ID. A semântica é que verifica se um desses IDs é uma String e o outro um Inteiro, o que causaria um erro de tipo.

6. **Resposta: B**
    - *Explicação:* A Tabela de Símbolos é o "banco de dados" do compilador. Ela guarda o nome da variável, o tipo, o escopo e o endereço de memória.

7. **Resposta: B**
    - *Explicação:* Como cita Menezes (2011), o objetivo é a produtividade humana. É mais fácil escrever em alto nível do que gerir bits manualmente.
    -
8. **Resposta: B**
    - *Explicação:* O pré-processador limpa o terreno (remove comentários, inclui bibliotecas) para que o Analisador Léxico receba apenas o código relevante.

9. **Resposta: B**
    - *Explicação:* Um erro léxico ocorre quando o "alfabeto" é violado. Por exemplo, usar um símbolo que a linguagem não reconhece de forma alguma.

10. **Resposta: B**
    - *Explicação:* Este é o princípio de *Separation of Concerns*. Cada fase resolve um problema específico (caracteres, depois estrutura, depois significado).

II. Respostas das Perguntas Reflexivas

1. **Abstração e Engenharia:**

A divisão é mais eficiente porque isola a complexidade. Se tentássemos fazer tudo num passo só, o código do compilador seria impossível de depurar. Com fases, se o erro é uma letra errada, sabemos que o problema está no Léxico. Se a lógica de tipos está errada, está no Semântico. Além disso, permite que especialistas trabalhem em partes diferentes (ex: um especialista em otimização foca apenas no Back-End).

2. **O Papel do Significado:**

A frase "As ideias verdes incolores dormem furiosamente" é **sintaticamente correta** (Sujeito + Adjetivo + Verbo + Advérbio), mas **semanticamente nula** (ideias não têm cor e não dormem). No compilador, o comando x = 5 / "texto" é sintaticamente correto (Variável = Número / Valor), mas semanticamente impossível, pois não se divide números por palavras.

III. Respostas dos Exercícios Práticos

1. Fluxo de Tokens para: resultado = (valor1 + 50) * 2

Token	Classificação
resultado	Identificador (ID)
=	Operador de Atribuição
(	Delimitador (LPAREN)
valor1	Identificador (ID)
+	Operador Aritmético
50	Literal Numérico (Inteiro)
)	Delimitador (RPAREN)
*	Operador Aritmético
2	Literal Numérico (Inteiro)

2. Simulação de Falhas para: print(x + 10)

- **Cenário de Erro Léxico:** Se o programador digitasse print(x @ 10) e o símbolo @ não existisse no alfabeto da linguagem. O scanner não conseguiria gerar um token para @.

- **Cenário de Erro Semântico:** O programador escreve print(x + 10). O léxico identifica os tokens, o sintático confirma que a estrutura está correta, mas o **Analisador Semântico** descobre que a variável x nunca foi declarada anteriormente ou que x contém um ficheiro aberto, e não um número, impedindo a soma.

Unidade 3: Do Código à Execução – Back-End e Interpretadores

I. Questões de Múltipla Escolha

1. **Resposta: B**
    - *Explicação:* O Back-End foca-se na síntese. Ele "monta" o código final para o processador específico (Intel, ARM, etc.).

2. **Resposta: B**
    - *Explicação:* O interpretador não gera um ficheiro; ele traduz "na hora". Se houver um erro na linha 10, ele executa as 9 primeiras e para apenas na 10.

3. **Resposta: B**
    - *Explicação:* Registadores são a memória mais rápida do mundo, mas existem poucos. O compilador tem de "lutar" para decidir quem fica neles para o código não ficar lento.

4. **Resposta: A**
    - *Explicação:* O compilador faz o trabalho pesado antes. É como preparar uma mala: demora a fazer, mas depois a viagem (execução) é rápida.

5. **Resposta: C**
    - *Explicação:* Transformar lógica em instruções como MOV e ADD (Assembly) é tarefa do Gerador de Código do Back-End.

6. **Resposta: B**
    - *Explicação:* A Análise Sintática verifica a estrutura. Se o print exige aspas para texto, é ela quem valida essa regra gramatical.

7. **Resposta: B**
    - *Explicação:* O Bytecode permite que o mesmo programa corra em Windows, Linux ou Mac, desde que tenham a Máquina Virtual (JVM ou .NET).

8. **Resposta: B**
    - *Explicação:* A IR é o "esperanto" dos compiladores. Ela permite que o Front-End comunique com o Back-End sem depender de uma máquina específica.

9. **Resposta: B**
    - *Explicação:* Um Cross-Compiler é usado, por exemplo, quando compilamos um jogo no PC (Windows/Intel) para correr numa consola ou telemóvel (ARM).

10. **Resposta: B**
    - *Explicação:* A otimização melhora a performance (ex: remove contas inúteis como x * 1), mas nunca pode mudar o resultado que o programador pretendia.

II. Perguntas Reflexivas

1. **Escolha de Ferramenta (ABS de um Carro):** Para sistemas de missão crítica como travões ABS, deve-se usar uma **linguagem compilada** (como C ou C++). Justificação: Num travão, milissegundos importam. Um interpretador poderia ter uma pausa para "traduzir" ou sofrer com variações de velocidade. O código compilado corre diretamente no metal do processador com a velocidade máxima e previsível.

2. **O Futuro da Tradução:** Embora os interpretadores estejam muito rápidos (graças ao JIT), os compiladores nunca deixarão de existir. Porquê? Porque para interpretar algo, precisas de um programa tradutor a correr ao mesmo tempo, o que gasta memória e bateria. Para sistemas embebidos (frigoríficos, relógios, carros) e software de alta performance, o código "pré-traduzido" (compilado) será sempre a escolha superior.

III. Exercícios Práticos

1. Mapeamento de Execução (y = 10 / 2) sem instrução de divisão: O Back-End transformaria a divisão numa estrutura de subtrações sucessivas:

- Ele pegaria no 10 e subtrairia 2 (Resultado 8, Contador 1).
- Subtrairia 2 novamente (Resultado 6, Contador 2).
- ...até chegar a zero. O valor final do Contador (5) seria o resultado guardado na variável y. O compilador decompõe o complexo em simples.

2. Definição de JIT (Just-In-Time Compiler): O JIT é um tradutor híbrido. Ele começa a interpretar o código linha a linha (como o Python), mas se percebe que um trecho de código (como um loop) é executado muitas vezes, ele compila esse trecho para código de máquina real "na hora". Assim, ele une a flexibilidade do interpretador com a velocidade final do compilador.

Unidade 4: Ferramentas de Construção (Os "Ajudantes")

I. Questões de Múltipla Escolha

1. **Resposta: B**
    - *Comentário:* Como afirma Aho et al. (2008), automatizar a criação de analisadores permite que o engenheiro foque na lógica da linguagem, enquanto a ferramenta cuida da implementação complexa dos algoritmos de busca e reconhecimento.
2. **Resposta: C**
    - *Comentário:* Enquanto o Lex cuida da parte léxica, o Yacc (*Yet Another Compiler-Compiler*) é o responsável por gerar o analisador sintático utilizando o algoritmo LALR.
3. **Resposta: A**
    - *Comentário:* O AntLR revolucionou a área ao trocar o olhar limitado do LL(1) pelo LL(*), permitindo olhar infinitos tokens à frente para resolver ambiguidades, além de gerar código para diversas linguagens modernas.
4. **Resposta: C**
    - *Comentário:* O Lark destaca-se pela ergonomia. Em poucas linhas de Python, você define uma gramática e ele já lhe entrega uma árvore sintática pronta para uso.
5. **Resposta: B**
    - *Comentário:* O Rust é a grande promessa para o Kernel do Linux justamente porque seu compilador impede erros de memória (como o famoso *buffer overflow*) antes mesmo do programa ser executado, através do seu *Borrow Checker*.
6. **Resposta: C**
    - *Comentário:* Devido ao suporte a múltiplas linguagens (Java, C#, Python, Go) e ao seu poder de análise, o AntLR é a escolha padrão da indústria para projetos complexos.
7. **Resposta: B**
    - *Comentário:* O algoritmo Earley é um diferencial do Lark, sendo capaz de processar qualquer Gramática Livre de Contexto, mesmo aquelas que seriam confusas para outros analisadores.

8. **Resposta: A**
    - *Comentário:* Desenvolver "na mão" gera erros de índice e memória. Ferramentas automáticas utilizam motores testados e provados matematicamente há décadas.
9. **Resposta: C**
    - *Comentário:* Como a regra expr define como os tokens (INT e +) se combinam para formar uma estrutura, ela é uma regra sintática.
10. **Resposta: C**
    - *Comentário:* Quando o hardware é limitado (sistemas embarcados) ou performance é o único objetivo (kernels), as ferramentas em C (Lex/Yacc) ainda reinam devido à proximidade com o hardware.

II. Perguntas Reflexivas

1. **Segurança vs. Flexibilidade (Rust vs. C):** Para o iniciante, a "rigidez" do Rust causa uma curva de aprendizado mais íngreme, pois o compilador recusa-se a gerar o executável se houver riscos de memória ("lutar com o Borrow Checker"). No entanto, em C, a flexibilidade permite que o iniciante crie programas rapidamente, mas que falham de forma catastrófica depois de rodar. A longo prazo, a rigidez do Rust aumenta a produtividade, pois o erro é corrigido na compilação, não em produção.

2. **A Escolha da Ferramenta (Lark em Python):** O Lark seria superior porque ele é "Puro Python" e sua sintaxe baseada em EBNF é muito mais intuitiva. Cientistas de dados geralmente precisam de agilidade e integração com bibliotecas como Pandas ou NumPy. Usar Lex/Yacc exigiria compilação externa em C e "pontes" (wrappers), o que aumentaria drasticamente a complexidade do projeto sem necessidade.

III. Exercícios Práticos

1. Modificação da Gramática Lark: Para aceitar multiplicação, precisamos adicionar uma nova regra ou expandir a existente. Uma forma elegante é criar a regra multiplicacao:

```
gramatica = """
    ?start: subtracao | multiplicacao
    subtracao: NUMBER "-" NUMBER
    multiplicacao: NUMBER "*" NUMBER

    %import common.NUMBER
    %import common.WS
    %ignore WS
"""
```

2. Identificação de Tokens (Entrada: v = 10 + 20): Seguindo a gramática e a ordem de leitura (esquerda para a direita):

1. v -> ID (Identificador)
2. = -> '=' (Operador de Atribuição)
3. 10 -> INT (Número Inteiro)
4. + -> '+' (Operador de Soma)
5. 20 -> INT (Número Inteiro)

Unidade 5: Análise Léxica – Identificando Palavra

I. Questões de Múltipla Escolha (Gabarito)

1. Resposta: B

- **Comentário:** Segundo **Aho et al. (2008)**, o **Token** é a classificação abstrata (ex: LITERAL_NUMERICO), enquanto o **Lexema** é a sequência de caracteres exata encontrada no código (ex: 123).

2. Resposta: C

- **Comentário:** As Expressões Regulares descrevem Linguagens Regulares (Nível 3 da Hierarquia de Chomsky), que é o formalismo perfeito para identificar padrões de palavras e símbolos sem necessidade de recursão.

3. Resposta: C

- **Comentário:** Comentários servem apenas para o desenvolvedor humano. O analisador léxico os descarta para que o analisador sintático não precise lidar com textos que não influenciam a lógica da execução.

4. Resposta: B

- **Comentário:** Em RegEx, o quantificador + indica "uma ou mais ocorrências". Se fosse o símbolo * (Fecho de Kleene), significaria "zero ou mais".

5. Resposta: C

- **Comentário:** Embora o espaço em branco seja essencial para separar tokens (como em int x), ele próprio não carrega significado semântico para o compilador e é descartado pelo scanner.

6. Resposta: C

- **Explicação:** Delimitadores como parênteses (), chaves {} e vírgulas , não realizam cálculos, mas são essenciais para o **Analisador Sintático** saber onde começa um bloco de código, onde termina uma lista de argumentos de uma função ou como separar comandos. Eles definem a hierarquia e o escopo.

7. Resposta: B

- **Explicação:** Seguindo a convenção da maioria das linguagens (C, Java, Python, Rust) e a definição de Identificadores baseada em **Expressões Regulares**, um nome válido deve começar com uma letra (maiúscula ou minúscula) ou o caractere de sublinhado _. Números são permitidos, mas nunca na primeira posição, para não confundir o analisador léxico com um literal numérico.

8. Resposta: D

- **Explicação:** De acordo com **Aho et al. (2008)**, um token é formalmente um par: <nome_do_token, valor_do_atributo>. Por exemplo, para o lexema total, o par seria <IDENTIFICADOR, ponteiro_para_tabela_de_simbolos>. Para o lexema +, seria apenas <SOMA, ->, já que o símbolo por si só já define sua função.

9. Resposta: A

- **Explicação:** Se o fluxo de caracteres apresenta um símbolo que não faz parte do alfabeto da linguagem ou não se encaixa em nenhum padrão de expressão regular definido na gramática léxica, o scanner dispara um **erro léxico**. Ele interrompe o processo por encontrar um "caractere inválido".

10. Resposta: B

- **Explicação:** A **Tabela de Símbolos** é alimentada assim que o Analisador Léxico identifica um novo **Identificador**. É nesse momento que o compilador anota: "Encontrei uma variável chamada 'x'". As fases posteriores (Sintática e Semântica) consultarão e atualizarão essa tabela com informações de tipo, escopo e valor.

Resumo para Estudo

- **Léxico:** Foca no caractere e na formação da palavra (Token).
- **Sintático:** Foca na ordem das palavras (Frase).
- **Semântico:** Foca no sentido da frase (Lógica).

II. Perguntas Reflexivas

1. **Por que identificadores não começam com números?** Isso facilita imensamente a vida do analisador léxico porque ele consegue ser **determinístico**. Se ele lê um dígito 1, ele já sabe que deve processar um **Literal Numérico**. Se ele lê uma letra v, ele sabe que é um **Identificador**. Se permitíssemos 1variavel, o analisador teria que "olhar para frente" (lookahead) e retroceder (backtracking) constantemente para decidir se o 1 é o início de um número ou de um nome, tornando o compilador muito mais lento e complexo.

2. **Espaços em branco no Fortran antigo:** No Fortran original,

   DO 10 I = 1, 5 e DO10I = 1, 5

   poderiam ser interpretados de forma ambígua. Isso dificultava a leitura humana (um erro de digitação de espaço podia mudar o sentido do código) e exigia que o compilador fizesse uma análise muito mais exaustiva para entender o contexto. As linguagens modernas usam o espaço como separador obrigatório para garantir clareza e facilitar a tokenização.

III. Exercícios Práticos

1. Expressão Regular (E-mail simplificado): Uma versão robusta e simplificada seria:

```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
```

- **Explicação:** Parte antes do @ (letras, números e símbolos comuns), o símbolo @, o domínio e a extensão (mínimo 2 letras).

2. Lark (Ignorar comentários com #): Para fazer o Lark ignorar comentários, precisamos definir o padrão do comentário e usar a diretiva %ignore.

```
from lark import Lark

gramatica_com_comentarios = """
    start: (TOKEN | OPERADOR | NUMERO)+

    TOKEN: /[a-zA-Z_][a-zA-Z0-9_]*/
    NUMERO: /[0-9]+/
    OPERADOR: "+" | "-" | "*" | "/" | "="

    # Definimos o que é um comentário: começa com # e vai até o fim da linha
    COMENTARIO: /#.*/

    %import common.WS
    %ignore WS
    %ignore COMENTARIO  // Esta linha faz o Lark descartar os comentários
"""

# Teste
texto_teste = "x = 10 # Isso é um comentário"
parser = Lark(gramatica_com_comentarios)
# O parser lerá apenas x, = e 10
```

Unidade 6: Máquinas de Estados – Autômatos Finitos

I. Respostas das Questões de Múltipla Escolha

1. **Resposta: C**
    - *Explicação:* No **AFD**, não há dúvida. Para cada estado e cada símbolo, existe uma, e apenas uma, seta de saída. Isso torna o processamento extremamente rápido.

2. **Resposta: B**
    - *Explicação:* A **Construção de Thompson** é o método clássico que utiliza estados iniciais, finais e transições vazias (ϵ) para "montar" o autômato a partir de uma Expressão Regular.

3. **Resposta: C**
    - *Explicação:* Na notação gráfica de autômatos, o círculo duplo indica que a sequência lida até aquele momento é válida e o token foi reconhecido com sucesso.

4. **Resposta: C**
    - *Explicação:* Diferente do AFN, o AFD não precisa "testar caminhos" ou voltar atrás (backtracking). Ele apenas segue o fluxo, o que garante performance O(n), onde n é o tamanho do texto.

5. **Resposta: B**
    - *Explicação:* O algoritmo de **Subconjuntos** agrupa os estados possíveis de um AFN em estados únicos de um AFD, eliminando o não-determinismo.

6. **Resposta: C**
    - *Explicação:* Se o autômato chega a um ponto onde não há saída prevista para o caractere lido e ele não está em um estado de aceitação, o scanner reporta que aquela "palavra" não pertence à linguagem.

7. **Resposta: C**
    - *Explicação:* Segundo a Hierarquia de Chomsky, os Autômatos Finitos são as máquinas de reconhecimento para as **Linguagens Regulares** (as mais simples e restritas).

8. **Resposta: B**
    - *Explicação:* O compilador é "guloso". Se ele lê >, ele espera para ver se o próximo é =. Se for, ele prefere o token >= (maior) em vez de apenas > seguido de =.

9. **Resposta: B**
    - *Explicação:* As arestas são os gatilhos. Elas indicam: "se você estiver neste estado e ler este caractere, mova-se para aquele estado".

10. **Resposta: B**
    - *Explicação:* Como destaca Menezes (2011), o determinismo é a chave. No AFD, o caminho é único; no AFN, podem existir várias opções para a mesma entrada.

II. Respostas às Perguntas Reflexivas

1. **Determinismo vs. Flexibilidade:**

Convertemos AFN em AFD porque o AFN é muito mais intuitivo para o ser humano descrever (é fácil pensar em padrões paralelos), mas é terrível para o computador executar (exigiria testar todas as possibilidades). O AFD, embora mais complexo e com mais estados, é uma "tabela de decisão" direta, o que resulta em uma execução muito mais veloz.

2. **Limites da Máquina (Contagem):**

Não, um Autômato Finito **não consegue contar**. Ele tem "memória finita" (apenas os estados). Para saber se ((())) está balanceado, a máquina precisaria lembrar de quantos parênteses abriu para exigir a mesma quantidade fechada. Como o número de parênteses pode ser infinito, precisaríamos de uma **Pilha** (memória extra), transformando o Autômato Finito em um **Autômato de Pilha**.

III. Respostas aos Exercícios Práticos

1. Desenho de Lógica (Palavra reservada if):

- **Estado S_0 (Inicial):** Aguarda entrada.
    - Lê 'i' → vai para **S_1**.
    - Lê qualquer outra coisa → **Erro Léxico**.
- **Estado S_1:**
    - Lê 'f' → vai para **S_2 (Aceitação)**.
    - Lê qualquer outra coisa → **Erro Léxico**.
- **Estado S_2 (Círculo Duplo):** Token IF reconhecido.

2. Lark Challenge (+=):

Para que o Lark reconheça += como um único token, ele deve ser definido antes ou ter prioridade. No Lark, terminais (tokens) mais específicos geralmente ganham a preferência no autômato gerado.

```
from lark import Lark

gramatica_mais_igual = """
    start: (ATRIBUICAO_SOMA | SOMA | IGUAL | NUMERO | ID)+

    ATRIBUICAO_SOMA: "+="
    SOMA: "+"
    IGUAL: "="
    ID: /[a-zA-Z]+/
    NUMERO: /[0-9]+/

    %import common.WS
    %ignore WS
"""

parser = Lark(gramatica_mais_igual)
texto = "valor += 10"
# Resultado: ID (valor), ATRIBUICAO_SOMA (+=), NUMERO (10)

print(f"Analisando: {texto}")
for token in parser.lex(texto):
    print(f"Token: {token.type:12} | Valor: {token.value}")
```

maquina_estados_2.py

Unidade 8: Análise Sintática – A Gramática do Código (Top-Down)

I. Questões de Múltipla Escolha

1. Resposta: C

- **Por que:** O Analisador Léxico já cuidou das palavras (tokens) na Unidade 6. Agora, o Sintático foca na **gramática**. Ele não traduz para máquina ainda, ele apenas garante que a "frase" faz sentido estrutural.

2. Resposta: B

- **Por que:** A característica do Top-Down é ser "preditivo". Ele olha para o símbolo inicial (start) e vai "quebrando" em regras menores (descendo) até encontrar os tokens reais (as folhas da árvore).

3. Resposta: B

- **Por que:** Em computação, recuo em árvores sempre indica **aninhamento**. Se atribuicao está recuada sob comando, significa que atribuicao é um tipo de comando. É a árvore criando seus galhos.

4. Resposta: B

- **Por que:** Esta é a base da **Precedência**. O computador resolve a árvore de baixo para cima. Se a multiplicação está mais profunda, o resultado dela (ex: 12) "nasce" primeiro e é entregue para o nó de cima (a soma) terminar a conta.

5. Resposta: C

- **Por que:** O parser Top-Down segue um "roteiro". Se o roteiro diz "aqui deve haver um nome" e aparece um número, o contrato é quebrado. O erro sintático é a forma de o compilador dizer: "Isso não estava no projeto original".

6. Resposta: C

- **Por que:** A recursividade (uma regra que chama a si mesma) é o que permite o **Infinito**. Sem ela, você só poderia ter um IF. Com ela, você pode ter um IF dentro de outro IF, criando árvores de qualquer profundidade.

7. Resposta: B

- **Por que:** Noam Chomsky definiu que linguagens de programação modernas (com parênteses e blocos aninhados) são **Livres de Contexto**. Elas precisam de uma pilha (memória) para serem validadas, o que o Lark faz brilhantemente.

8. Resposta: B

- **Por que:** O | é o "encruzilhada" do compilador. Quando o parser chega nesse símbolo, ele avalia o próximo token para decidir qual caminho (galho) seguir. É a **tomada de decisão sintática**.

9. Resposta: B

- **Por que:** Árvores com muitos nós intermediários inúteis (que só têm 1 filho) deixam o compilador lento e difícil de ler. O ? no Lark serve para "limpar" a árvore, mantendo apenas o que é essencial (AST - Árvore Sintática Abstrata).

10. Resposta: C

- **Por que:** Hierarquia significa que existem níveis de subordinação. Na sintaxe, um comando é subordinado ao programa, e um operador é subordinado a uma expressao. A árvore é a prova visual dessa hierarquia.

II. Respostas Reflexivas (Sugestão)

1. Na análise léxica, buscamos apenas padrões isolados (como identificar uma peça de quebra-cabeça). Na análise sintática, buscamos o encaixe (como montar o quebra-cabeça). Se a peça 2 vem antes da peça 1, o "projeto" da árvore não consegue se sustentar.

2. A flexibilidade geralmente simplifica a árvore (menos nós intermediários), mas exige que os delimitadores (como a indentação no Python ou as chaves no C) sejam muito claros para que o parser saiba onde um "galho" termina e outro começa.

III. Exercícios os Práticos

Exercício 1: O Compilador de Receitas

```
from lark import Lark

# Gramática Sequencial (Top-Down)
# A ordem das regras após os dois pontos (:) define a obrigatoriedade
gramatica_receita = """
    start: receita

    # A sequência obrigatória: primeiro 'pegar', depois 'misturar', por
fim 'cozinhar'
    receita: pegar_cmd misturar_cmd cozinhar_cmd

    pegar_cmd: "PEGAR" INGREDIENTE ";"
    misturar_cmd: "MISTURAR" ";"
    cozinhar_cmd: "COZINHAR" ";"

    INGREDIENTE: /[a-zA-Z_]+/

    %import common.WS
    %ignore WS
"""

parser = Lark(gramatica_receita)

# --- Teste 1: Receita Correta ---
print("--- Árvore da Receita Correta ---")
receita_ok = "PEGAR ovo; MISTURAR; COZINHAR;"
print(parser.parse(receita_ok).pretty())

# --- Teste 2: Erro de Ordem ---
print("\n--- Teste de Erro (Cozinhar antes de Pegar) ---")
receita_errada = "COZINHAR; PEGAR ovo; MISTURAR;"
try:
    parser.parse(receita_errada)
except Exception as e:
    print("❌ Erro Sintático: Você não pode cozinhar sem antes pegar os
ingredientes!")
```

Exercício 2: Calculadora de Áreas Hierárquica

A **Gramática (Lark)** cuida apenas da **Sintaxe** (a estrutura e a ordem). Ela é como a planta de uma casa: diz onde as paredes devem ficar, mas não constrói a casa. Para realizar o cálculo real, precisamos da **Semântica**, que no Lark implementamos usando o **Transformer**.

```
from lark import Lark, Transformer, v_args
import math

# --- 1. A GRAMÁTICA (SINTAXE) ---
gramatica_areas = """
    ?start: calculo
    calculo: "AREA" forma

    # Ramificação da Árvore de Decisão
    forma: quadrado | circulo

    quadrado: "QUADRADO" NUMBER ";"
    circulo: "CIRCULO" NUMBER ";"

    %import common.NUMBER
    %import common.WS
    %ignore WS
"""

# --- 2. O TRANSFORMER (SEMÂNTICA / CÁLCULO) ---
class CalculadorArea(Transformer):

    @v_args(inline=True)
    def quadrado(self, valor):
        # Converte o token NUMBER para float e calcula Lado * Lado
        lado = float(valor)
        area = lado ** 2
        return f"Área do Quadrado: {area:.2f}"

    @v_args(inline=True)
    def circulo(self, valor):
        # Calcula PI * Raio^2
        raio = float(valor)
        area = math.pi * (raio ** 2)
        return f"Área do Círculo: {area:.2f}"

    def calculo(self, children):
        # Retorna o resultado vindo da regra 'forma'
        return children[0]

# --- 3. EXECUÇÃO ---
parser = Lark(gramatica_areas)
calc = CalculadorArea()

# Testando os dois caminhos da Árvore de Decisão
print("--- RESULTADOS DOS CÁLCULOS ---")

# Caminho 1: Quadrado
quadrado = "AREA QUADRADO 5;"
arvore_q = parser.parse(quadrado)

print(f"--- Árvore de Decisão: '{quadrado}' ---")
print(parser.parse(quadrado).pretty())
print("--- Detalhamento:")
print(calc.transform(arvore_q))

# Caminho 2: Círculo

circulo = "AREA CIRCULO 10;"
arvore_c = parser.parse(circulo)

print(f"--- Árvore de Decisão: '{circulo}' ---")
print(parser.parse(quadrado).pretty())
print("--- Detalhamento:")
print(calc.transform(arvore_c))
```

Revisão – Unidade 1 até Unidade 8

🧠 Explicação do Gabarito (Unidades 1 a 8)

Bloco A: Gramáticas e Árvores (Teoria de Chomsky)

- **Questão 1 (B):** Gramáticas Livres de Contexto (GLC) são ideais porque conseguem lidar com estruturas que exigem um "balanço", como parênteses abertos e fechados ou blocos de código aninhados, algo que gramáticas regulares não fazem.
- **Questão 2 (C):** Em formalismo matemático, **Σ** (Sigma) representa o alfabeto, que no contexto de compiladores são os símbolos terminais (tokens) que o programador digita.
- **Questão 3 (B):** A Árvore de Derivação não é apenas um desenho; sua estrutura vertical define o que é executado primeiro (precedência). O que está mais longe da raiz (fundo da árvore) tem prioridade.
- **Questão 4 (C):** A análise LL (Top-Down) lê da esquerda para a direita e expande o não-terminal mais à esquerda, tentando "prever" a árvore de cima para baixo.
- **Questão 5 (B):** A ambiguidade é um erro de design porque o computador é uma máquina determinística. Se uma gramática permite dois significados para a mesma linha, o compilador não sabe qual instrução de máquina gerar.

Bloco B: Arquitetura e Fluxo do Compilador

- **Questão 6 (E):** A modularidade é a alma da engenharia de compiladores. Separar o Front-End (análise) do Back-End (síntese) permite que você crie um compilador para uma nova CPU reaproveitando toda a lógica da linguagem.
- **Questão 7 (C):** A Semântica cuida do "sentido". Enquanto a sintaxe vê se a frase está bem escrita, a semântica vê se você não está tentando, por exemplo, somar um int com um string.
- **Questão 8 (A):** Programas rodam em "espaço de usuário" e não tocam no hardware. Eles precisam pedir permissão ao Kernel através de *System Calls* para tarefas como escrever no disco ou mostrar algo na tela.
- **Questão 9 (B):** O compilador faz o "trabalho pesado" uma única vez, gerando um arquivo pronto. O interpretador precisa traduzir cada linha toda vez que o programa roda.
- **Questão 10 (D):** Java e C# usam Bytecode para garantir portabilidade. O compilador gera o Bytecode e a Máquina Virtual (JVM/.NET) o interpreta em qualquer sistema operacional.

Bloco C: Ferramentas e Automação

- **Questão 11 (A):** Escrever analisadores à mão é complexo e gera bugs de memória. Ferramentas como AntLR e Lark garantem um motor de parsing matematicamente correto.
- **Questão 12 (B):** O diferencial do $LL(*)$ é não ficar preso a um número fixo de tokens à frente. Ele "navega" pela frase o quanto for preciso para decidir qual regra aplicar.
- **Questão 13 (E):** EBNF é a linguagem padrão para descrever gramáticas. Ela usa símbolos como *, + e ? para definir repetições e opcionalidade.
- **Questão 14 (D):** Autômatos Finitos (AF) são máquinas de estados simples sem memória de pilha, por isso só reconhecem Linguagens Regulares (Tipo 3).
- **Questão 15 (B):** O Rust impede erros de ponteiro e invasões de memória (comuns em C) através do *Borrow Checker*, que faz essas verificações durante a compilação.

Bloco D: Análise Léxica e Autômatos

- **Questão 16 (B):** Um Token precisa de uma categoria (ex: IDENTIFICADOR) e pode ter um atributo (ex: o nome real da variável) para ser usado na Tabela de Símbolos.
- **Questão 17 (C):** Lexema é o "bruto" (ex: 123). Token é o "processado" (ex: LITERAL_INTEIRO).
- **Questão 18 (A):** Em Expressões Regulares, o + indica repetição (1 ou mais vezes). É o que permite que 1, 10 ou 1000 sejam todos reconhecidos pelo mesmo padrão.
- **Questão 19 (B):** O *Maximal Munch* evita confusões. Se o compilador vê + e depois outro +, ele prefere formar o token ++ (maior) em vez de dois tokens + separados.
- **Questão 20 (D):** No AFD, o caminho é sempre único e direto. Isso garante uma performance de O(n), processando o texto na velocidade da leitura.

Bloco E: Análise Sintática Top-Down

- **Questão 21 (C):** A sintaxe é a "cola" que une as palavras. Ela garante que um if tenha uma condição e que uma atribuição tenha um valor.
- **Questão 22 (B):** A análise Top-Down é uma tentativa de confirmar uma hipótese: "Eu acho que isto é um programa, vamos ver se os tokens confirmam isso".

- **Questão 23 (B):** A profundidade na árvore define a ordem de avaliação. Em 2 + 3 * 4, o nó de multiplicação fica mais fundo para que seu resultado alimente a soma acima.
- **Questão 24 (C):** O Lark usa o ? para simplificar a árvore (AST). Se uma regra tem apenas um filho, ele remove o nó pai desnecessário, deixando a árvore mais limpa para o programador.
- **Questão 25 (E):** Um erro sintático é uma quebra de expectativa. Se a gramática espera um ; e recebe um ID, o parser trava e reporta a falha.

Bloco F: Projeto Prático e Semântica

- **Questão 26 (B):** Erros contextuais (Semântica) dependem do histórico. Você só pode usar o que alocou. O compilador rastreia esse estado para evitar falhas de segmentação.
- **Questão 27 (A):** Gramáticas Livres de Contexto não olham para os lados. Gramáticas Sensíveis ao Contexto verificam se o ambiente (tokens vizinhos) permite aquela transformação.
- **Questão 28 (C):** O Transformer é onde a "mágica" do Python acontece. Ele pega a árvore estática e a transforma em valores reais ou executa validações lógicas.
- **Questão 29 (B):** Se a validade de um campo (número do RG) depende do valor de outro campo (estado SP/RJ), isso é uma regra de contexto (Semântica).
- **Questão 30 (D):** A Tabela de Símbolos é o "caderno de anotações". Ela guarda onde cada variável foi criada e qual seu tipo, sendo consultada em quase todas as fases.

Unidade 9: Análise Sintática Avançada (Bottom-Up)

Questões de Múltipla Escolha

Questão 1: C – A III está incorreta porque a AST é uma simplificação (abstração), não é idêntica à árvore de derivação concreta.

Questão 2: C – A recursão à esquerda é o "calcanhar de Aquiles" do *Top-Down*, mas é o comportamento natural e eficiente do *Bottom-Up*.

Questão 3: D – Por definição, a análise ascendente começa nos tokens (folhas) e sobe até o símbolo inicial (raiz).

Questões Discursivas

Questão 4: A redução substitui um corpo de prova por um não-terminal; nesse momento, o compilador instancia um objeto (nó) na memória cujos ponteiros apontam para os elementos reduzidos. A AST facilita a análise semântica pois remove o ruído sintático, permitindo que o verificador de tipos percorra uma estrutura puramente lógica.

Questão 5: O *Top-Down* tenta prever a árvore (expansão), enquanto o *Bottom-Up* a constrói por evidência (redução). Analisadores ascendentes são matematicamente mais complexos (exigem grandes tabelas de estados), por isso ferramentas de automação são essenciais para garantir a correção que seria difícil de alcançar manualmente.

Unidade 10: Projeto Prático II – Gerador Sintático

Questões de Múltipla Escolha

Questão 1 - B: O léxico atua como um "garçom" que entrega os ingredientes (tokens) prontos para o cozinheiro (parser).

Questão - C: Na notação EBNF, o asterisco representa o Fecho de Kleene (zero ou mais).

Questões Discursivas

Questão 3: A gramática garante que a estrutura mínima necessária para o hardware processar a função (endereço de retorno, parâmetros, escopo) esteja presente e organizada. Se a estrutura estiver errada, o compilador barra o erro antes mesmo de o programa rodar, garantindo segurança.

Questão 4: O analisador gera uma exceção de erro sintático (*Syntax Error*), interrompe a tradução e, idealmente, informa ao usuário a linha, a coluna e qual símbolo era esperado naquele ponto.

Unidade 11: Análise Semântica – O Significado

Questões de Múltipla Escolha

Questão 1 - C: A frase está bem escrita (Sintaxe OK), mas os tipos são incompatíveis (Semântica Errada).

Questão 2 - C: O escopo é o que impede que uma variável i dentro de um for sobrescreva acidentalmente uma variável i importante fora dele.

Questões Discursivas

Questão Q3: Porque as GLCs focam na estrutura (ordem dos tokens), mas não têm "memória" de longo prazo para saber o que foi declarado linhas atrás. A Semântica precisa da Tabela de Símbolos para ter esse contexto.

Questão 4: O compilador consultaria a Tabela de Símbolos em busca do nome da função. Como o nome não estaria presente na tabela, o analisador semântico emitiria um erro de "Identificador Não Definido".

Unidade 12: Código Intermediário – A "Língua de Esperanto" da TI

Questões de Múltipla Escolha

Gabarito: A: Usando a fórmula de eficiência de arquitetura modular, em vez do produto **M X N (4 X 3 = 12)**, o uso da IR reduz o esforço para a soma **M + N (4 + 3 = 7).** São necessários 4 *Front-Ends* (um para cada linguagem) e 3 *Back-Ends* (um para cada arquitetura de hardware).

Gabarito: B: O código intermediário apresenta uma verificação de saída no topo (`if a >= b goto L2`), um corpo de instrução que altera o valor avaliado (`a = a + 1`) e um desvio incondicional de retorno ao topo (`goto L1`). Esse comportamento caracteriza nativamente um laço de repetição condicional contínua (*while loop*).

Questões Discursivas

Questão 3: As CPUs reais operam com base em uma arquitetura de registradores limitada, onde instruções de máquina executam uma operação por vez sobre um número fixo de operandos (geralmente dois registradores de entrada e um de destino). Uma expressão complexa com múltiplos operadores aninhados exige o cálculo de subexpressões e o armazenamento de resultados temporários. O Código de Três Endereços (TAC) realiza exatamente esse papel de linearização, quebrando a complexidade da árvore sintática em passos sequenciais que condizem com a capacidade física do hardware.

Questão 4: O "Código Morto" é qualquer trecho de código que nunca será executado pelo fluxo do programa ou cujo resultado não altera o estado de nenhuma variável de saída. Para identificar isso em estruturas condicionais, o compilador avalia expressões constantes em tempo de compilação (técnica conhecida como *Constant Folding*). Se a condição de um `if` for avaliada estaticamente como falsa (ex: `if (false)` ou `if (5 > 10)`), o analisador de fluxo de controle na IR identifica o bloco interno como inacessível e o remove da árvore/fluxo antes de gerar o binário final.

Unidade 13: Onde o Código Vive – Ambientes de Execução

Questões de Múltipla Escolha

Questão 1: B: Variáveis de controle local possuem escopo delimitado e tamanho fixo conhecido, sendo alocadas no topo da Pilha (*Stack*). Já matrizes dinâmicas e coleções cujo tamanho varia em tempo de execução exigem alocação flexível, sendo criadas no Heap.

Questão 2 C: Para garantir que o grafo de objetos não mude enquanto está sendo mapeado, os algoritmos tradicionais de *Mark-and-Sweep* suspendem a execução das instruções do usuário, criando uma latência conhecida como *Stop-the-World*, o que representa um desafio para sistemas de tempo real crítico.

Questões Discursivas

Questão 3: O aluno deve explicar que cada chamada de função empilha um novo *Stack Frame* contendo variáveis locais e endereços de retorno. Sem uma condição de parada, a função continuará alocando quadros consecutivamente de forma linear para cima. Como a Pilha possui um limite de tamanho fixado pelo sistema operacional, esse crescimento contínuo esgota o espaço delimitado do segmento de memória, causando a falha crítica de hardware/software denominada *Stack Overflow*.

Questão 4: O aluno deve apontar que a eliminação do GC remove o consumo extra de processamento (CPU) e memória necessário para rodar o motor de monitoramento em segundo plano. Sem o GC, o programa não sofre as pausas imprevisíveis de latência (*Stop-the-World*), o executável final se torna muito menor e o consumo de memória RAM cai drasticamente, tornando o ambiente de execução ideal para sistemas embarcados, drivers de hardware e aplicações de alta performance.

Unidade 14: Geração e Otimização de Código

Questões de Múltipla Escolha

Questão 1: B*:* Os registradores da CPU são escassos. A alocação e atribuição de registradores usa algoritmos complexos (como a coloração de grafos) para maximizar o uso dessas memórias internas rápidas, diminuindo o acesso à lentidão da memória RAM.

Questão 2: B *:* Como o valor de `(largura_tela / 2)` não se altera em nenhuma das 1000 repetições do laço, efetuar essa divisão a cada iteração é um desperdício de processamento. O compilador joga o cálculo para fora do loop.

Questões Discursivas

Questão 3: A operação de multiplicação aritmética tradicional (*) consome múltiplos ciclos de clock da CPU para ser resolvida pelas portas lógicas da Unidade Lógica e Aritmética (ULA). Ao identificar que o multiplicador é uma potência de 2 ($8 = 2^3$), o compilador pode aplicar a redução de força, substituindo a multiplicação por uma instrução de deslocamento de bits para a esquerda em 3 posições (operação conhecida como *Bitwise Left Shift*). No nível de hardware, o deslocamento de bits é executado de forma muito mais rápida (geralmente em apenas um ciclo de clock), otimizando o tempo de execução.

Questão 4: As otimizações independentes de máquina focam na lógica algorítmica pura e na estrutura do código na camada intermediária (IR); exemplos incluem a eliminação de código morto, eliminação de subexpressões comuns e movimentação de código em laços. Já as otimizações dependentes de máquina atuam diretamente sobre as características físicas do processador alvo, envolvendo a escolha de instruções específicas do conjunto da CPU (ISA), aproveitamento do *pipeline* de hardware específico para evitar bolhas de execução e alocação ótima dos registradores físicos disponíveis no chip.

Revisão Para Prova

1-C | 2-D | 3-B | 4-C | 5-B | 6-C | 7-B | 8-B | 9-C | 10-C

11-C | 12-B | 13-C | 14-C | 15-B | 16-C | 17-B | 18-C | 19-B | 20-C

21-C | 22-B | 23-C | 24-A | 25-C | 26-B | 27-C | 28-C | 29-A | 30-B