CENTRO UNIVERSITÁRIO FMU LEVEL UP

COMPILADORES

https://portal.fmu.br

COMPILADORES

Público-alvo: Estudantes do Centro Universitário FMU.

Objetivo: Esta apostila foi cuidadosamente estruturada para guiar o estudante através da fascinante jornada de transformação do código-fonte em instruções executáveis, fundamentando-se na tríade teórica de **Aho et al. (2008)**, **Menezes (2011)** e **Diverio (2011)**. Iniciamos explorando o "Front-End" do compilador, onde a **Análise Léxica** e a **Análise Sintática** convertem sequências de caracteres em estruturas lógicas, utilizando Gramáticas Livres de Contexto e Árvores de Derivação para validar a integridade do código. Esta etapa é o coração da disciplina, pois permite ao desenvolvedor compreender como a precedência matemática e a estrutura das linguagens são rigorosamente definidas por formalismos matemáticos.

Avançando para a camada intermediária e final, abordamos a **Análise Semântica** e a **Geração de Código**, fases em que o significado do programa é verificado e traduzido para linguagens de baixo nível. Estudaremos como a Tabela de Símbolos e as Verificações de Tipos garantem que a lógica do programador seja coerente, culminando na produção de código otimizado que o hardware possa processar com eficiência. O domínio dessas etapas capacita o aluno não apenas a usar linguagens de programação, mas a projetar suas próprias ferramentas de tradução e automação, elevando seu patamar técnico no mercado de desenvolvimento.

Por fim, esta apostila mergulha nos limites da computação, explorando a **Hierarquia de Chomsky** e a **Máquina de Turing** para entender a computabilidade e a decidibilidade. Ao estudar problemas que não possuem solução algorítmica, o futuro profissional desenvolve uma visão crítica e analítica sobre a complexidade dos sistemas. Esta introdução serve como o primeiro passo para consolidar o pensamento abstrato indispensável para criar soluções tecnológicas robustas, unindo o rigor da teoria acadêmica à prática moderna de engenharia de software com Python e outras tecnologias de ponta.

Professor: Robson Carmo, mestre pelo programa de mestrado profissional em Gestão e Tecnologia em Sistemas Produtivos do Centro Paula Souza, concluiu 3 MBAs (Inovação e Transformação Digital, Gestão de Empresarial, Engenharia de Software). Instrutor de certificações da SAFE, auto e coautor de vários livros e ebooks.

Linkedin: https://www.linkedin.com/in/robsonanderson/
Lattes: http://lattes.cnpq.br/8753112683856964
Orcid: https://orcid.org/0009-0002-7102-4993

LLM utilizada: Gemini 3 – Fevereiro de 2026.

Módulos e Conteúdo

Unidade 1: O Coração das Linguagens – Gramáticas e Árvores

- 1.1 Introdução às Gramáticas Livres de Contexto (GLC): O que são e por que são a base das linguagens de programação.
- 1.2 Regras de Produção: Como definir "frases" válidas em uma linguagem (ex: como o computador sabe que x = 1 + 2 está correto).
- 1.3 Árvores de Derivação: Representação visual da estrutura de um código.
- 1.4 Derivações à Esquerda e à Direita: Como o compilador escolhe o caminho para entender seu código e o impacto disso na performance.

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

- 2.1 Evolução Histórica: Das linguagens de máquina aos compiladores modernos.
- 2.2 Arquitetura do computador (CPU, Memória, Device, Kernel e Programas)
- 2.3 A Estrutura em Fases: Por que dividimos o compilador em "caixas" menores?
- 2.4 O Front-End: Foco na análise (entender o que o programador escreveu).
- 2.5 As Três Análises Principais: Introdução rápida à Léxica, Sintática e Semântica.

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

- 3.1 O Back-End: Foco na síntese (traduzir o que foi entendido para a língua do computador).
- 3.2 Compilador vs. Interpretador: * *Exemplo:* O Compilador é como traduzir um livro inteiro antes de ler; o Interpretador é como um tradutor simultâneo em uma palestra.
- 3.3 Estudo de Caso: O caminho completo de um comando "Print" do teclado até a tela.

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

- 4.1 Automação na Criação: Por que não escrevemos tudo do zero?
- 4.2 Ferramentas de Mercado: Introdução ao Lex/Yacc (C) e AntLR (Java).
- 4.3 Vantagens Práticas: Ganho de produtividade e redução de erros humanos.
- 4.4 Exemplos Práticos.

Unidade 5: Análise Léxica – Identificando Palavras

- 5.1 O que são Tokens: A menor unidade com significado (palavras-chave, números, símbolos).
- 5.2 Especificação com Expressões Regulares: Como descrever padrões de texto.
- 5.3 Reconhecimento na Prática: Como o compilador ignora espaços em branco e comentários.

Unidade 6: Máquinas de Estados – Autômatos Finitos

- 6.1 Autômatos Finitos: Criando diagramas que "decidem" se um token é válido.
- 6.2 Conversão de Padrão para Máquina: Transformando uma regra de texto em um fluxo lógico de decisão.

Unidade 7: Projeto Prático I – Gerador Léxico

- 7.1 Planejamento do Analisador: Definindo os tokens de nossa própria mini-linguagem.
- 7.2 Implementação com Ferramentas: Mão na massa utilizando LARK para gerar o primeiro componente do compilador.

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

- 8.1 O Papel do Analisador Sintático: Verificar se a ordem das palavras faz sentido.
- 8.2 Análise Descendente (Top-Down): Começar do objetivo geral e descer aos detalhes.

Unidade 9: Análise Sintática Avançada (Bottom-Up)

- 9.1 Análise Ascendente (Bottom-Up): Partir das palavras individuais para construir a estrutura completa.
- 9.2 Comparação de Estratégias: Quando usar cada uma e por que a análise Bottom-Up é geralmente mais poderosa para linguagens complexas.

Unidade 10: Projeto Prático II – Gerador Sintático

- 10.1 Construção da Gramática: Definindo as regras de estrutura (ex: como declarar uma função).

- **10.2 Integração:** Unindo o analisador léxico ao sintático.

Unidade 11: Análise Semântica – O Significado

- **11.1 Verificação de Coerência:** O código pode estar escrito certo, mas faz sentido?
- **11.2 Tabela de Símbolos:** O "caderno de anotações" do compilador sobre as variáveis.

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

- **12.1 Por que um código intermediário?** Facilitar a tradução para diferentes tipos de computadores.
- **12.2 Verificação de Tipos e Fluxo:** Garantir que o if e o while funcionem logicamente.

Unidade 13: Onde o Código Vive – Ambientes de Execução

- **13.1 Organização da Memória:** Diferença entre Pilha (Stack) e Heap.
- **13.2 Coleta de Lixo (Garbage Collection):** Como linguagens como Java e Python limpam a memória automaticamente.

Unidade 14: Geração e Otimização de Código

- **14.1 Linguagem Objeto (Assembly/Binário):** O destino final do nosso código.
- **14.2 Otimização:** Como o compilador torna seu programa mais rápido sem você precisar mudar uma linha de código.
- **14.3 Visão Geral do Ciclo de Vida:** Revisão de todo o processo, do teclado ao processador.

Unidade 1: O Coração das Linguagens – Gramáticas e Árvores

1.1 Introdução às Gramáticas Livres de Contexto (GLC)

1.1.1 Histórico e Evolução do Contexto

Para compreendermos as Gramáticas Livres de Contexto (GLC), precisamos olhar para a década de 1950. Naquela época, o grande desafio da computação era encontrar uma forma de comunicação que não fosse nem tão ambígua quanto a linguagem humana (natural), nem tão limitada quanto a linguagem de máquina puramente numérica.

A virada de chave ocorreu com o trabalho de Noam Chomsky, que buscava modelar as linguagens naturais. Ele propôs uma hierarquia de classes de gramáticas. Embora o objetivo inicial fosse a linguística, a Ciência da Computação rapidamente percebeu que a **Classe 2 (Linguagens Livres de Contexto)** era o "ponto ideal" para definir linguagens de programação.

Como destaca **Aho et al. (2008)**, o desenvolvimento da notação BNF (Backus-Naur Form) para descrever a sintaxe do ALGOL 60 foi o marco que uniu a teoria das GLC à prática da construção de compiladores. Sem esse formalismo, seria impossível garantir que um código escrito em um computador se comportasse da mesma forma em outro.

1.1.2 Por que são a base das linguagens de programação?

As GLCs permitem definir estruturas recursivas e aninhadas, como parênteses dentro de parênteses ou blocos if dentro de if. As gramáticas regulares (mais simples) não conseguem "contar" ou equilibrar esses elementos.

Menezes (2011) reforça que a simplicidade das GLCs reside no fato de que a substituição de um símbolo não depende dos símbolos ao seu redor (o contexto), o que torna a análise sintática computacionalmente eficiente e previsível.

1.2 Regras de Produção: Definindo a Validade de "Frases"

1.2.1 A Formalização Matemática

Uma gramática **G** é formalmente definida pela quádrupla **G = (V, Σ, P, S)**, onde:

- **V (Variáveis/Não-terminais):** Símbolos abstratos que representam estruturas (ex: <Comando>, <Expressão>).
- **Σ (Alfabeto/Terminais):** Os símbolos reais que compõem o código (ex: int, +, (, identifier).
- **P (Produções):** O conjunto de regras de substituição.
- **S (Símbolo Inicial):** O ponto de partida da linguagem.

1.2.2 Como o computador sabe que x = 1 + 2 está correto?

Através das **regras de produção**, o compilador tenta "reconstruir" a frase fornecida. Considere as regras simplificadas:

1. <Atribuição> → <id> = <Expr>
2. <Expr> → <Expr> + <Expr> | <num>
3. <id> → x | y
4. <num> → 1 | 2 | 3

Para validar **x = 1 + 2**, o compilador inicia em <Atribuição>. Ele vê que x é um <id> válido, encontra o símbolo = e então tenta derivar 1 + 2 a partir de <Expr>. Como existe uma regra que permite somar duas expressões, e cada expressão pode se tornar um número, o código é aceito. Se o programador escrevesse x = + 1 2, a gramática falharia ao tentar encontrar uma regra que começasse com + logo após o =, resultando no famoso **Erro Sintático**.

Como afirmam **Diverio e Menezes (2011)**, "as produções definem a sintaxe da linguagem, estabelecendo uma relação de causa e efeito entre a abstração do projeto e a realidade da implementação".

1.3 Árvores de Derivação: A Geometria do Pensamento

1.3.1 Representação Visual e Hierárquica

Enquanto as regras de produção são textuais, a **Árvore de Derivação** (ou *Parse Tree*) é a prova geométrica de que uma sentença pertence a uma linguagem. Cada nó interno é um não-terminal, e cada folha é um terminal.

A importância da árvore reside no fato de que ela dita a **precedência**. Em uma linguagem onde a multiplicação tem prioridade sobre a soma, a árvore deve ser construída de forma que os nós da multiplicação estejam "mais longe" da raiz (ou seja, sejam resolvidos primeiro na subida da árvore).

Exemplo Profundo:

Considere a expressão 3 * 4 + 5.

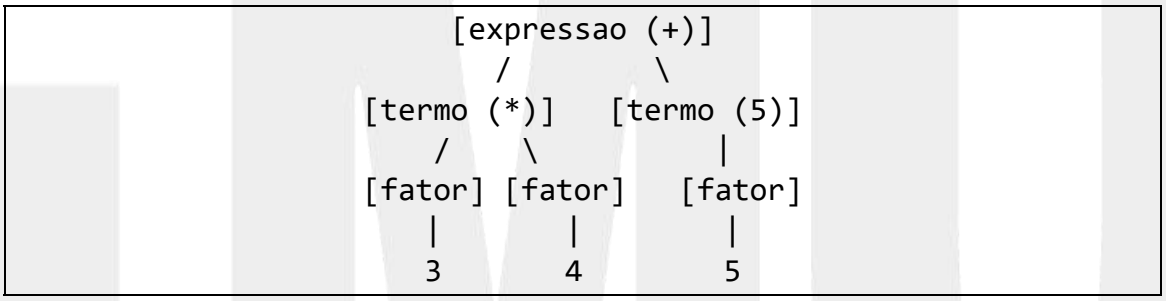
- Se a árvore agrupar (3 * 4) e depois somar 5, ela respeita a matemática padrão.
- Se a árvore agrupar 3 * (4 + 5), ela altera o sentido do código.

Explicação Técnica – 3 * 4 + 5

De acordo com **Aho et al. (2008)** no livro *"Compiladores: Princípios, Técnicas e Ferramentas"*, a estrutura hierárquica garante que a semântica da frase seja respeitada:

1. **Nível Inferior (Multiplicação):** Os números **3** e **4** são filhos do operador *. O compilador calcula 3 * 4 = 12.
2. **Nível Superior (Soma):** O resultado anterior (**12**) torna-se um operando para o nó +, que tem o número **5** como seu outro filho.
3. **Resultado Final:** 12 + 5 = 17.

Representação Visual da Árvore de Decisão



Aho et al. (2008) explicam que o analisador sintático não apenas valida o código, mas cria essa estrutura intermediária que será usada posteriormente pela análise semântica para gerar código de máquina.

1.4 Derivações à Esquerda, à Direita e Performance

1.4.1 O Caminho da Análise

O processo de substituir não-terminais pode seguir ordens diferentes:

1. **Derivação Mais à Esquerda (Leftmost):** Em cada passo, o não-terminal mais à esquerda é substituído. É a base para analisadores **Top-Down (LL)**.
2. **Derivação Mais à Direita (Rightmost):** O não-terminal mais à direita é substituído. É a base para analisadores **Bottom-Up (LR)**.

1.4.2 Impacto na Performance e Ambiguidade

A escolha entre essas derivações não é apenas estética. Analisadores que utilizam derivação à direita (como os da família LR) conseguem lidar com uma gama maior de gramáticas sem entrar em loops infinitos, sendo geralmente mais potentes, porém mais complexos de implementar manualmente.

A ambiguidade ocorre quando uma gramática permite duas árvores diferentes para a mesma frase. Segundo **Menezes (2011)**, "a ambiguidade é um defeito de projeto na gramática de uma linguagem de programação, pois um compilador não pode ter dúvidas sobre o que o programador quis dizer".

Exercício Prático em Python: Resolvendo x = 5 * 12 / 6

Nesta unidade, vimos que um compilador não lê apenas texto; ele interpreta estruturas. O código abaixo reflete as quatro fases que discutimos: **Léxica** (identificação de tokens), **Sintática** (montagem da árvore), **Semântica** (conversão de tipos) e **Síntese** (cálculo final).

```python
from lark import Lark, Transformer

# ========================================
# 1. DEFINIÇÃO DA GRAMÁTICA (GLC)
# ========================================
# Aqui aplicamos a Hierarquia de Chomsky.
# A ordem das regras define a precedência matemática.
gramatica_fmu_v7 = """
    ?start: atribuicao

    # Uma atribuição exige um ID, o sinal de '=' e uma expressão à direita.
    atribuicao: ID "=" expressao

    # ?expressao: Nível mais baixo de precedência (Soma/Subtração).
    # O '?' diz ao Lark: "Se houver apenas um termo, não crie um nó extra na árvore".
    ?expressao: termo (SOMA termo)*

    # ?termo: Nível médio de precedência (Multiplicação/Divisão).
    ?termo: fator (MULT fator)*

    # ?fator: Maior precedência. Resolve números, variáveis ou parênteses primeiro.
    ?fator: NUMBER           -> numero
          | "(" expressao ")"
          | ID               -> variavel

    # Definição dos Terminais (Tokens)
    SOMA: "+" | "-"
    MULT: "*" | "/"

    # Importações de padrões comuns: Identificadores e Números
    %import common.CNAME -> ID
    %import common.NUMBER
    %import common.WS
    %ignore WS
"""

# ========================================
# 2. TRANSFORMER (ANÁLISE SEMÂNTICA)
# ========================================
# O Transformer percorre a árvore de baixo para cima (Bottom-Up).
class CalculadoraFMU(Transformer):

    # Executada quando o Lark encontra um número (Terminal)
    def numero(self, tokens):
        # Converte o texto (string) em um número real (float)
        return float(tokens[0])
```

```python
    # Executada para nomes de variáveis
    def variavel(self, tokens):
        return str(tokens[0])

    # Executada para resolver Multiplicações e Divisões
    def termo(self, tokens):
        # O acumulador 'res' inicia com o primeiro valor do termo
        res = tokens[0]
        i = 1
        # Percorre os tokens em pares: [operador, valor]
        while i < len(tokens):
            op = tokens[i]
            val = tokens[i+1]
            if op == '*':
                res *= val
            elif op == '/':
                res /= val
            i += 2
        return res

    # Executada para resolver Somas e Subtrações
    def expressao(self, tokens):
        res = tokens[0]
        i = 1
        while i < len(tokens):
            op = tokens[i]
            val = tokens[i+1]
            if op == '+':
                res += val
            else:
                res -= val
            i += 2
        return res

    # Regra final: monta o dicionário de saída
    def atribuicao(self, tokens):
        return {str(tokens[0]): tokens[1]}

# ==========================================
# 3. EXECUÇÃO E TESTE
# ==========================================
# Criamos o objeto analisador com base na nossa gramática
parser = Lark(gramatica_fmu_v7, start='atribuicao')

# Nossa sentença de teste (O "Código Fonte")
equacao = "x = 5 * 12 / 6"

try:
    # FASE SINTÁTICA: Gera a Árvore de Derivação
    arvore = parser.parse(equacao)
    print("--- 1. ÁRVORE SINTÁTICA (ESTRUTURA) ---")
    print(arvore.pretty())

    # FASE SEMÂNTICA/SÍNTESE: Percorre a árvore e gera o valor
    resultado = CalculadoraFMU().transform(arvore)
    print("--- 2. RESULTADO DO PROCESSAMENTO ---")
    print(resultado)

except Exception as e:
    print(f"Erro no processo de compilação: {e}")
```

arvore-decisao.py

Saída para: `5 * 12 / 6`

```
--- 1. ÁRVORE SINTÁTICA (ESTRUTURA) ---
atribuicao
  x
  termo
    numero      5
    *
    numero      12
    /
    numero      6

--- 2. RESULTADO DO PROCESSAMENTO ---
{'x': 10.0}
```

Saída para: `5 / 12 * 6`

```
--- 1. ÁRVORE SINTÁTICA (ESTRUTURA) ---
atribuicao
  x
  termo
    numero      5
    /
    numero      12
    *
    numero      6

--- 2. RESULTADO DO PROCESSAMENTO ---
{'x': 2.5}
```

Saída para: `(5 * 12) / 2`

```
--- 1. ÁRVORE SINTÁTICA (ESTRUTURA) ---
atribuicao
  x
  termo
    termo
      numero    5
      *
      numero    12
    /
    numero      2

--- 2. RESULTADO DO PROCESSAMENTO ---
{'x': 30.0}
```

I. Questões de Múltipla Escolha

1. Considere a definição formal de uma GLC. Se um símbolo A pertence ao conjunto V, isso significa que:

A) A é um token final que aparecerá no código executável.

B) A é uma variável que deve ser substituída por outras sequências de símbolos.

C) A é um erro de sintaxe detectado pelo Front-End.

D) A é um caractere reservado como ponto e vírgula.

2. A notação BNF (Backus-Naur Form) é amplamente utilizada para descrever:

A) A alocação de memória no Heap.

B) A sintaxe das linguagens de programação através de regras de produção.

C) O tempo de execução de um algoritmo de ordenação.

D) A interface gráfica de um sistema operacional.

3. Qual a principal característica que diferencia uma Gramática Livre de Contexto de uma Gramática Regular?

A) A capacidade de lidar com recursividade à esquerda e estruturas aninhadas.

B) O fato de ser lida apenas da direita para a esquerda.

C) A impossibilidade de gerar números inteiros.

D) O uso obrigatório de compiladores Java.

4. Em uma árvore de derivação, o que representam os nós folha?

A) As regras de produção que não foram utilizadas.

B) Os símbolos terminais (tokens) da sentença analisada.

C) Os símbolos iniciais de cada subprograma.

D) Os ponteiros de memória para a tabela de símbolos.

5. Se uma gramática produz duas árvores de derivação distintas para a expressão 2 + 3 * 4, dizemos que ela é:

A) Determinística.

B) Otimizada para performance.

C) Ambígua.

D) Uma gramática de tipo 3 na hierarquia de Chomsky.

6. Analisadores sintáticos do tipo LL(k) utilizam qual tipo de derivação?

A) Derivação mais à direita.

B) Derivação aleatória baseada em heurísticas.

C) Derivação mais à esquerda.

D) Não utilizam derivações, apenas busca exaustiva.

7. Sobre o símbolo inicial S de uma gramática, é correto afirmar que:

A) Ele é sempre um símbolo terminal.

B) Ele representa a menor unidade da linguagem.

C) Ele é o ponto de partida para qualquer derivação da linguagem.

D) Ele só pode aparecer do lado direito das regras de produção.

8. O que caracteriza uma derivação "mais à direita"?

A) O compilador ignora todos os erros do lado esquerdo.

B) Em cada passo, o não-terminal posicionado mais ao fim da sequência atual é substituído.

C) O código é traduzido diretamente para binário sem passar por intermediários.

D) A árvore de derivação cresce apenas para o lado direito da página.

9. De acordo com Aho et al. (2008), por que as GLCs são preferidas para a análise sintática?

A) Porque são fáceis de traduzir para linguagem de montagem (Assembly).

B) Porque possuem um equilíbrio entre expressividade e eficiência de reconhecimento por autômatos de pilha.

C) Porque ocupam menos espaço em disco rígido.

D) Porque foram criadas especificamente para a linguagem C.

10. Qual a função das regras de produção na fase de "Front-End" de um compilador?

A) Gerar o arquivo .EXE final.

B) Definir a estrutura lógica que permite ao analisador validar se o código segue as normas da linguagem.

C) Alocar espaço na memória RAM para as variáveis globais.

D) Realizar a coleta de lixo (Garbage Collection).

II. Perguntas Reflexivas

1. Analisando a evolução das linguagens, por que você acredita que a ambiguidade em uma gramática de linguagem de programação é considerada um erro de projeto, enquanto na linguagem humana ela é uma ferramenta de expressão (poesia, ironia)?

2. Como a estrutura de uma árvore de derivação pode influenciar a forma como um processador executa cálculos matemáticos?

III. Exercícios Práticos

1. **Modelagem:** Crie uma pequena gramática (3 regras) que seja capaz de gerar números pares de dois dígitos (ex: 10, 22, 98). Identifique quem são os Terminais e os Não-Terminais.

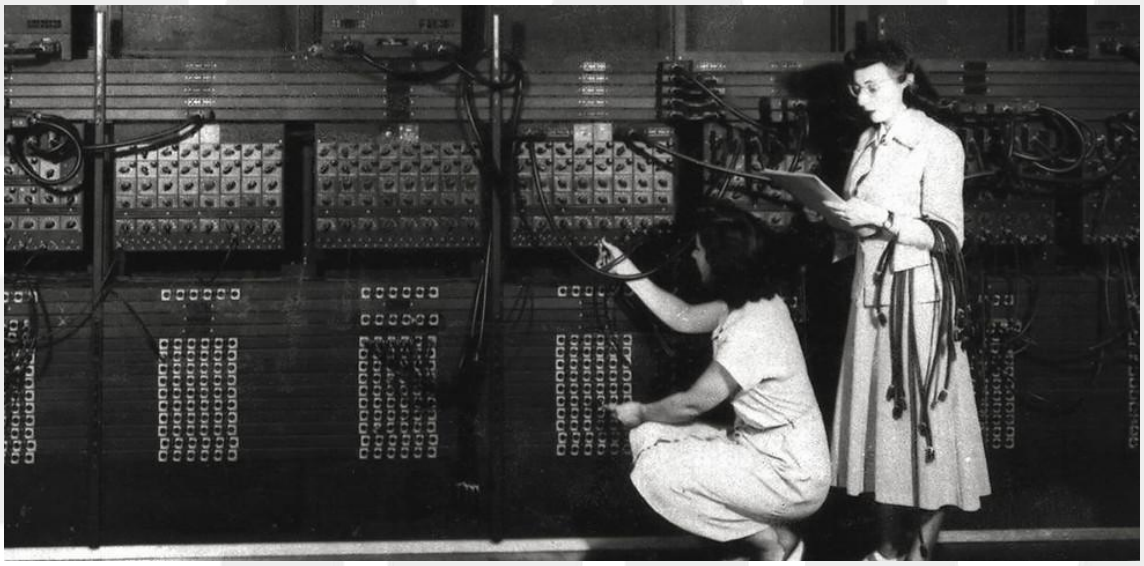
2. **Desenho Técnico:** Dada a gramática:
   - <S> → if (<C>) { <S> } | comando_simples
   - <C> → true | false

   Desenhe a árvore de derivação para a sentença: if (true) { comando_simples }.

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

2.1 Evolução Histórica: Das linguagens de máquina aos compiladores modernos

Nos primórdios da computação, programar significava manipular chaves físicas ou cartões perfurados com códigos binários (0 e 1). Segundo **Menezes (2011)**, essa abordagem era extremamente propensa a erros e dependente do hardware específico. A evolução passou pela linguagem **Assembly**, que usava mnemônicos (como MOV ou ADD), mas ainda exigia que o programador conhecesse os registradores da CPU.



ENIAC (Electronic Numerical Integrator and Computer), 1940
Fonte: https://www.yahoo.com/lifestyle/eniac-legacy-1940s-invention-shaped-170000795.html

O grande salto ocorreu na década de 1950 com a criação do **FORTRAN** por John Backus. Como destaca **Aho et al. (2008)**, pela primeira vez, uma linguagem permitia escrever fórmulas matemáticas quase como as conhecemos, deixando para o compilador o trabalho pesado de tradução. Desde então, saímos de tradutores simples para sistemas complexos que otimizam o código para inteligência artificial, dispositivos móveis e computação em nuvem.

FORTRAN 1957

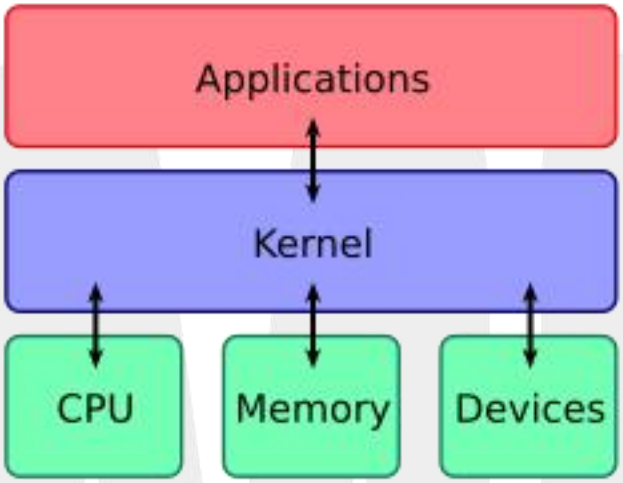
```
      C
            PIN=0.02
            IF(DDT.NE.0.0)  THEN
            DT=DDT
            ELSE
            DT=PIN
            ENDIF
            WRITE(*,'(A)') '    PLEASE ENTER NAME OF OUTPUT FILE (FOR EXAMPLE
           *  B:SIZ.DAT)'
            READ(*,'(A)') FNAMEO
            OPEN(6,FILE=FNAMEO,STATUS='UNKNOWN')
            FV=WFLX/TH
            RS=REQ*ROU*KD/TH
            C0=CS
      C
            TIME=0.000
            EF=0.000
        5   CONTINUE
            GAMMA=DT/(2.D0*DX*DX)
            BETA=DT/DX
            IF((BETA*FV).GT.0.5D0) GO TO 7
            IF((GAMMA*D/(BETA*FV)).LT.0.5D0) GO TO 6
            GO TO 8
        6   DX=DX/2
            GO TO 5
        7   DT=DT/2
            GO TO 5
        8   CONTINUE
            M=COL/DX
            NM1=M-1
            NM2=M-2
            NP1=M+1
            GAMMA=DT/(2*DX*DX)
```

COBOL 1959

```
ZBNKPRT1.cbl
ZBNKPRT1.cbl
      *A-B--+----2----+----3----+----4----+----5----+----6----+----7--I-+----8
001500*****************************************************************zbnkprt1
001600 IDENTIFICATION DIVISION.                                         zbnkprt1
001700 PROGRAM-ID.                                                      zbnkprt1
001800     ZBNKPRT1.                                                    zbnkprt1
001900 DATE-WRITTEN.                                                    zbnkprt1
002000     September 2002.                                              zbnkprt1
002100 DATE-COMPILED.                                                   zbnkprt1
002200     Today.                                                       zbnkprt1
002300                                                                  zbnkprt1
002400 ENVIRONMENT DIVISION.                                            zbnkprt1
002500 INPUT-OUTPUT   SECTION.                                          zbnkprt1
002600   FILE-CONTROL.                                                  zbnkprt1
002700     SELECT EXTRACT-FILE                                          zbnkprt1
002800            ASSIGN       TO EXTRACT                           zbnkprt1
002900            ORGANIZATION IS SEQUENTIAL                        zbnkprt1
003000            ACCESS MODE  IS SEQUENTIAL                        zbnkprt1
003100            FILE STATUS  IS WS-EXTRACT-STATUS.                zbnkprt1
```

2.2 Arquitetura do computador (CPU, Memória, Device, Kernel e Programas)

Para entender como um programa funciona, precisamos visualizar o computador como uma cebola composta por várias camadas de abstração. O compilador é a ferramenta que atravessa essas camadas.



Fonte: https://en.wikipedia.org/wiki/Kernel_%28operating_system%29

2.2.1 A Hierarquia do Sistema

Um computador moderno é organizado da seguinte forma:

1. **Hardware (CPU, Memória, Devices):** A camada física. A CPU só entende sinais elétricos que representam 0 e 1 (instruções de máquina).
2. **Kernel (o coração do SO):** É o software de mais baixo nível que tem controle total sobre o hardware. Ele gerencia quem usa a CPU e quando um dado é gravado no disco (Device).
3. **Sistema Operacional (SO):** Fornece uma interface para que os programas interajam com o Kernel sem precisar conhecer os detalhes físicos de cada placa-mãe.
4. **Programas (User Space):** São os aplicativos que nós criamos. Eles rodam em um ambiente protegido e não podem tocar no hardware diretamente.

2.2.2 O Compilador como Mediador

De acordo com **Aho et al. (2008)**, o compilador deve traduzir o código de alto nível para algo que respeite as regras do Sistema Operacional e da CPU. Quando você escreve um programa em C# ou Python para salvar um arquivo, o compilador/interpretador faz o seguinte:

- **Tradução para System Calls:** O compilador traduz comandos como `File.Write()` em uma **Chamada de Sistema (System Call)**. Essa é a única forma de um programa "pedir permissão" ao **Kernel** para acessar um **Device** (como o SSD ou HD).
- **Gerenciamento de Memória:** O compilador organiza como as variáveis serão alocadas na **Memória RAM**, garantindo que o programa não tente acessar o espaço de memória de outro processo, o que causaria o famoso erro de *Segmentation Fault*.
- **Otimização para a CPU:** O Back-End do compilador escolhe as instruções que a **CPU** executa de forma mais eficiente, aproveitando os registradores internos para minimizar o tráfego de dados entre a memória e o processador.

2.2.3 A Interação Direta: Programas e Kernel

Sem o compilador, o programador teria que escrever código em Assembly específico para cada modelo de processador e conhecer as tabelas de interrupção do Kernel. Como destaca **Diverio (2011)**, o compilador abstrai essa complexidade. Ele cria um contrato: o programador foca na lógica do negócio, e o compilador garante que o binário gerado saiba "conversar" com o Kernel para exibir dados na tela ou receber inputs do teclado.

2.3 A Estrutura em Fases: Por que dividimos o compilador em "caixas" menores?

Construir um compilador é uma tarefa de altíssima complexidade. Para torná-la gerenciável, aplicamos o princípio de "dividir para conquistar". De acordo com **Diverio (2011)**, a divisão em fases permite que:

1. **Reutilização:** O mesmo "Back-End" possa ser usado para diferentes linguagens.
2. **Manutenção:** Se a sintaxe da linguagem mudar, alteramos apenas a "caixa" correspondente.
3. **Otimização:** Cada fase pode ser aprimorada de forma independente.

2.4 O Front-End: Foco na análise

O compilador é dividido em duas grandes partes: **Front-End** (Análise) e **Back-End** (Síntese). O Front-End é a parte que "lê" o código do programador e tenta entendê-lo.

Se o programador escrever x = 10 +, o Front-End é o responsável por gritar: "Ei, falta alguma coisa depois do mais!". Ele não se preocupa com o processador que vai rodar o código; ele se preocupa em garantir que o que foi escrito faz sentido segundo as regras

da linguagem. Ele termina sua tarefa gerando uma **Representação Intermediária (IR)**, que é um código neutro pronto para ser otimizado.

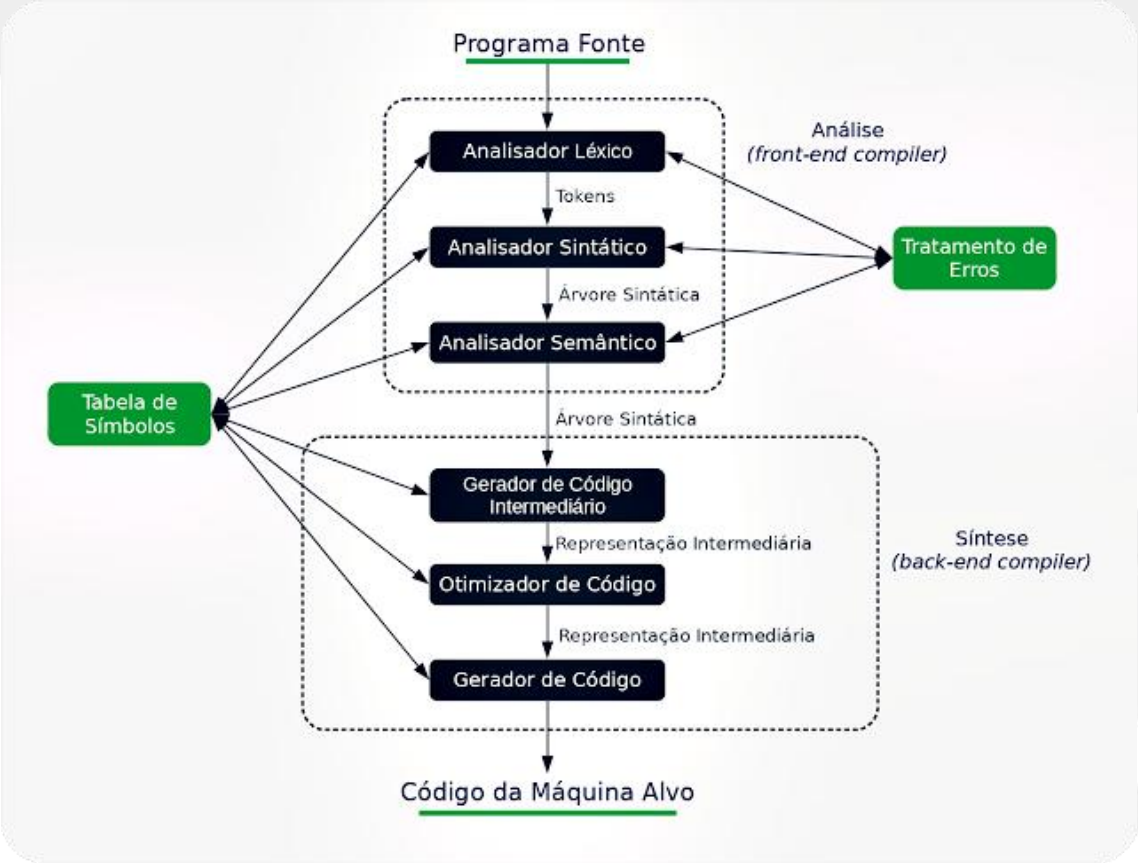
2.5 As Três Análises Principais: Introdução rápida

Para entender o código, o Front-End realiza três tarefas sequenciais:

1. **Análise Léxica (Scanner):** Lê o texto, caractere por caractere, e agrupa-os em "tokens" (palavras-chave, números, símbolos). É como identificar o que é substantivo e o que é verbo em uma frase.

2. **Análise Sintática (Parser):** Pega os tokens e verifica se a ordem deles forma uma estrutura válida (a árvore que vimos na Unidade 1). Verifica a "gramática".

3. **Análise Semântica (Checker):** Verifica o significado. Por exemplo: "Você está tentando somar um texto com um número?" ou "Esta variável foi declarada antes de ser usada?".



Fonte: https://www.cadcobol.com.br/compiladores_wikipedia_caracteristicas.htm

I. Questões de Múltipla Escolha

1. A arquitetura de um compilador é tradicionalmente dividida em Front-End e Back-End. Qual é o principal objetivo desta separação?

A) Facilitar a interface gráfica para o utilizador final.

B) Permitir a independência entre a linguagem de origem (análise) e a arquitetura de destino (síntese).

C) Garantir que o compilador seja escrito na mesma linguagem que o código-fonte.

D) Reduzir o número de passes necessários para gerar o código binário.

2. Durante o processo de compilação, a Análise Léxica é responsável por transformar o fluxo de caracteres em:

A) Uma Árvore Sintática Abstrata (AST).

B) Uma sequência de unidades lógicas chamadas Tokens.

C) Código de máquina diretamente executável.

D) Uma tabela de endereços de memória RAM.

3. De acordo com Aho et al. (2008), o Front-End termina a sua execução ao gerar uma:

A) Listagem de erros semânticos apenas.

B) Documentação técnica do sistema.

C) Representação Intermediária (IR).

D) Interface de programação de aplicações (API).

4. Se um Analisador Sintático recebe a sequência de tokens IF, (, ID,) mas não encontra um bloco de comandos a seguir, ele reportará um erro de:

A) Semântica, pois o significado está incompleto.

B) Léxico, pois o token IF foi mal escrito.

C) Sintaxe, pois a estrutura da frase não respeita a gramática da linguagem.

D) Tempo de execução (Runtime), pois o programa irá crashar.

5. A Análise Semântica diferencia-se da Sintática por ser capaz de verificar:

A) Se o programador utilizou nomes de variáveis muito longos.

B) Se as chaves { } estão devidamente balanceadas.

C) A compatibilidade de tipos em operações, como somar um número a uma string.

D) Se todos os caracteres do ficheiro pertencem ao alfabeto da linguagem.

6. No modelo de "Compilador em Fases", qual é a função da Tabela de Símbolos?

A) Armazenar os ícones utilizados na interface do compilador.

B) Atuar como uma estrutura de dados central que armazena informações sobre identificadores (variáveis, funções) para todas as fases.

C) Converter o código de alto nível em linguagem Assembly.

D) Listar apenas as palavras reservadas da linguagem.

7. Segundo Menezes (2011), as linguagens de máquina foram substituídas por compiladores modernos principalmente para:

A) Aumentar a velocidade física do processador.

B) Proporcionar um nível de abstração que aumenta a produtividade e a portabilidade do código.

C) Permitir que o computador funcione sem sistema operativo.

D) Tornar o hardware mais barato.

8. O que acontece na fase de Pré-processamento, que ocorre antes da Análise Léxica?

A) A otimização final do código binário.

B) A inclusão de ficheiros de cabeçalho (headers) e a expansão de macros.

C) A execução dos testes unitários do software.

D) A verificação da licença de utilização do compilador.

9. Um erro léxico ocorre quando:

A) O programador divide um número por zero.

B) O analisador encontra um caractere ou sequência que não forma nenhum token válido na linguagem (ex: um símbolo @ numa linguagem que não o permite).

C) Uma função é chamada com o número errado de argumentos.

D) O disco rígido fica sem espaço durante a compilação.

10. A modularização do compilador em "caixas" (Léxico, Sintático, Semântico) é um exemplo de qual princípio de Engenharia de Software?

A) Encapsulamento de dados apenas.

B) Divisão para conquistar (Separação de preocupações).

C) Programação orientada a objetos pura.

D) Desenvolvimento ágil.

II. Perguntas Reflexivas

1. **Abstração e Engenharia:** Por que razão a divisão em fases (Léxico, Sintático e Semântico) é considerada mais eficiente do que tentar criar um único programa que converta o texto diretamente em binário num só passo? Reflita sobre a manutenção e a detecção de erros.

2. **O Papel do Significado:** Imagine uma frase em português que é gramaticalmente correta, mas não faz sentido (ex: "As ideias verdes incolores dormem furiosamente"). Como esta frase exemplifica a diferença entre a Análise Sintática e a Análise Semântica num compilador?

III. Exercícios Práticos

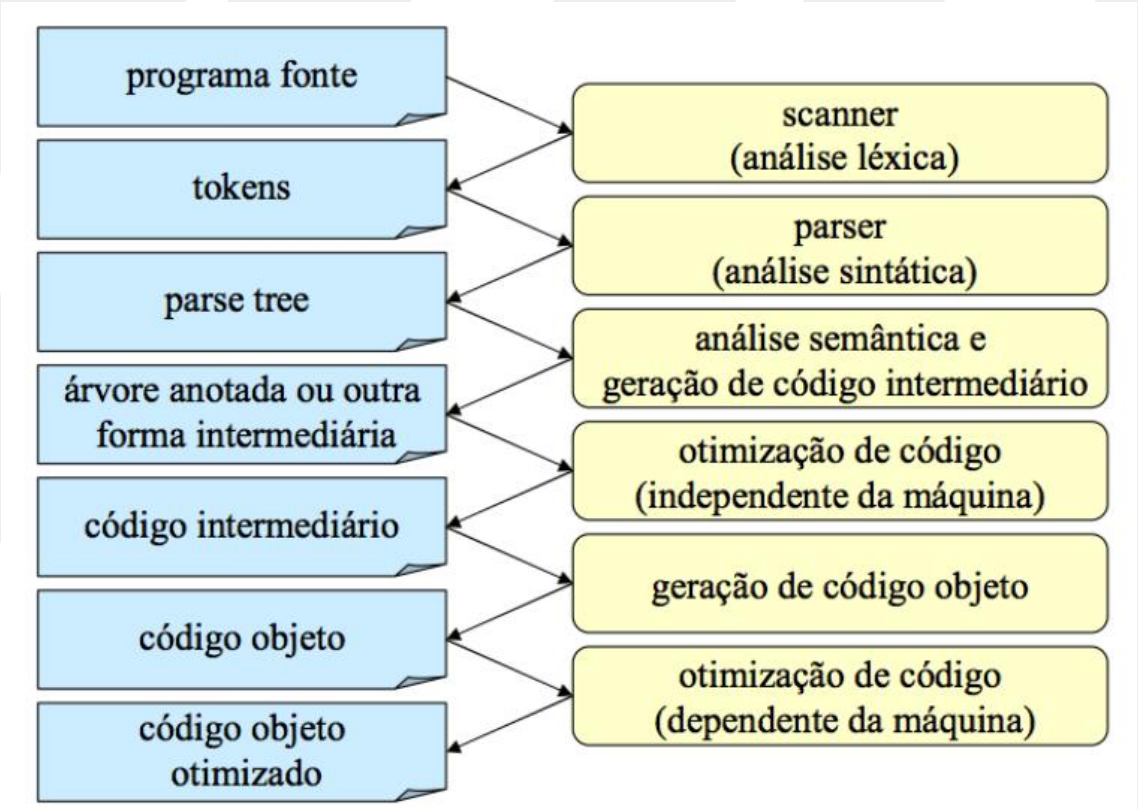
1. **Fluxo de Tokens:** Dada a linha de código Python: resultado = (valor1 + 50) * 2.
    - Liste todos os tokens que o **Analisador Léxico** deve identificar.
    - Identifique quais destes tokens são *Identificadores*, quais são *Operadores* e quais são *Literais Numéricos*.

2. **Simulação de Falhas:** Considere o comando: print(x + 10).
    - Descreva um cenário de erro que seria apanhado na **Análise Léxica**.
    - Descreva um cenário de erro que passaria pelo Léxico e Sintático, mas seria bloqueado na **Análise Semântica**.

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

3.1 O Back-End: Foco na síntese

Enquanto o Front-End é analítico, o **Back-End** é sintético. Sua missão é pegar a Representação Intermediária (IR) — que já foi validada e está livre de erros sintáticos — e traduzi-la para a "língua" do computador de destino.



De acordo com **Aho et al. (2008)**, o Back-End lida com detalhes de baixo nível que o programador geralmente ignora:

1. **Seleção de Instruções:** Escolher quais comandos do processador (ex: Intel x86 ou ARM) são os melhores para realizar uma soma ou um loop.
2. **Alocação de Registradores:** Decidir quais dados ficarão nos espaços mais rápidos da CPU (registradores) para ganhar velocidade.
3. **Otimização de Código:** Reescrever partes do código para que ele rode mais rápido ou consuma menos bateria, sem alterar o resultado.

3.2 Compilador vs. Interpretador

Uma dúvida comum na disciplina é a diferença entre compilar e interpretar. Como destaca **Menezes (2011)**, a diferença principal reside quando a tradução ocorre.

- **Compilador:** É como **traduzir um livro inteiro**. Você pega o código-fonte, passa por todas as fases e gera um arquivo executável (.exe, .app). Depois disso, você não precisa mais do código original ou do compilador para rodar o programa.

- **Interpretador:** É como um **tradutor simultâneo**. Ele lê uma linha de código, traduz e já pede para o computador executar na hora. Linguagens como Python e PHP utilizam este modelo (embora usem uma forma híbrida com *Bytecodes*).

Característica	Compilador	Interpretador
Execução	Gera um arquivo pronto para rodar.	Executa o código linha por linha.
Velocidade	Mais rápido após compilado.	Mais lento (tradução em tempo real).
Erros	Mostra todos os erros antes de rodar.	Para a execução assim que encontra um erro.

3.3 Estudo de Caso: O caminho de um "Print"

Para consolidar o conhecimento, vamos traçar a jornada do comando **`print("Olá")`** desde que você aperta o Enter até o texto aparecer no monitor:

1. **Léxico:** O compilador/interpretador quebra o texto em: print (função), ((delimitador), "Olá" (string) e) (delimitador).
2. **Sintático:** Verifica se a função print está escrita corretamente e se os parênteses estão fechados.
3. **Semântico:** Checa se a função print existe na biblioteca padrão e se o que está dentro dela pode ser exibido.
4. **Back-End/Interpretador:** O código é convertido em uma chamada de sistema (System Call). O compilador diz ao Sistema Operacional: "Ei, envie estes bytes para a saída padrão (vídeo)".
5. **Hardware:** O processador executa a instrução, a placa de vídeo desenha os pixels e você lê "Olá" na tela.

Exemplo de etapas do compilador:

x = 10 + 5

```
--- FASE 1: ANÁLISE LÉXICA ---
Tokens identificados: ['x', '=', '10', '+', '5']

--- FASE 2: ANÁLISE SINTÁTICA ---
Estrutura gramatical válida!
Árvore Sintática (AST): {'atribuicao': {'alvo': 'x', 'expressao':
['10', '+', '5']}}

--- FASE 3: ANÁLISE SEMÂNTICA ---
Tipos validados: Números Inteiros detectados.

--- FASE 4 E 5: OTIMIZAÇÃO E SÍNTESE ---
Otimização: Substituindo '10 + 5' por '15'
Código Final Gerado (Python): x = 15
```

Simulação do Compilador em Python

```python
# SIMULADOR DAS ETAPAS DE PROCESSAMENTO DE UM COMPILADOR
def simulador_compilador(linha_codigo):
    print(f"--- FASE 1: ANÁLISE LÉXICA ---")
    # Quebrando a string em 'palavras' (tokens)
    tokens = linha_codigo.replace('=', ' = ').replace('+', ' + ').split()
    print(f"Tokens identificados: {tokens}")

    print(f"\n--- FASE 2: ANÁLISE SINTÁTICA ---")
    # Verificando se a estrutura é: VARIÁVEL = VALOR + VALOR
    if len(tokens) == 5 and tokens[1] == "=":
        print("Estrutura gramatical válida!")
        arvore = {"atribuicao": {"alvo": tokens[0], "expressao": [tokens[2], tokens[3], tokens[4]]}}
        print(f"Árvore Sintática (AST): {arvore}")

    print(f"\n--- FASE 3: ANÁLISE SEMÂNTICA ---")
    # Verificando se os valores são números
    try:
        val1 = int(tokens[2])
        val2 = int(tokens[4])
        print("Tipos validados: Números Inteiros detectados.")
    except ValueError:
        print("Erro Semântico: Você tentou somar algo que não é número!")
        return

    print(f"\n--- FASE 4 E 5: OTIMIZAÇÃO E SÍNTESE ---")
    resultado_otimizado = val1 + val2
    print(f"Otimização: Substituindo '10 + 5' por '{resultado_otimizado}'")
    print(f"Código Final Gerado (Python): {tokens[0]} = {resultado_otimizado}")

# Execução do teste
simulador_compilador("x = 10 + 5")
```

simulador-compilador.py

I. Questões de Múltipla Escolha

1. Qual é a principal responsabilidade da fase de Back-End de um compilador?

A) Verificar se o programador esqueceu algum ponto e vírgula.

B) Traduzir a Representação Intermediária para código de máquina específico de uma arquitetura.

C) Desenhar a interface gráfica do programa.

D) Criar o ícone do arquivo executável.

2. Sobre os Interpretadores, é correto afirmar que:

A) Eles geram um arquivo .EXE independente.

B) Eles traduzem e executam o código em tempo real, linha por linha.

C) São sempre mais rápidos que os compiladores.

D) Não precisam realizar análise léxica.

3. A técnica de "Alocação de Registradores" no Back-End serve para:

A) Organizar os nomes das variáveis em ordem alfabética.

B) Decidir quais dados devem ficar nos espaços de memória mais rápidos da CPU para otimizar a performance.

C) Salvar o código-fonte na nuvem.

D) Instalar o compilador no Windows.

4. Por que o processo de compilação costuma levar mais tempo do que a interpretação inicial, mas resulta em programas mais rápidos?

A) Porque o compilador gasta tempo otimizando e traduzindo todo o código de uma vez antes da execução.

B) Porque o compilador precisa de internet para funcionar.

C) Porque os interpretadores ignoram os erros de lógica.

D) Porque compilar exige mais memória RAM do que rodar.

5. Qual fase é responsável por converter x = 2 + 3 em instruções de baixo nível como MOV EAX, 2 e ADD EAX, 3?

A) Análise Léxica.

B) Análise Sintática.

C) Geração de Código (Back-End).

D) Pré-processamento.

6. No estudo de caso do comando "Print", a etapa que verifica se o texto está entre aspas é a:

A) Geração de Código.

B) Análise Sintática.

C) Otimização de Registradores.

D) Alocação de Memória.

7. Linguagens modernas como Java e C# utilizam um modelo híbrido. Elas compilam para um "Bytecode" que é interpretado por uma máquina virtual. Qual a vantagem disso?

A) O código fica mais difícil de ler.

B) Portabilidade: o mesmo Bytecode roda em qualquer sistema que tenha a máquina virtual instalada.

C) Não é necessário usar variáveis.

D) O programa ocupa menos espaço no HD.

8. O que é a Representação Intermediária (IR)?

A) É o código-fonte original escrito pelo usuário.

B) É uma linguagem "meio do caminho", independente de máquina, que conecta o Front-End ao Back-End.

C) É a tela de erro que aparece para o programador.

D) É o manual de instruções do processador Intel.

9. Um compilador que gera código para uma arquitetura diferente daquela em que está rodando é chamado de:

A) Compilador Reverso.

B) Compilador Cruzado (Cross-Compiler).

C) Interpretador Lógico.

D) Tradutor Simultâneo.

10. A fase de "Otimização de Código" no Back-End busca:

A) Mudar o nome das variáveis para nomes mais curtos.

B) Melhorar a eficiência do programa (tempo e espaço) sem alterar seu comportamento original.

C) Adicionar novos recursos que o programador esqueceu.

D) Corrigir erros de digitação automaticamente.

II. Perguntas Reflexivas

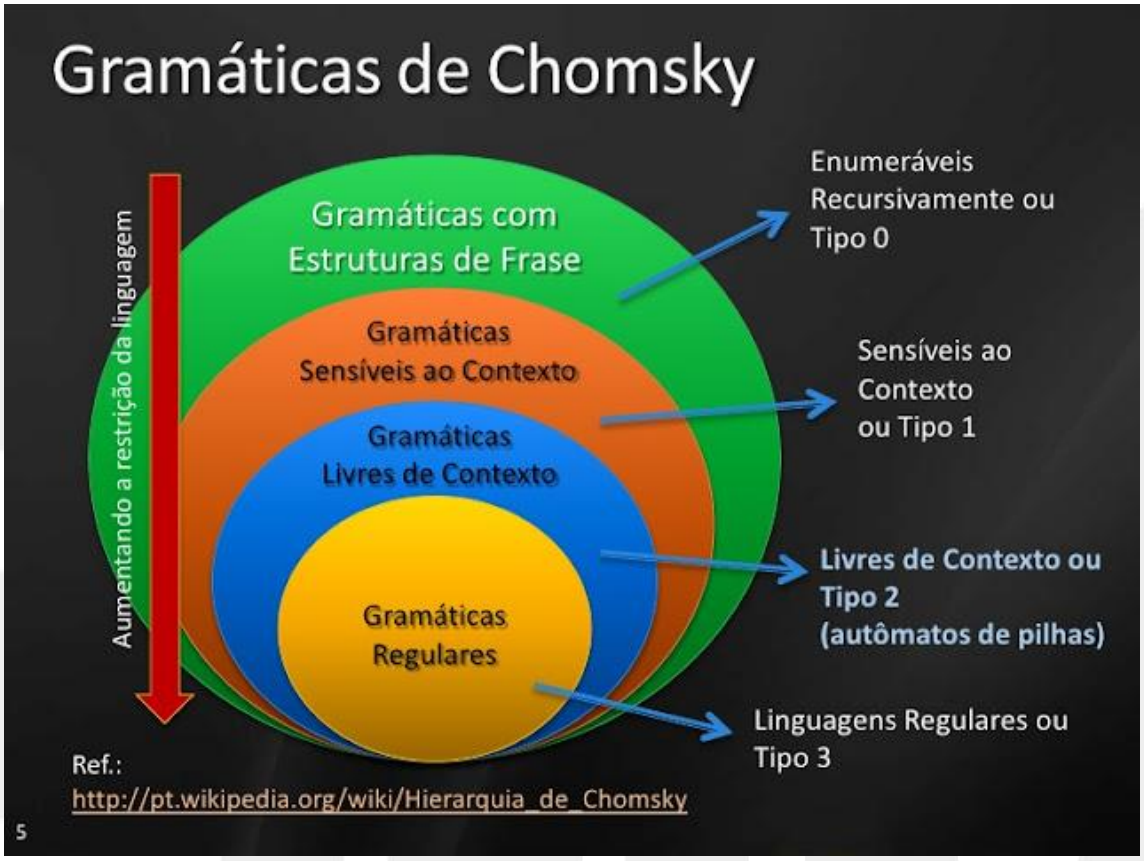
1. **Escolha de Ferramenta:** Se você estivesse desenvolvendo um sistema de controle de freios ABS para um carro (onde o tempo de resposta é crítico), você usaria uma linguagem compilada ou interpretada? Justifique com base na Unidade 3.

2. **O Futuro da Tradução:** Com o aumento do poder de processamento, a diferença de velocidade entre interpretadores e compiladores está diminuindo. Você acredita que, no futuro, os compiladores deixarão de existir?

III. Exercícios Práticos

1. **Mapeamento de Execução:** Imagine o comando y = 10 / 2.
    - Como o **Back-End** traduziria isso para um processador que não possui a instrução de "DIVISÃO" (comum em microcontroladores simples)? *Dica: Pense em subtrações sucessivas.*

2. **Pesquisa Acadêmica:** Procure a definição de **JIT (Just-In-Time Compiler)**. Explique em um parágrafo como ele tenta unir o "melhor dos dois mundos" entre compiladores e interpretadores.



Fonte: https://radiomaffei.blogspot.com/2012/12/esse-cara-vale-pena-ser-lido.html

4.1 Automação na Criação: Por que não escrevemos tudo do zero?

Escrever um analisador léxico e sintático do zero é um processo repetitivo e extremamente suscetível a erros de lógica. Como afirma Aho et al. (2008), a automação permite que o desenvolvedor se concentre na **especificação** da linguagem (o que ela deve fazer) em vez da **implementação** dos algoritmos de busca (como ela deve ler).

Ao utilizarmos geradores de compiladores, fornecemos as Regras de Produção (vistas na Unidade 1) e a ferramenta gera automaticamente o código em C, Java ou Python que executa a análise. Isso garante que o motor do compilador seja matematicamente provado e livre de *bugs* comuns de indexação ou estouro de memória.

4.2 Ferramentas de Mercado: A História do Lex/Yacc e AntLR

Lex e Yacc (A "Velha Guarda" em C)

Surgidos nos laboratórios Bell na década de 1970, o **Lex** (Léxico) e o **Yacc** (*Yet Another Compiler-Compiler*) são as ferramentas fundamentais que moldaram o Unix.

- **Lex:** Gera analisadores léxicos baseados em Expressões Regulares.
- **Yacc:** Gera analisadores sintáticos baseados em gramáticas LALR. Eles permitiram que linguagens como C e SQL fossem implementadas com rigor acadêmico. Embora antigos, sua lógica ainda é a base para ferramentas modernas como o **Flex** e o **Bison**.

AntLR (A Evolução em Java)

O **AntLR** (*ANother Tool for Language Recognition*), criado por Terence Parr, revolucionou o mercado por ser mais amigável e poderoso. Ao contrário do Yacc, o AntLR utiliza análise **LL(*),** o que permite mensagens de erro muito mais claras e suporte a diversas linguagens de saída, como Java, C#, Python e JavaScript. É amplamente utilizado no desenvolvimento de linguagens modernas e ferramentas de análise de dados.

O significado da sigla LL(*)

- **O primeiro L (Left-to-right):** Indica que o analisador lê o código da **esquerda para a direita**, exatamente como nós lemos um livro.
- **O segundo L (Leftmost derivation):** Significa que, ao construir a árvore de derivação (que vimos na Unidade 1), o compilador expande sempre o símbolo não-terminal mais à esquerda primeiro.
- **O asterisco (*):** Esta é a grande inovação do AntLR. Em analisadores mais simples, como o LL(1), o compilador só consegue olhar **um** token à frente para decidir o que fazer. Se ele encontrar um conflito, ele "trava". Já o **LL(*)** possui um *lookahead* dinâmico (infinito), o que significa que ele pode olhar quantos tokens forem necessários para frente até ter certeza de qual regra gramatical deve aplicar.

Por que isso é uma "revolução"?

Imagine que você está em uma estrada que se bifurca.

- Um analisador **LL(1)** só consegue ver 1 metro à frente. Se as duas estradas parecerem iguais nos primeiros 10 metros, ele se perde.
- O analisador **LL(*)** consegue "enviar um drone" para sobrevoar as duas estradas até o fim da curva para descobrir qual é a correta.

Vantagens Práticas

1. **Mensagens de Erro Claras:** Como o AntLR consegue olhar muito à frente, quando algo está errado, ele sabe exatamente o que o programador *tentou* fazer, permitindo dar avisos como: *"Você esqueceu um parêntese na linha 10"* em vez de apenas *"Erro de sintaxe"*.
2. **Gramáticas mais Naturais:** Você não precisa "torturar" sua gramática para que ela seja aceita pelo compilador; o LL(*) lida com ambiguidades que fariam ferramentas mais antigas (como o Yacc) falharem.
3. **Poder de Predição:** Ele utiliza Autômatos Finitos Determinísticos (DFA) dinâmicos para tomar essas decisões em tempo de execução, garantindo alta performance mesmo com esse olhar "infinito" para frente.

Exemplo – LL(1) e LL(*) funcionam bem

1. **Atribuição:** x = 5;
2. **Chamada de Função:** x(5);

Para um analisador **LL(1)**, quando ele lê o token x, ele olha para o próximo token (*lookahead* de 1).

- Se o próximo for =, ele sabe que é uma atribuição.
- Se o próximo for (, ele sabe que é uma função.

Exemplo - Onde o LL(1) falha e o LL(*) funciona

Imagine que a nossa linguagem permita nomes de variáveis compostos ou caminhos de pacotes, como em Java:

- fmu.projeto.compiladores.resultado = 10; (atribuição)
- fmu.projeto.compiladores.calcular(10); (chamada)

O Problema para o LL(1): Quando o analisador lê **fmu**, ele olha o próximo token '.'.

Ambas as regras (atribuição e chamada) começam com **ID .**. O LL(1) não consegue decidir qual regra seguir porque ele só vê um passo à frente. Ele precisaria ver 4, 5 ou 10 tokens à frente para encontrar o **=** ou o **(** que define a estrutura.

A Solução do LL(*): O AntLR não se limita. Ele vê o **fmu**, depois o **.**, depois o **projeto**, depois o **.**, e continua "**caminhando**" pela frase até encontrar o **símbolo** que **desfaz a dúvida**.

4.3 Vantagens Práticas: Produtividade e Segurança

A principal vantagem é o **ganho de produtividade**. Um desenvolvedor pode alterar uma regra da gramática em segundos, e a ferramenta regenera milhares de linhas de código instantaneamente. Além disso, há uma redução drástica de erros humanos, especialmente em verificações de tipos e limites de memória.

O Caso Rust: Kernel e Sistemas Embarcados

Atualmente, a linguagem **Rust** tem ganhado destaque por trazer a segurança de memória para o nível do **Kernel**. Diferente do C, o compilador do Rust possui um "verificador de empréstimos" (*Borrow Checker*) que impede erros de ponteiros e invasões de memória em tempo de compilação.

- **No Kernel Linux:** O Rust está sendo integrado para permitir a criação de drivers mais seguros, reduzindo as famosas "telas azuis" causadas por falhas de memória.
- **Embarcados:** Por não possuir uma "coleta de lixo" (*Garbage Collector*) pesada, o Rust é ideal para microcontroladores onde cada byte de RAM conta.

4.4 Exemplos Práticos

4.4. Lex & Yacc (Linguagem C)

No mundo C, trabalhamos com dois arquivos separados. O Lex cuida das palavras e o Yacc da estrutura.

Arquivo do Lex (scanner.l):

```
%{
#include "y.tab.h"
%}
%%
[0-9]+    { yylval = atoi(yytext); return NUM; }
[ \t\n]   ; /* ignora espaços */
"+"       { return PLUS; }
.         { return yytext[0]; }
%%
```

Arquivo do Yacc (parser.y):

```
%token NUM PLUS
%%
exp: NUM PLUS NUM { printf("Resultado: %d\n", $1 + $3); }
    ;
%%
```

Essas ferramentas foram fundamentais para o desenvolvimento do sistema Unix e da linguagem C.

4.4.2 AntLR (Java/Multi-plataforma)

O AntLR usa um formato unificado (.g4) e gera código em várias linguagens (Java, C#, Python).

Arquivo de Gramática (Calculadora.g4):

```
grammar Calculadora;
// Regra Sintática
prog: ID '=' expr;
expr: INT '+' INT;

// Regras Léxicas
ID: [a-z]+;
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

4.4.3 Lark (Python)

O **Lark** é uma biblioteca (lib) moderna de **parsing** (análise sintática) para a linguagem Python. Ela serve para **analisar, interpretar e processar** textos estruturados ou linguagens de programação, transformando-os em uma **árvore de sintaxe abstrata** (AST - Abstract Syntax Tree) que o Python consegue manipular facilmente.

Para que serve o Lark?

- **Criar Interpretadores/Compiladores:** Você pode definir regras gramaticais para sua própria linguagem e o Lark a analisará.
- **Análise de Dados Complexos:** Útil para ler formatos de arquivo personalizados, configurar arquivos complexos ou logs que não seguem formatos padrão como JSON ou XML.
- **Criar DSLs (Linguagens de Domínio Específico):** Desenvolver linguagens simples para resolver problemas específicos.

- **Parsing Ambiguo:** O Lark é excelente em lidar com gramáticas ambíguas ou complexas que outras ferramentas (como PLY) têm dificuldade em processar.

Principais Características e Vantagens

1. **Fácil de usar (Ergonômico):** Projetado para ser amigável, permitindo criar parsers com muito pouco código.
2. **Geração Automática de Árvore:** Constrói automaticamente uma árvore de sintaxe (AST) baseada na estrutura da gramática definida.
3. **Algoritmos Poderosos:** Inclui algoritmos de parsing como Earley (para qualquer gramática livre de contexto), LALR(1) (rápido, para gramáticas menos complexas) e CYK.
4. **Baseado em EBNF:** Utiliza uma sintaxe baseada em EBNF (Extended Backus-Naur Form) para definir gramáticas de forma intuitiva.
5. **Reconstrução de Texto:** O Lark pode gerar texto novamente a partir de uma árvore de sintaxe, útil para transformações de código.
6. **Puro Python:** Funciona em qualquer interpretador Python.

```
# 1. Importamos a ferramenta Lark
# Imagine que o Lark é um mestre de obras que sabe ler uma planta (gramática)
# e construir uma máquina que entende textos.
from lark import Lark

# 2. Definimos a "Gramática
# Usamos uma string (texto) de várias linhas com aspas triplas.
gramatica = """
    # 'start' é o ponto de entrada. O '?' diz para simplificar a árvore se possível.
    ?start: soma

    # Aqui dizemos: uma 'soma' é formada por um NÚMERO, o símbolo "+" e outro NÚMERO.
    soma: NUMBER "+" NUMBER

    # Importamos definições prontas do Lark:
    # NUMBER: sabe reconhecer números como 10, 20, 3.14.
    # WS: significa 'White Space' (espaços, tabs, quebras de linha).
    %import common.NUMBER
    %import common.WS

    # Dizemos ao computador para IGNORAR espaços.
    # Assim, "10+20" ou "10   +   20" serão lidos da mesma forma.
    %ignore WS
"""

# 3. Criamos o nosso 'parser' (analisador).
# Passamos a nossa planta (gramatica) para o Lark construir o motor de análise.
parser = Lark(gramatica)

# 4. Pedimos ao parser para analisar o texto "10 + 20".
# O comando .parse() transforma o texto em uma Árvore'.
arvore = parser.parse("10 + 20")
```

```
# 5. O .pretty() serve para imprimir essa árvore no console.
# Isso ajuda o aluno a visualizar a hierarquia da operação.
print(arvore.pretty())
```

4.4.4 Diferenças entre as Ferramentas

Característica	Lex/Yacc (Flex/Bison)	AntLR	Lark (Python)
Linguagem Base	C / C++	Java (gera para várias)	Python
Algoritmo	LALR (Bottom-Up)	LL(*) (Top-Down)	Earley, LALR, Cyk
Curva de Aprendizado	Alta (Complexo)	Média	Baixa (Muito simples)
Uso em Sistemas	Kernel, Drivers, GCC	Grandes IDEs, Hibernate	Automação, Ciência de Dados
Configuração	Exige compilação externa	Exige instalação do Java	pip install lark

Resumo

- **Lex/Yacc:** É o "trabalho pesado". Excelente para quando a performance máxima é necessária, como no desenvolvimento de um **Kernel** ou sistemas embarcados.

- **AntLR:** É a ferramenta industrial mais versátil. Se você quer criar uma linguagem que rode em .NET, Java e Python ao mesmo tempo, use AntLR.

- **Lark:** É a agilidade. Para prototipagem rápida, análise de logs ou DSLs (Linguagens de Domínio Específico) em Python, é imbatível.

I. Questões de Múltipla Escolha

1. De acordo com Aho et al. (2008), qual é a principal justificativa para a utilização de ferramentas de automação na criação de analisadores léxicos e sintáticos?

A) Reduzir o custo de hardware necessário para a compilação.

B) Permitir que o desenvolvedor foque na especificação da linguagem em vez da implementação repetitiva de algoritmos de busca.

C) Eliminar a necessidade de conhecer Gramáticas Livres de Contexto.

D) Tornar o código-fonte original mais curto para o usuário final.

2. As ferramentas Lex e Yacc, surgidas na década de 1970, são consideradas a "velha guarda" por terem moldado o sistema Unix. Qual a função específica do Yacc nesse conjunto?

A) Gerar analisadores léxicos baseados em Expressões Regulares.

B) Traduzir código C diretamente para Python.

C) Gerar analisadores sintáticos baseados em gramáticas LALR.

D) Realizar a otimização de registros na CPU.

3. O AntLR (ANother Tool for Language Recognition) trouxe inovações importantes em relação ao Yacc. Uma de suas principais características diferenciais é:

A) O uso de análise LL(*) e o suporte a múltiplas linguagens de saída, como Java, C# e Python.

B) A exigência de que o programador escreva o código em binário.

C) Ser uma biblioteca exclusiva para a linguagem C.

D) Não permitir o tratamento de mensagens de erro.

4. Sobre a biblioteca Lark para Python, qual característica a torna ideal para prototipagem rápida e análise de dados complexos?

A) A necessidade de compilação externa antes de rodar o código.

B) O fato de ser baseada em Java e exigir uma Máquina Virtual.

C) Sua facilidade de uso (ergonomia) e a capacidade de gerar automaticamente árvores de sintaxe (AST).

D) O suporte exclusivo para gramáticas regulares simples.

5. No contexto de sistemas operacionais e segurança, por que a linguagem Rust tem sido integrada ao Kernel do Linux?

A) Porque o Rust não possui um compilador, facilitando a execução.

B) Devido ao seu "verificador de empréstimos" (Borrow Checker), que impede erros de memória e ponteiros em tempo de compilação.

C) Para substituir o uso de Python em scripts de automação do kernel.

D) Porque o Rust é a única linguagem que suporta a ferramenta Lex/Yacc.

6. Analisando a comparação entre ferramentas, qual delas é descrita como a "ferramenta industrial mais versátil" para projetos multi-plataforma?

A) Lex/Yacc.

B) Lark.

C) AntLR.

D) GCC.

7. O algoritmo Earley, disponível na biblioteca Lark, é particularmente útil para:

A) Analisar apenas gramáticas extremamente simples.

B) Lidar com qualquer gramática livre de contexto, incluindo gramáticas ambíguas.

C) Aumentar o consumo de bateria em sistemas embarcados.

D) Substituir o Kernel do Windows.

8. Uma vantagem prática crucial do uso de geradores de compiladores em relação ao desenvolvimento manual é:

A) A ferramenta garante um motor de compilador matematicamente provado e livre de bugs comuns de indexação.

B) O código gerado é sempre em linguagem humana, facilitando a leitura por leigos.

C) Eles dispensam a definição de Regras de Produção.

D) Eles permitem que o programa rode sem memória RAM.

9. Na gramática do AntLR apresentada, a regra expr: INT '+' INT; é classificada como:

A) Uma regra léxica.

B) Uma regra de pré-processamento.

C) Uma regra sintática.

D) Um comentário de código.

10. Qual ferramenta é recomendada quando a prioridade absoluta é a performance máxima em sistemas embarcados ou kernels?

A) Lark.

B) AntLR.

C) Lex/Yacc Flex/Bison).

D) Bloco de notas.

II. Perguntas Reflexivas

1. **Segurança vs. Flexibilidade:** O Rust utiliza um compilador rigoroso para garantir a segurança de memória (Borrow Checker). Como essa "rigidez" do compilador impacta a produtividade do desenvolvedor iniciante em comparação com linguagens mais flexíveis como C?

2. **A Escolha da Ferramenta:** Se você fosse desenvolver uma linguagem para cientistas de dados processarem logs de servidores em Python, por que o Lark seria uma escolha superior ao Lex/Yacc? Reflita sobre a curva de aprendizado e a integração com o ecossistema.

III. Exercícios Práticos

1. **Análise de Gramática (Lark):** Observe o código Lark abaixo:

```
gramatica = """
    ?start: subtracao
    subtracao: NUMBER "-" NUMBER
    %import common.NUMBER
    %import common.WS
    %ignore WS
"""
```

- Modifique esta gramática para que ela aceite também operações de multiplicação utilizando o símbolo *.

2. **Identificação de Tokens (AntLR):** Com base no exemplo da Unidade 4, identifique quais seriam os tokens gerados para a entrada v = 10 + 20 seguindo a gramática:

```
prog: ID '=' expr;
expr: INT '+' INT;
ID: [a-z]+;
INT: [0-9]+;
```

- Liste os tokens na ordem em que aparecem.

Unidade 5: Análise Léxica – Identificando Palavra

5.1 O que são Tokens: A menor unidade com significado

A análise léxica é a fase que lê o fluxo de caracteres do código-fonte e os agrupa em unidades significativas chamadas **Tokens**. De acordo com **Aho et al. (2008)**, um token é um par composto por um nome de token e um valor de atributo opcional.

- **Definição**: O token representa uma categoria lógica, enquanto o **Lexema** é a sequência real de caracteres no código.
- **Exemplos Comuns**:
  - **Palavras-chave**: if, else, while, int.
  - **Identificadores**: Nomes de variáveis como resultado, x1.
  - **Literais**: Números como 10.5 ou textos como "Olá".
  - **Operadores**: +, -, *, /, ==.
  - **Delimitadores**: (,), {, }, ;.

I. Palavras Reservadas (Keywords / Reserved Words)

São termos que possuem um significado fixo para o compilador e não podem ser usados como nomes de variáveis.

- **Controle de Fluxo:** if, else, switch, case, default, break, continue, return.
- **Laços de Repetição:** for, while, do, foreach.
- **Tipos de Dados:** int, float, double, char, boolean, void, string, long, short.
- **Modificadores e Escopo:** public, private, protected, static, final, const, abstract, volatile.
- **Estruturas e Objetos:** class, interface, struct, enum, extends, implements, new, this, super.
- **Tratamento de Exceções:** try, catch, finally, throw, throws.

II. 2. Identificadores (Identifiers)

Nomes escolhidos pelo programador para representar entidades.

- **Nomes de Variáveis e Constantes:** idade, valor_total, PI.
- **Nomes de Funções/Métodos:** calcularMedia(), imprimir().
- **Nomes de Classes e Namespaces:** Usuario, SistemaFinanceiro.

III. 3. Literais (Literals / Constants)

Valores fixos escritos diretamente no código.

- **Literais Inteiros:** 42, -10, 0.
- **Literais de Ponto Flutuante:** 3.14, 1.0e-10.
- **Literais de Caractere:** 'A', '\n'.
- **Literais de String:** "Olá Mundo", "FMU 2024".
- **Literais Booleanos:** true, false.
- **Literal de Nulo:** null, None, nullptr.

IV. 4. Operadores (Operators)

Símbolos que representam operações matemáticas ou lógicas.

- **Aritméticos:** +, -, *, /, %, ++, --.
- **Relacionais (Comparação):** ==, !=, >, <, >=, <=.
- **Lógicos:** &&, ||, !, and, or, not.
- **Atribuição:** =, +=, -=, *=, /=.
- **Bitwise (Bits):** &, |, ^, ~, <<, >>.

V. 5. Delimitadores e Pontuação (Punctuators / Separators)

Símbolos que organizam a estrutura e a hierarquia do código.

- **Agrupamento:** ( ) (parênteses), [ ] (colchetes), { } (chaves).
- **Separação:** , (vírgula), ; (ponto e vírgula), . (ponto), : (dois pontos).
- **Acesso e Referência:** ->, ::.

Diferença Fundamental: Nome do Token vs. Valor do Atributo

Para o compilador funcionar, ele não guarda apenas "Número". Ele guarda o **Tipo** e o **Valor**. Veja o exemplo da expressão idade = 20;:

Lexema (O que está escrito)	Nome do Token (Categoria)	Valor do Atributo (Informação Extra)
idade	IDENTIFICADOR	Ponteiro para a Tabela de

		Símbolos
=	ATRIBUICAO	(Nenhum)
20	LITERAL_INTEIRO	Valor numérico 20
;	PONTO_E_VIRGULA	(Nenhum)

5.2 Especificação com Expressões Regulares

Para que o compilador saiba o que é um "Número" ou um "Identificador", precisamos descrever esses padrões. A ferramenta matemática para isso são as **Expressões Regulares** (RegEx).

Segundo **Menezes (2011)**, as expressões regulares definem as **Linguagens Regulares**, que são o nível mais simples da hierarquia de Chomsky.

- **Padrão para Dígitos**: [0-9]+ (significa um ou mais números de 0 a 9).
- **Padrão para Letras**: [a-zA-Z]+ (uma ou mais letras).
- **Padrão para Identificadores**: [a-zA-Z_][a-zA-Z0-9_]* (deve começar com letra ou sublinhado).

5.3 Reconhecimento na Prática: Espaços e Comentários

Na prática, o código-fonte está cheio de "ruído" que não interessa para a lógica da computação, como espaços em branco, tabulações, quebras de linha e comentários.

Como destaca **Aho et al. (2008)**, uma das funções vitais do analisador léxico é filtrar esses caracteres, agindo como um "limpador" para que o Analisador Sintático receba apenas tokens puros.

- **White Spaces (WS)**: São ignorados após servirem para separar tokens (ex: int x precisa do espaço para não virar intx).
- **Comentários**: São descartados imediatamente, pois servem apenas para o programador humano.

Exemplo Prático em Lark (Análise Léxica)

No Lark, definimos os tokens usando letras MAIÚSCULAS. Veja como especificamos a "limpeza" de espaços e o reconhecimento de padrões:

```python
from lark import Lark

# Definição da Gramática com foco no Léxico
# Usamos terminais em MAIÚSCULO para definir os Tokens
gramatica_lexica = """
    start: (TOKEN | OPERADOR | NUMERO)+

    # Definição de Tokens via Expressões Regulares
    TOKEN: /[a-zA-Z_][a-zA-Z0-9_]*/
    NUMERO: /[0-9]+/
    OPERADOR: "+" | "-" | "*" | "/" | "="

    # Tratamento de Ruídos
    %import common.WS
    %ignore WS  // Aqui o Lark descarta espaços automaticamente
"""

# Criando o Scanner
parser = Lark(gramatica_lexica)

# Testando a identificação de palavras
texto = "total = 50 + valor"
tokens = parser.lex(texto)

print(f"Texto original: {texto}")
print("Tokens identificados:")
for t in tokens:
    print(f"Tipo: {t.type:10} | Valor: {t.value}")
```

analise_lexica.py

Saída:

```
Texto original: total = 50 + valor
Tokens identificados:
Tipo: TOKEN      | Valor: total
Tipo: OPERADOR   | Valor: =
Tipo: NUMERO     | Valor: 50
Tipo: OPERADOR   | Valor: +
Tipo: TOKEN      | Valor: valor
```

I. Questões de Múltipla Escolha

1. Qual é a principal diferença entre um Lexema e um Token, segundo a bibliografia básica?

A) Lexema é a categoria gramatical; Token é o texto escrito.

B) Token é a categoria lógica (ex: NUMERO); Lexema é o texto real (ex: "123").

C) Não há diferença; ambos representam a mesma coisa no compilador.

D) Lexemas são usados apenas em interpretadores; Tokens apenas em compiladores.

2. As Expressões Regulares são a base para qual fase do compilador?

A) Análise Sintática.

B) Geração de Código.

C) Análise Léxica.

D) Otimização de Performance.

3. O que acontece com os comentários de um programa durante a análise léxica?

A) São traduzidos para linguagem de máquina.

B) São armazenados na Tabela de Símbolos.

C) São ignorados/descartados pelo analisador léxico.

D) São enviados para o analisador sintático para validação.

4. Na expressão regular [a-z]+, o símbolo + significa:

A) Uma operação de soma.

B) Que o padrão deve ocorrer uma ou mais vezes.

C) Que o padrão é opcional.

D) Que apenas letras maiúsculas são aceitas.

5. Qual destes itens NÃO é considerado um token em uma linguagem de programação comum?

A) `while` (palavra-chave).

B) `(` (delimitador).

C) ` ` (espaços em branco após a análise).

D) `minhaVariavel` (identificador).

6. Qual a função de um "delimitador" como token?

A) Realizar cálculos.

B) Armazenar valores.

C) Organizar a estrutura e hierarquia (ex: separar argumentos ou blocos).

D) Substituir o uso de variáveis.

7. O identificador _soma2 é válido na maioria das linguagens porque:

A) Começa com um número.

B) Segue o padrão de começar com letra ou sublinhado.

C) Não contém letras maiúsculas.

D) É uma palavra-chave reservada.

8. O que compõe o par que define um Token?

A) Nome do arquivo e linha

B) Endereço de memória e valor.

C) Cor do texto e tamanho.

D) Nome do token e valor do atributo.

9. Se o analisador léxico encontrar o caractere $ em uma linguagem que não o suporta, o que ocorre?

A) Um erro léxico.

B) O programa roda normalmente.

C) O caractere é convertido em zero.

D) O compilador ignora e pula para a próxima linha.

10. Pergunta: A Tabela de Símbolos começa a ser preenchida principalmente em qual fase?

A) Geração de Código.

B) Análise Léxica (ao encontrar identificadores).

C) Otimização.

D) Documentação.

II. Perguntas Reflexivas

1. **Contexto:** Por que você acha que a maioria das linguagens de programação (como Java e C#) exige que identificadores não comecem com números (ex: 1variavel é inválido)? Como isso facilita a vida do analisador léxico?

2. **Evolução**: Em linguagens antigas (como o Fortran original), os espaços em branco eram ignorados em qualquer lugar. Como isso dificultava a leitura e o desenvolvimento do compilador em comparação com as linguagens modernas?

III. Exercícios Práticos

1. **Expressão Regular:** Escreva uma expressão regular simples para reconhecer um endereço de e-mail simplificado (ex: usuario@dominio.com).

2. **Lark**: Modifique o código Python acima para que ele também ignore comentários que comecem com #. Dica: Use %ignore com uma regra para o símbolo #.

Unidade 6: Máquinas de Estados – Autômatos Finitos

6.1 Autômatos Finitos: O "Cérebro" do Scanner

Um Autômato Finito (AF) é um modelo matemático de computação que consiste em um conjunto de estados e transições. Na construção de compiladores, o AF é o mecanismo que decide se uma sequência de caracteres (lexema) pertence a uma categoria de token.

De acordo com **Menezes (2011)**, um autômato pode ser visualizado como um grafo onde os nós são estados e as arestas são as transições disparadas pela leitura de um caractere. Existem dois tipos principais:

Autômato Finito Não-Determinístico (AFN)

Pode ter várias transições para o mesmo caractere a partir de um estado ou transições vazias (**ϵ**), podem existir:

- **Várias transições** para o mesmo símbolo a partir de um único estado.
- **Nenhuma transição** para um determinado símbolo.
- **Transições sem ler nenhum símbolo** (transições vazias ou ϵ).

Autômato Finito Determinístico (AFD)

Para cada estado e cada caractere de entrada, existe exatamente uma transição. É o modelo preferido para implementação devido à sua previsibilidade e velocidade. A palavra **"Determinístico"** é a chave: significa que, para qualquer estado atual e qualquer símbolo lido, existe **apenas um** caminho possível. Não há sorte, escolha ou ambiguidade.

Características Principais

- **Sem Memória Extra:** Diferente de um computador comum, o AFD não tem um "HD" ou "Pilha" para guardar dados extras. Sua única memória são os estados.
- **Tempo Real:** Ele processa a entrada um símbolo por vez, da esquerda para a direita, e nunca volta atrás.
- **Eficiência:** É o modelo mais rápido de processamento, usado por compiladores para a fase de Análise Léxica (identificar palavras-chave, números e variáveis).

"O analisador léxico é o lugar onde o autômato finito é mais frequentemente aplicado. A tarefa do analisador é ler caracteres e agrupá-los em unidades lógicas." (**AHO et al., 2008**).

6.1 O que é um Estado?

Um estado representa uma etapa específica no processamento de uma informação. De acordo com **Diverio (2011)**, a mudança de um estado para outro é disparada por um evento externo (um símbolo do alfabeto).

Exemplo Clássico: O Semáforo

- **Estado q_0:** Verde.
- **Evento:** Sensor de tempo esgotado.
- **Transição:** O sistema muda para o Estado q_1 (Amarelo).
- **Determinismo:** No sistema ideal, não há dúvida. Se o tempo acabou e está no Verde, o *único* destino possível é o Amarelo.

O Semáforo é um exemplo de **AFD**, pois ele precisa atender a dois critérios fundamentais:

1. **Destino Único:** Para um estado atual e um evento específico, existe apenas **um** estado de destino. Se o estado é Verde e o evento é Tempo Esgotado, o sistema **sempre** vai para o Amarelo. Não há uma "escolha" ou múltiplos caminhos possíveis para o mesmo evento.
2. **Ausência de Transições Vazias (ε):** O sistema só muda de estado quando um evento real (o sensor) acontece. Em um AFN (Não-Determinístico), o sistema poderia mudar de estado "do nada", sem entrada nenhuma, o que não ocorre em um semáforo funcional.

6.2 Conversão de Padrão para Máquina: Do Texto ao Fluxo Lógico

Como transformamos uma regra como **[0-9]+** em código? O processo segue a **Construção de Thompson** (para converter Expressões Regulares em AFN) e o **Algoritmo de Subconjuntos** (para converter AFN em AFD).

O que é a Construção de Thompson? (O rascunho)

Imagine que você está desenhando um mapa de trilhas. Para a regra **[0-9]+**, a Construção de Thompson cria um caminho que diz:

- "Comece aqui."
- "Leia um número e vá para o próximo ponto."
- "Se quiser ler outro número, volte para o ponto anterior (isso é o +)."

O problema é que esse mapa pode ter "atalhos invisíveis" (as transições ε). Isso cria um **AFN (Não-Determinístico)**, que é como um mapa onde uma única placa indica dois caminhos diferentes. O computador odeia dúvida, ele não sabe qual escolher.

O que é o Algoritmo de Subconjuntos? (A limpeza)

Como o computador não lida bem com dúvidas (AFN), nós usamos esse algoritmo para "limpar" o mapa. Ele transforma o mapa confuso em um **AFD (Determinístico)**.

- Ele agrupa todas as possibilidades.
- O resultado é um fluxo onde, para cada caractere, só existe **uma** opção de caminho.

O Fluxo de Decisão na Prática

Vamos ver o "esqueleto" de como a regra **[0-9]+** vira um programa:

Componente	O que acontece na prática?
S0 (Estado Inicial)	O programa começa a rodar e fica "esperando" o primeiro caractere.
S1 (Aceitação)	Se você digitar 5, o programa diz: "Ok, recebi um dígito, vou para o estado **S1 (Aceitação)**".
Laço (Loop)	Se você digitar outro número (ex: 2), o programa continua no mesmo estado. Se você digitar um caractere não numérico, o programa vai para Erro

Matriz de Transição

Estado Atual	Caractere [0-9]	Outro	Descrição do Estado
S0 (Início)	S1	Erro	Aguardando o primeiro dígito.

S1* (Aceitação)	S1	Erro	Já li um dígito, posso ler mais ou parar.
Erro	Erro	Erro	Entrada inválida (não começou com número).

O Fluxo de Decisão:

1. **Estado Inicial (S_0):** O ponto onde a leitura começa.
2. **Transições:** Ao ler '1', o autômato move para o estado "S_1 (Aceitação)".

Como isso vira código?

Basicamente, o compilador transforma esse desenho de estados em um comando **switch/case** ou **if/else** dentro de um **loop while**.

Exemplo mental do código:

```
estado = "INICIAL"
for caractere in "123":
    if estado == "INICIAL" and caractere.isdigit():
        estado = "LENDO_NUMERO"
    elif estado == "LENDO_NUMERO" and caractere.isdigit():
        estado = "LENDO_NUMERO" # Continua aqui

if estado == "LENDO_NUMERO":
    print("Sucesso: Isso é um Token de Número!")
```

Exemplo Prático com Lark e "Máquina de Estados"

O Lark abstrai a criação do autômato, mas podemos ver sua lógica ao definir tokens que possuem estados parecidos, como os operadores **+** e **++**.

```
from lark import Lark

# Note como o Lark precisa decidir entre '+' (SOMA) e '++' (INCREMENTO)
# Isso exige que o autômato "olhe para frente"
gramatica_automato = """
    start: (INCREMENTO | SOMA | NUMERO)+

    INCREMENTO: "++"
    SOMA: "+"
```

```
    NUMERO: /[0-9]+/

    %import common.WS
    %ignore WS
"""

parser = Lark(gramatica_automato, parser='lalr')
# O parâmetro parser='lalr' gera um autômato de estados eficiente

texto = "10 + ++"
print(f"Analisando: {texto}")
for token in parser.lex(texto):
    print(f"Token: {token.type:12} | Valor: {token.value}")
```

maquina_estados.py

Desafio: O que acontece se o autômato ler apenas um + seguido de um espaço? Ele entra no estado de aceitação de **SOMA** ou **aguarda o segundo +**?

A regra **Maximal Munch** (também conhecida como *Longest Match* ou "Maior Casamento") é um princípio fundamental utilizado pelos analisadores léxicos para resolver ambiguidades durante a formação de tokens.

De forma simples: **O compilador sempre tentará formar o maior token possível a partir dos caracteres que ele está lendo.**

Imagine o operador **++** (incremento) em linguagens como C++ ou Java. Se o compilador ler **++**, ele tem duas opções:

1. Reconhecer como **um único token** de INCREMENTO.
2. Reconhecer como **dois tokens** de SOMA (+ seguido de +).

Pela regra do **Maximal Munch**, ele não para no primeiro +. Ele continua "mastigando" o texto até que o próximo caractere não faça mais parte de um token válido. Como ++ é um token válido e maior que apenas +, ele escolhe o INCREMENTO.

Exemplo Prático com Lark: "O Filtro de Segurança Numérica"

Imagine que você está criando uma linguagem para um sistema de criptografia onde **apenas números primos** são aceitos como chaves de entrada. Se o usuário digitar um número composto, o sistema deve recusar.

Passo 1: A Gramática (Sintaxe)

Primeiro, o Lark verifica se o que foi digitado é, de fato, um número inteiro positivo.

```
from lark import Lark, Transformer

# Gramática simples: aceita apenas um número por vez
gramatica_primos = """
    ?start: entrada
    entrada: NUMBER

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

Passo 2: O Transformer (Lógica Semântica)

Aqui entra a "mágica". O Transformer pega o resultado da análise sintática e aplica a lógica matemática de números primos em Python.

```
class ValidadorPrimos(Transformer):
    def entrada(self, tokens):
        # Converte o token NUMBER (texto) para int
        numero = int(tokens[0])

        if numero < 2:
            return f"ERRO: {numero} não é primo (menor que 2)."

        # Lógica de verificação de número primo
        for i in range(2, int(numero**0.5) + 1):
            if numero % i == 0:
                return f"RECUSADO: {numero} é composto (divisível por {i})."

        return f"ACEITO: {numero} é um número primo!"

# --- Execução do Sistema ---
parser = Lark(gramatica_primos)
validador = ValidadorPrimos()

def testar_chave(texto):
    try:
        # 1. Faz o Parsing (Sintaxe)
        arvore = parser.parse(texto)
        # 2. Faz a Transformação (Lógica/Semântica)
        resultado = validador.transform(arvore)
        print(resultado)
    except Exception as e:
        print(f"ERRO DE SINTAXE: Entrada inválida.")
```

```
# Testes
testar_chave("1")   # Não primo
testar_chave("13")  # Primo
testar_chave("15")  # Composto
testar_chave("abc") # Erro de Sintaxe
```

Por que dividimos assim?

De acordo com **Aho et al. (2008)**, um compilador é dividido em fases para manter a organização e a eficiência:

1. **Análise Léxica/Sintática (Lark):** Verifica se os caracteres formam um número.
2. **Análise Semântica (Transformer):** Verifica se o número atende a requisitos lógicos (ser primo). Esta parte pode ser mais "pesada" computacionalmente.

Se tentássemos fazer o Lark verificar se é primo apenas com regras de gramática (sem Python), precisaríamos de uma "Gramática Sensível ao Contexto" extremamente complexa, o que tornaria o processo ineficiente.

DESAFIO

O código acima aceita apenas um número por vez. **Você conseguiria modificar a gramática para que ela aceite uma lista de números separados por vírgula (ex: 13, 7, 20) e valide cada um deles?**

I. Questões de Múltipla Escolha

1. Um Autômato Finito Determinístico (AFD) é caracterizado por:

A) Permitir múltiplas transições para o mesmo caractere a partir de um estado.

B) Ter sempre transições vazias (ϵ) entre os estados.

C) Possuir exatamente uma transição de saída para cada símbolo do alfabeto em cada estado.

D) Não possuir estados de aceitação.

2. Segundo Aho et al. (2008), qual algoritmo é comumente usado para transformar uma Expressão Regular em um Autômato Finito Não-Determinístico (AFN)?

A) Algoritmo de Dijkstra.

B) Construção de Thompson.

C) Eliminação de Gauss.

D) QuickSort.

3. O que representa um "Círculo Duplo" em um diagrama de estados de um autômato?

A) O estado inicial da máquina.

B) Um erro léxico irrecuperável.

C) Um estado de aceitação (reconhecimento de token).

D) Uma transição nula.

4. A principal vantagem de um AFD sobre um AFN na implementação de um compilador é:

A) O AFD ocupa menos memória RAM.

B) O AFD é mais fácil de desenhar à mão.

C) O AFD garante que a decisão de transição seja única e rápida ($O(n)$).

D) O AFD permite reconhecer gramáticas ambíguas.

5. O processo de converter um AFN para um AFD é conhecido como:

A) Minimização de estados.

B) Construção de Subconjuntos (Subset Construction).

C) Compilação Reversa.

D) Interpretação Dinâmica.

6. Se um autômato está no estado inicial e lê um caractere que não possui transição prevista, ocorre um:

A) Loop infinito.

B) Estado de aceitação forçado.

C) Erro léxico.

D) Descarte do caractere e avanço automático.

7. Na hierarquia de Chomsky, os Autômatos Finitos são as máquinas que reconhecem:

A) Linguagens Sensíveis ao Contexto.

B) Linguagens Livres de Contexto.

C) Linguagens Regulares (Tipo 3).

D) Linguagens Recursivamente Enumeráveis.

8. O conceito de "Maximal Munch" (Maior Casamento) dita que o autômato deve:

A) Parar assim que encontrar o primeiro estado de aceitação.

B) Consumir o maior número possível de caracteres que formem um token válido.

C) Ignorar todos os números e focar apenas em letras.

D) Sempre preferir tokens mais curtos.

9. Em um diagrama de transição, as arestas (setas) representam:

A) A memória do computador.

B) A entrada (caractere lido) que provoca a mudança de estado.

C) O valor final da variável.

D) O nome do arquivo fonte.

10. De acordo com Menezes (2011), a diferença fundamental entre AFN e AFD reside no:

A) Número de estados de aceitação.

B) Determinismo das transições para um dado símbolo.

C) Tipo de hardware onde o código será executado.

D) Uso de cores no diagrama.

II. Perguntas Reflexivas

1. **Determinismo vs. Flexibilidade:** Por que convertemos um AFN (mais fácil de criar a partir de RegEx) em um AFD (mais complexo de desenhar) para a execução final do compilador?

2. **Limites da Máquina:** Um Autômato Finito consegue "contar" parênteses aninhados (ex: saber se ((())) está balanceado)? Por que isso exige uma máquina mais poderosa (Autômato de Pilha)?

Unidade 7: Projeto Prático I – Gerador Léxico

7.1 Planejamento do Analisador: Definindo a Mini-Linguagem

O planejamento de um compilador exige definir não apenas os blocos (tokens), mas as **dependências** entre eles. De acordo com **Aho et al. (2008)**, a análise semântica é responsável por verificar se as frases, embora gramaticalmente corretas, fazem sentido lógico no contexto em que estão inseridas.

O Conceito de Sensibilidade ao Contexto

Em uma gramática comum, a regra **A→bc** acontece independente do que está ao redor de **A**. Em uma **GSC**, a regra é **αAβ→αΓβ**. Ou seja, A só vira Γ se estiver "abraçado" por α e β.

7.1.1 A Regra Livre de Contexto (Context-Free)

$$A \rightarrow bc$$

Como se lê: "O símbolo não-terminal **A** produz (ou deriva) a sequência **bc**."

O que significa na prática:

- Nesta regra, não importa o que está escrito antes ou depois de **A**. Ele sempre pode ser substituído por **bc**.
- É como uma regra de etiqueta: "Toda vez que você vir um [Soma], você pode trocá-lo por [Número] + [Número]". O ambiente ao redor não muda a regra.

7.2.2. A Regra Sensível ao Contexto (Context-Sensitive)

$$\alpha A \beta \rightarrow \alpha \Gamma \beta$$

Como se lê: A produz **Γ** (Gama) somente no contexto de **α** (Alfa) à esquerda e **β** (Beta) à direita.

O que significa na prática:

- Aqui, o símbolo **A** é "tímido". Ele só tem permissão para se transformar em **Γ** se estiver exatamente "abraçado" por **α** e **β**.
- **α** e **β** representam o **Contexto**. No nosso projeto do "Gestor de Memória", o contexto **α** seria o comando ALLOC. O símbolo **A** (o uso da variável) só se torna válido (**Γ**) se o ALLOC (**α**) aparecer antes dele no histórico do programa.

Símbolo	Nome Técnico	Significado no Compilador
→	Produção / Derivação	"Transforma-se em" ou "Resulta em".
A,B,C	Não-Terminais	Variáveis ou estruturas (ex: comando, expressao).
a,b,c	Terminais	O texto final (ex: if, 10, ;).
α,β	Contexto	O que vem antes e depois na linha de código.
Γ	Substituição	O novo conteúdo gerado após a validação.

7.2 Implementação com Lark (Mão na Massa)

Projeto 1: Analisador de Compliance de Identidade (SP vs RJ)

Neste cenário, o número do documento (RG) é "viciado" pelo estado. Se o estado é SP, o número DEVE começar com 11, se o estado é RJ, o número DEVE começar com 21.

```
from lark import Lark, Transformer, v_args

# ============================================================================
# 1. DEFINIÇÃO DA GRAMÁTICA (A "ESTRUTURA" DO CÓDIGO)
# ============================================================================
# Usamos a notação EBNF. Regras minúsculas geram árvores, MAIÚSCULAS geram texto.
gramatica_id = """
    # O ponto de entrada (root) da nossa linguagem.
    # O '?' diz ao Lark: "se houver só 1 filho, não crie um nó extra na árvore".
    ?start: documento

    # A regra principal: um documento é composto por um ESTADO, um ":" e um NUMERO.
    documento: ESTADO ":" NUMERO

    # Terminais (Tokens):
    ESTADO: "SP" | "RJ"      // Aceita apenas as strings literais SP ou RJ
    NUMERO: /[0-9]+/         // Aceita um ou mais dígitos de 0 a 9

    # Importações e IGNORE:
    %import common.WS        // Importa a definição de espaços em branco (Whitespace)
    %ignore WS               // Ignora espaços entre os tokens (ex: "SP : 11")
"""

# ============================================================================
# 2. IMPLEMENTAÇÃO DO TRANSFORMER (A "INTELIGÊNCIA" / SEMÂNTICA)
# ============================================================================
# O Transformer percorre a árvore de baixo para cima, executando funções
# que tenham o mesmo nome das regras da gramática.
class ValidadorCompliance(Transformer):

    # @v_args(inline=True) "desempacota" os filhos da regra 'documento'
    # transformando-os em argumentos diretos (estado, numero) da função.
    @v_args(inline=True)
    def documento(self, estado, numero):
        # Como ESTADO e NUMERO são terminais (maiúsculos),
        # eles chegam aqui como objetos 'Token', que convertemos para String.
        sigla_estado = str(estado)
        num_completo = str(numero)

        # --- GRAMÁTICA SENSÍVEL AO CONTEXTO (GSC) ---
        # Aqui o valor de um campo (estado) valida o formato do outro (numero).
```

```
        # Verificação para São Paulo (Prefixo 11)
        if sigla_estado == "SP" and not num_completo.startswith("11"):
            return f"❌ ERRO DE COMPLIANCE: Estado SP não aceita prefixo {num_completo[:2]}"

        # Verificação para Rio de Janeiro (Prefixo 21)
        if sigla_estado == "RJ" and not num_completo.startswith("21"):
            return f"❌ ERRO DE COMPLIANCE: Estado RJ não aceita prefixo {num_completo[:2]}"

        # Se passar pelas validações, retorna a mensagem de sucesso
        return f"☑ SUCESSO: Documento {sigla_estado} válido com número {num_completo}"

    # O método 'start' recebe o resultado do seu único filho (documento)
    # e simplesmente o repassa para cima, limpando a saída final.
    def start(self, children):
        return children[0]

# ==============================================================================
# 3. EXECUÇÃO E TESTES
# ==============================================================================

# Inicializa o 'Motor' do Lark com a nossa gramática
parser = Lark(gramatica_id)

# Inicializa o nosso 'Fiscal' (Transformer)
validador = ValidadorCompliance()

# Função auxiliar para testar entradas e imprimir o resultado limpo
def testar(texto):
    # 1. O parser transforma o texto em uma Árvore de Sintaxe
    arvore_sintatica = parser.parse(texto)
    # 2. O validador percorre a árvore aplicando as regras lógicas
    resultado_final = validador.transform(arvore_sintatica)
    print(resultado_final)

# Execução dos testes práticos
print("--- RELATÓRIO DE VALIDAÇÃO ---")
testar("SP : 11999")   # Deve dar SUCESSO
testar("RJ : 21888")   # Deve dar SUCESSO
testar("SP : 21000")   # Deve dar ERRO (Prefixo do RJ em SP)
testar("RJ : 11000")   # Deve dar ERRO (Prefixo de SP no RJ)
```

analise_lexica_1.py

Projeto 2: Compilador HDL (Hardware Description Language)

Enquanto em uma linguagem como Python você pode escrever x = 10 em quase qualquer lugar, em hardware (HDL - *Hardware Description Language*), as instruções dependem do estado físico dos pinos.

Iremos simular um **Barramento (Bus)**. Imagine um fio compartilhado por vários componentes. Se dois componentes tentarem escrever nesse fio ao mesmo tempo, ocorre um curto-circuito. O objetivo deste compilador é impedir que o programador escreva um código que "frite" o hardware.

O Conceito: Sensibilidade ao Contexto no Hardware

A regra de ouro deste projeto é:

> O comando **WRITE** só é um **"token válido"** se o contexto anterior estabeleceu que o pino **ENABLE** está em nível **HIGH** (Ligado).

Se o pino estiver LOW, a instrução WRITE é gramaticalmente correta (escrita com as letras certas), mas **semanticamente proibida** (o contexto não permite).

A Estrutura da Mini Linguagem

Para o Lark processar isso, dividimos a linguagem em duas partes:

- **Léxico/Sintaxe (A Forma):** Define que o código deve ter o formato STATUS seguido de AÇÃO.
- **Semântica (O Contexto):** Onde o Python verifica se o STATUS permite a AÇÃO.

Por que isso é um salto na Hierarquia de Chomsky?

Neste projeto, o comando WRITE é como uma "instrução protegida".

1. O **Analisador Léxico** (Unidade 6) identifica as palavras ENABLE, HIGH e WRITE.
2. O **Analisador Sintático** vê que a ordem está correta.
3. O **Analisador Sensível ao Contexto** (nosso Transformer) diz: "Pode até estar escrito certo, mas o estado do hardware (contexto) proíbe essa ação agora".

```
from lark import Lark, Transformer

# --- 1. GRAMÁTICA ---
gramatica_hdl = """
    ?start: comando

    # Um comando é a união de um estado de pino e uma ação de escrita
    comando: status_pino acao

    # Definimos os estados possíveis do hardware
    status_pino: "ENABLE:HIGH" -> liga
               | "ENABLE:LOW"  -> desliga

    # A ação de escrita no barramento
    acao: "WRITE" -> escrita

    %import common.WS
    %ignore WS
"""

# --- 2. TRANSFORMER (O FISCAL DE HARDWARE) ---
class VerificadorHDL(Transformer):
    # Esta variável guarda o estado "físico" simulado do pino
    def __init__(self):
        self.pino_ativado = False
```

```
    # Quando o Lark encontra "ENABLE:HIGH", ele chama esta função
    def liga(self, _):
        self.pino_ativado = True
        return "PINO: LIGADO"

    # Quando o Lark encontra "ENABLE:LOW", ele chama esta função
    def desliga(self, _):
        self.pino_ativado = False
        return "PINO: DESLIGADO"

    # Quando o Lark encontra "WRITE", ele chama esta função
    def escrita(self, _):
        return "OPERACAO: ESCREVER"

    # Esta função final une o contexto (pino) com a ação (escrita)
    def comando(self, children):
        # children[0] é o resultado de 'status_pino'
        # children[1] é o resultado de 'acao'

        if self.pino_ativado == False:
              return "❌ ERRO CRÍTICO: Tentativa de WRITE com ENABLE:LOW. Bloqueado
para evitar curto-circuito!"

        return "☑ SUCESSO: Escrita realizada com segurança no barramento."

# --- 3. TESTES ---
hdl_parser = Lark(gramatica_hdl)
fiscal = VerificadorHDL()

# Teste 1: Caminho Seguro
print(fiscal.transform(hdl_parser.parse("ENABLE:HIGH WRITE")))

# Teste 2: Caminho Perigoso (O Compilador deve barrar)
# Reiniciamos o fiscal para o novo teste
fiscal = VerificadorHDL()
print(fiscal.transform(hdl_parser.parse("ENABLE:LOW WRITE")))
```

analise_lexica_2.py

Projeto 3: O Gestor de Memória (Alocação vs. Liberação)

Este é um clássico de linguagens como C e Rust. A regra é: **você não pode liberar (free) uma variável que não foi alocada (malloc)**, e **não pode usar uma variável após ela ter sido liberada**.

A "Inteligência": O Transformer mantém um conjunto (set) de variáveis "vivas".

Exemplo de Código:

```
ALLOC x;
USE x;
FREE x;
USE x;  <-- ERRO GSC: Variável x não está mais na memória!
```

Por que é GSC? O token **USE x** é sintaticamente correto em qualquer lugar, mas sua validade depende do *contexto* (se houve um ALLOC anterior e nenhum FREE).

```
from lark import Lark, Transformer

#
================================================================================
# 1. A GRAMÁTICA (SINTAXE - A FORMA)
#
================================================================================
# Aqui o Lark apenas verifica se o usuário escreveu as palavras corretamente
# e na ordem certa. Ele NÃO sabe se a variável existe ou não.
gramatica_memoria = """
    # Regra inicial: o programa é composto por um ou mais comandos (+)
    ?start: programa
    programa: comando+

    # Define quais tipos de comandos nossa linguagem aceita
    ?comando: alloc_cmd | free_cmd | use_cmd

    # Estrutura específica de cada comando (Ex: ALLOC x;)
    alloc_cmd: "ALLOC" ID ";"
    free_cmd: "FREE" ID ";"
    use_cmd: "USE" ID ";"

    # Regra para identificadores (nomes de variáveis)
    # Deve começar com letra/underline e pode conter números
    ID: /[a-z_][a-z0-9_]*/

    # Ignora espaços, tabs e quebras de linha entre os comandos
    %import common.WS
    %ignore WS
"""

#
================================================================================
# 2. O TRANSFORMER (SEMÂNTICA/CONTEXTO - A INTELIGÊNCIA)
#
================================================================================
# Esta classe é o "cérebro" do compilador. Ela percorre a árvore gerada pelo
# Lark e aplica as regras lógicas de Sensibilidade ao Contexto (GSC).
class GestorSemantico(Transformer):
    def __init__(self):
        # Nossa 'Tabela de Símbolos' simplificada.
        # Um 'set' em Python é perfeito para rastrear o que está "vivo" na
memória.
        self.variaveis_alocadas = set()

    # Executado sempre que o Lark encontra a regra 'alloc_cmd'
    def alloc_cmd(self, nodes):
        var_name = str(nodes[0]) # Pega o nome da variável (ID)
        if var_name in self.variaveis_alocadas:
            return f"⚠ AVISO: Variável '{var_name}' já estava alocada.
Reatribuindo..."

        self.variaveis_alocadas.add(var_name) # Adiciona ao contexto de
'vivas'
        return f"☑ SUCCESS: Memória reservada para '{var_name}'."
```

```
    # Executado sempre que o Lark encontra a regra 'free_cmd'
    def free_cmd(self, nodes):
        var_name = str(nodes[0])
        # VERIFICAÇÃO DE CONTEXTO: Só podemos liberar o que foi alocado antes!
        if var_name not in self.variaveis_alocadas:
            return f"❌ ERRO GSC: Tentativa de liberar '{var_name}' que
NUNCA foi alocada!"

        self.variaveis_alocadas.remove(var_name) # Remove do contexto
        return f"🗑️ SUCCESS: Memória de '{var_name}' liberada."

    # Executado sempre que o Lark encontra a regra 'use_cmd'
    def use_cmd(self, nodes):
        var_name = str(nodes[0])
        # O "Segmentation Fault" ocorre quando acessamos algo que não está no
contexto 'vivo'
        if var_name not in self.variaveis_alocadas:
            return f"💣 ERRO CRÍTICO: 'Segmentation Fault' ao tentar usar
'{var_name}'. Contexto inválido!"

        return f"💎 SUCCESS: Acessando valor de '{var_name}'."

    # Junta todos os resultados dos comandos em um relatório final legível
    def programa(self, comandos):
        return "\n".join(comandos)

#
# ============================================================================
# 3. EXECUÇÃO (O PROCESSO DE COMPILAÇÃO)
#
# ============================================================================

# Criamos o Analisador (Parser) usando a gramática definida
parser = Lark(gramatica_memoria)

# Criamos o Fiscal Semântico (Transformer)
gestor = GestorSemantico()

# Exemplo de código-fonte que será "mastigado" pelo nosso compilador
codigo_fonte = """
    ALLOC buffer;
    USE buffer;
    FREE buffer;
"""

try:
    # PASSO 1: Análise Sintática (Gera a Árvore)
    arvore = parser.parse(codigo_fonte)

    # PASSO 2: Análise Semântica (Transforma a Árvore em Resultados Lógicos)
    resultado = gestor.transform(arvore)

    print("--- RELATÓRIO DO COMPILADOR (UNIDADE 7) ---")
    print(resultado)
except Exception as e:
    # Captura erros de escrita (ex: faltou o ponto e vírgula)
    print(f"❌ Erro de Sintaxe (Escrita incorreta): {e}")
```

analise_lexica_3.py

Desafio Prático da Unidade 7

Agora que você tem o Gestor de Memória básico, tente adicionar uma nova regra: **"O sistema deve ter um limite de memória de 3 variáveis simultâneas."**

- *Dica:* No método alloc_cmd, verifique se o tamanho de self.variaveis_alocadas é maior que 3 antes de adicionar uma nova. Se for, retorne um erro de "Out of Memory".

I. Questões de Múltipla Escolha

1. Em linguagens de programação, a "Sensibilidade ao Contexto" refere-se à capacidade do compilador de validar uma instrução baseando-se em eventos ou declarações prévias. No projeto do Gestor de Memória, o erro "Segmentation Fault" é classificado como:

A) Um erro léxico, pois a palavra "USE" foi escrita incorretamente.

B) Um erro sintático, pois falta um ponto e vírgula na instrução.

C) Um erro semântico/contextual, pois a variável existe na gramática, mas não está ativa no estado atual da memória.

D) Um erro de hardware, indicando que a memória RAM do computador pifou.

2. Considere a regra de produção $\alpha A\beta \rightarrow \alpha\Gamma\beta$. Se aplicarmos esta lógica ao validador de Compliance de IDs (SP/RJ), o que representa o símbolo α?

A) O número do documento isolado.

B) O contexto anterior, ou seja, o Estado (SP ou RJ) que dita a regra para o próximo token.

C) O símbolo de dois pontos (":").

D) Qualquer caractere aleatório digitado pelo usuário.

3. Por que utilizamos a classe Transformer do Lark para implementar a lógica de Hardware (HDL) e Memória?

A) Porque o Lark não consegue ler textos sem ajuda do Python.

B) Porque gramáticas Livres de Contexto (CFG) sozinhas não conseguem validar dependências entre valores de tokens distintos.

C) Para converter o código automaticamente para C++.

D) Para aumentar a velocidade de digitação do programador.

4. No código do Gestor de Memória, o que aconteceria se o usuário digitasse ALLOC x (sem o ponto e vírgula)?

A) O Transformer corrigiria o erro automaticamente.

B) O programa exibiria "Success", pois o ponto e vírgula é opcional.

C) O Lark dispararia uma exceção de erro sintático antes mesmo de chegar ao Transformer.

D) A variável seria alocada, mas com um nome inválido.

5. A regra do "Maior Casamento" (Maximal Munch) é crucial em analisadores léxicos. Se definirmos os tokens + e ++, qual será a saída para a entrada +++?

A) Três tokens de soma (+, +, +).

B) Um token de incremento (++) e um de soma (+).

C) Um erro léxico, pois +++ não existe.

D) Dois tokens de incremento (++, ++).

6. Analise a afirmação: "Um AFD (Autômato Finito Determinístico) é suficiente para validar se uma variável foi liberada antes de ser usada". Esta afirmação é:

A) Verdadeira, pois o AFD tem memória infinita.

B) Falsa, pois validar dependências de longo prazo entre tokens exige uma análise de contexto (Semântica) além do poder de um AFD puro.

C) Verdadeira, desde que o código tenha menos de 10 linhas.

D) Falsa, porque AFDs só funcionam com números binários.

7. O uso de set() no __init__ do GestorSemantico simula qual componente real de um compilador?

A) O Gerador de Código Binário.

B) A Tabela de Símbolos (Symbol Table).

C) O Otimizador de Consultas SQL.

D) O Interpretador de Comandos de Voz.

8. Na Hierarquia de Chomsky, as linguagens que o Lark processa nativamente (antes do Transformer) são de Tipo 2. Ao usar o Transformer para validar o contexto, "subimos" para qual nível?

A) Tipo 3 (Regulares).

B) Tipo 2 (Livres de Contexto).

C) Tipo 1 (Sensíveis ao Contexto).

D) Tipo 0 (Irrestritas).

9. No projeto de HDL, a instrução ENABLE:LOW WRITE resulta em erro. Esse bloqueio ocorre em qual fase da compilação?

A) Análise Léxica.

B) Análise Sintática.

C) Análise Semântica.

D) Geração de Código.

10. Qual a vantagem de usar @v_args(inline=True) em um método do Transformer?

A) Deixa o código mais lento, porém mais seguro.

B) Facilita o acesso aos dados, passando os filhos da árvore como argumentos individuais da função.

C) Permite que o Lark ignore todos os erros de digitação.

D) Faz com que o compilador rode dentro do navegador web.

II. Perguntas Reflexivas

1. **Memória vs. Regras:** Se pudéssemos criar uma regra sintática para cada combinação possível de alocação de memória, a gramática seria infinita. Como a separação entre Sintaxe (Lark) e Semântica (Python) resolve esse problema de escala?

2. **Segurança em Hardware:** No projeto HDL, o compilador barra o erro antes da execução. Como isso impacta o custo de manutenção de um sistema em comparação a descobrir o erro apenas quando o hardware queima durante o uso?

III. Exercícios Práticos

1. **Desafio do Saldo:** Implemente uma mini linguagem de banco com dois comandos: DEPOSIT <valor>; e WITHDRAW <valor>;. Use um Transformer para garantir que o usuário nunca saque mais do que o saldo atual disponível.

2. **Bloqueio de Variáveis Reservadas:** No Gestor de Memória, altere o método alloc_cmd para que ele proíba a alocação de variáveis com nomes de sistema, como admin, root ou config.

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

8.1 O Papel do Analisador Sintático: A Ordem das Palavras

O **Analisador Sintático** (ou *Parser*) é o componente que verifica se a sequência de tokens gerada pelo Analisador Léxico obedece às **regras estruturais da linguagem**.

De acordo com **Aho et al. (2008)**, sua função é construir uma representação intermediária, geralmente uma **Árvore Sintática**, que define a hierarquia do programa. Se o léxico garante que as palavras existem no dicionário, a sintaxe garante que a gramática está correta.

"A análise sintática transforma uma lista plana de tokens em uma estrutura em árvore, onde a precedência e a aninhamento de comandos tornam-se explícitos." (**MENEZES, 2011**).

8.2 Análise Descendente (Top-Down)

A estratégia **Top-Down** (de cima para baixo) começa pelo símbolo inicial da gramática (o objetivo mais abstrato, como programa) e tenta "expandi-lo" repetidamente até que ele coincida com os tokens reais lidos do código-fonte.

O processo mental do Parser Top-Down:

1. **Início:** "Eu preciso encontrar um programa."
2. **Expansão:** "Um programa começa com uma **declaração_de_variável**."
3. **Verificação:** "O primeiro token que recebi é VAR? Se sim, a previsão está correta. Prossiga."

🌳 Entendendo o Conceito: Árvore de Decisão e Hierarquia

Antes de programarmos, precisamos entender como o computador "enxerga" a organização das regras.

1. O Conceito de "Expandir"

Imagine a regra para um **Sanduíche**. O Parser não vê o sanduíche pronto; ele vê o projeto:

- **Regra Geral:** Sanduíche → Pão + Recheio + Pão.
- Se o que chegar na "esteira" for Pão, Queijo e Pão, a planta baixa foi seguida com sucesso.

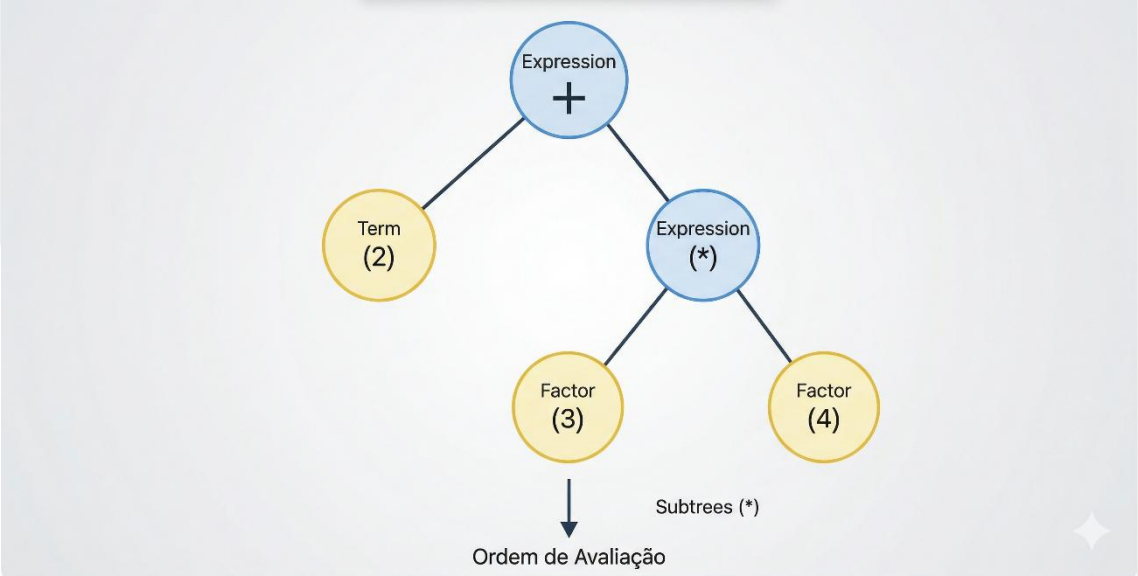
2. A Árvore de Decisão (O "OU" Lógico)

Muitas vezes, o parser precisa decidir qual caminho seguir. Se a regra diz que um Comando pode ser um IF ou um WHILE, ele cria uma ramificação.

- Se ele lê a palavra if, ele "desce" pelo galho da estrutura condicional e ignora o galho da repetição.

A imagem a seguir mostra a **Árvore de Sintaxe Abstrata (AST)** para a expressão 2 + 3 * 4.

Note como a estrutura não é plana; ela força a multiplicação (*) a ficar em um nível inferior (mais profundo) que a soma (+). Isso obriga o computador a resolver primeiro o "galho" de baixo (3 * 4 = 12) para depois levar o resultado para o "tronco" principal (2 + 12 = 14).



Como ler esta árvore? (Análise Pedagógica)

1. **A Raiz (Top):** É o símbolo mais genérico. O compilador prevê: "Eu espero encontrar uma **Soma**".
2. **O "OU" Lógico:** A regra de soma pode ser reescrita como Termo + Termo. O Parser desce para a esquerda e para a direita.
3. **Precedência:** Note que a operação 3 * 4 está contida dentro da ramificação direita. Na análise **Top-Down**, o computador só consegue subir o resultado da ramificação direita (12) após ter finalizado toda a sub-árvore que ela gerou.

Aho et al. (2008) explicam que o analisador sintático não apenas valida o código, mas cria essa estrutura intermediária que será usada posteriormente pela análise semântica para gerar código de máquina.

3. Hierarquia (Pai e Filhos)

Na árvore sintática, a posição de um item define sua importância. Em uma operação como 2 + 3 * 4, a multiplicação fica em um "galho" mais baixo (mais profundo) que a soma. Isso força o computador a resolver primeiro a multiplicação para depois levar o resultado para cima, garantindo a **precedência matemática**.

Exercícios Práticos (LARK + Python)

Estes exercícios focam em visualizar a árvore descendente gerada pelo Lark.

Exercício 1: A Árvore da Atribuição

Crie uma gramática que aceite atribuições (ex: x = 50). Use o método .pretty() para visualizar a árvore.

```
from lark import Lark
gramatica = """
    start: atribuicao
    atribuicao: ID "=" NUMERO
    ID: /[a-zA-Z_]+/
    NUMERO: /[0-9]+/
    %import common.WS
    %ignore WS
"""
parser = Lark(gramatica)
print(parser.parse("x = 50").pretty())
```

analise_sintatica_1.py

Saída:

```
start
  atribuicao
    x
    50
```

Exercício 2: Árvore de Decisão (Ações de Robô)

Crie uma gramática onde o start pode ser um comando de MOVER ou PARAR.

- **Entrada:** MOVER 10
- **Objetivo:** Ver como a árvore ramifica a partir do nó comando.

```
from lark import Lark

# ============================================================
# A GRAMÁTICA DA TOMADA DE DECISÃO
# ============================================================
gramatica_robo = """
    # O 'start' é o nosso ponto de origem (Raiz da Árvore).
    start: comando

    # A REGRA DE OU (Bifurcação):
    # O símbolo '|' funciona como uma encruzilhada.
    # O Parser olha para o Token e decide: "Vou para o galho MOVER ou
para o PARAR?"
    comando: mover | parar

    # Galho 1: Exige a palavra 'MOVER' seguida de um número.
    mover: "MOVER" NUMERO

    # Galho 2: Exige apenas a palavra 'PARAR'. É uma folha simples.
    parar: "PARAR"

    # Definições de Terminais (Tokens finais)
    NUMERO: /[0-9]+/

    %import common.WS
    %ignore WS
"""

# Inicializamos o motor de análise (O Arquiteto)
parser = Lark(gramatica_robo)

# --- ANÁLISE DO CAMINHO "MOVER" ---
print("--- DECISÃO 1: MOVER ---")
# O Parser lê "MOVER", vê que bate com a regra 'mover' e desce por esse
galho.
print(parser.parse("MOVER 10").pretty())

# --- ANÁLISE DO CAMINHO "PARAR" ---
print("\n--- DECISÃO 2: PARAR ---")
# O Parser lê "PARAR", ignora o galho 'mover' e desce pelo galho 'parar'.
print(parser.parse("PARAR").pretty())
```

analise_sintatica_2.py

Saída:

```
--- DECISÃO 1: MOVER ---
start
  comando
    mover       10


--- DECISÃO 2: PARAR ---
start
  comando
    parar
```

Exercício 3: Hierarquia de Operações

Implemente uma gramática simples de soma e multiplicação.

- **Entrada:** 2 + 3 * 4
- **Objetivo:** Observar que o nó da multiplicação aparece "dentro" (abaixo) do nó da soma no print da árvore.

```
from lark import Lark

# ============================================================
# A ESTRUTURA DA PRECEDÊNCIA (HIERARQUIA)
# ============================================================
# Regra de ouro: O que você quer que seja resolvido PRIMEIRO
# deve estar no nível mais BAIXO (mais longe do 'start').
# ============================================================

gramatica_matematica = """
    # O ponto de entrada tenta resolver uma 'expressao'
    ?start: expressao

    # NÍVEL 1: SOMA (Menor prioridade)
    # Uma expressão pode ser um termo sozinho OU uma soma.
    # O '?' antes do nome diz ao Lark: "Se não houver '+' aqui,
    # não crie o nó 'expressao', passe direto para o 'termo'".
    ?expressao: termo
              | expressao "+" termo   -> soma

    # NÍVEL 2: MULTIPLICAÇÃO (Média prioridade)
    # Como 'expressao' chama 'termo', o computador é obrigado a
    # entrar aqui para tentar entender o que é o 'termo'.
    ?termo: fator
          | termo "*" fator          -> multiplicacao

    # NÍVEL 3: FATOR (Maior prioridade / Base)
    # Aqui é onde estão os números ou o que está entre parênteses.
```

```
    # O parênteses força o computador a voltar para o topo (expressao).
    ?fator: NUMBER                  -> numero
          | "(" expressao ")"

    # Definições de Tokens
    %import common.NUMBER
    %import common.WS
    %ignore WS
"""

# Inicializamos o motor de análise
parser = Lark(gramatica_matematica)

# Exemplo: 2 + 3 * 4
# O Parser Top-Down lê o '2', vê o '+', mas antes de somar,
# ele "desce" para ver se o que vem depois do '+' é algo
# que 'manda' mais (como o '*' do 3 * 4).
try:
    print("--- ANÁLISE HIERÁRQUICA DE 2 + 3 * 4 ---")
    arvore = parser.parse("2 + 3 * 4")
    print(arvore.pretty())
except Exception as e:
    print(f"Erro na análise: {e}")
```

analise_sintatica_3.py

Saída:

```
--- Árvore do Exercício 3 (2 + 3 * 4) ---
soma
  numero        2
  multiplicacao
    numero      3
    numero      4
```

O que está acontecendo por trás das cortinas?

1. A "Descida" (Top-Down): O Parser começa em start. Ele pergunta: "Isso é uma soma?". Ele vê o +. Mas para completar a soma, ele precisa processar o lado direito (termo).
2. A "Emboscada" da Multiplicação: Ao entrar em termo, ele descobre que existe um *. O Parser percebe que o 3 e o 4 estão "presos" em uma regra de multiplicação.
3. A Resolução: Como o nó da multiplicacao é um filho do nó da soma, o computador é obrigado a resolver a multiplicação primeiro para poder entregar o resultado "para cima", para o pai (a soma).

Por que usamos o -> nome (Aliases)?

Note que usamos **-> soma** e **-> multiplicacao**. No Lark, isso se chama **Alias**. Ele serve para que, no print do .pretty(), você veja o nome da operação em vez de apenas nomes genéricos como expressao. Isso facilita muito o debug de árvores complexas.

Exercício 4: Múltiplos Comandos

Altere a gramática do robô para aceitar vários comandos em sequência (ex: MOVER 10 MOVER 20).

- **Objetivo:** Entender como a regra descendente agrupa uma lista de filhos sob um mesmo pai.

```
from lark import Lark

gramatica_multiplos = """
    start: comando+

    comando: "MOVER" NUMERO
           | "GIRAR" NUMERO

    NUMERO: /[0-9]+/
    %import common.WS
    %ignore WS
"""

parser = Lark(gramatica_multiplos)
entrada = """
        MOVER 10
        GIRAR 90
        MOVER 20
    """

print("--- Árvore do Exercício 4 ---")
print(parser.parse(entrada).pretty())
```

analise_sintatica_4.py

Saída:

```
-- Árvore do Exercício 4 ---
start
  comando       10
  comando       90
  comando       20
```

Neste exercício, usamos o **operador +** para permitir que o start tenha vários filhos do tipo comando.

Exercício 5: Analisador de Configurações (A Ordem importa)

Neste cenário, uma ferramenta de *Deploy* exige que o arquivo siga esta ordem:

1. Definição do Host
2. Definição da Porta
3. Comando de RUN

```python
from lark import Lark

# ============================================================
# GRAMÁTICA DE CONFIGURAÇÃO (SEQUENCIAL)
# ============================================================
gramatica_config = """
    # O objetivo final é uma 'configuracao'
    start: configuracao

    # A REGRA DA ORDEM:
    # O parser SÓ aceita se vier primeiro o HOST, depois PORTA, depois START.
    configuracao: host_decl porta_decl start_cmd

    host_decl: "HOST" TEXTO ";"
    porta_decl: "PORT" NUMBER ";"
    start_cmd: "RUN" ";"

    # Definição de tipos básicos
    TEXTO: /"[^"]*"/
    %import common.NUMBER
    %import common.WS
    %ignore WS
"""

parser = Lark(gramatica_config)

# --- CASO 1: ORDEM CORRETA ---
print("--- Teste 1: Ordem Correta ---")
config_correta = 'HOST "servidor.com"; PORT 8080; RUN;'
print(parser.parse(config_correta).pretty())

# --- CASO 2: ORDEM INCORRETA ---
print("\n--- Teste 2: Ordem Incorreta (PORT antes de HOST) ---")
config_errada = 'PORT 8080; HOST "servidor.com"; RUN;'

try:
    parser.parse(config_errada)
except Exception as e:
    # O erro acontece porque o Parser Top-Down "olhou" para o topo da regra
    # 'configuracao' e esperava o token 'HOST', mas encontrou 'PORT'.
    print("❌ ERRO SINTÁTICO: A ordem dos fatores alterou o produto!")
    print("O compilador esperava 'HOST' no início do arquivo.")
```

analise_sintatica_5.py

Saída:

```
--- Teste 1: Ordem Correta ---
start
  configuracao
    host_decl   "servidor.com"
    porta_decl  8080
    start_cmd


--- Teste 2: Ordem Incorreta (PORT antes de HOST) ---
❌ ERRO SINTÁTICO: A ordem dos fatores alterou o produto!
O compilador esperava 'HOST' no início do arquivo
```

Exercício 6: O Mistério do IF Aninhado

Objetivo: Criar uma gramática que aceite comandos de decisão (IF) que podem conter outros comandos dentro deles. Observe como a árvore cresce para a direita (profundidade) quando colocamos um IF dentro de outro.

O Cenário

Imagine uma linguagem de automação residencial onde:

1. Você verifica uma condição: **IF sensor_presenca == 1**
2. Executa uma ação: **LIGAR luz;**
3. Ou abre outro bloco: **IF hora > 18 ...**

Exemplo de código que a linguagem deverá suportar:

```
IF movimento == 1 THEN {
    LIGAR sala;
}
```

Ou

```
IF movimento == 1 THEN {
    LIGAR sala;
    IF escuro == 1 THEN {
        LIGAR jardim;
    }
}
```

I. Questões de Múltipla Escolha

1. Qual é o principal objetivo da fase de Análise Sintática em um compilador?

A) Traduzir o código-fonte diretamente para linguagem de máquina.

B) Agrupar caracteres em unidades lógicas chamadas tokens.

C) Verificar se a sequência de tokens obedece às regras estruturais da gramática.

D) Otimizar o uso de memória das variáveis declaradas no programa.

2. A análise descendente (Top-Down) é caracterizada por:

A) Começar pelas folhas da árvore (tokens) e subir até a raiz (start).

B) Partir do símbolo inicial da gramática e tentar prever a estrutura até chegar nos tokens.

C) Ignorar a ordem dos tokens, focando apenas se eles existem no dicionário léxico.

D) Executar o código linha por linha enquanto verifica erros de lógica matemática.

3. Na visualização de uma árvore sintática com o método .pretty() do Lark, o recuo (identação) para a direita indica:

A) Que um comando foi escrito incorretamente.

B) Uma relação de hierarquia, onde o item recuado é um "filho" ou "sub-regra" do item acima.

C) Que o tempo de execução daquela linha será maior.

D) Que o token é um terminal irrelevante para o compilador.

4. Considere a expressão 2 + 3 * 4. Em uma gramática que respeita a precedência, por que a multiplicação aparece em um nível mais profundo da árvore?

A) Porque o número 4 é maior que o número 2.

B) Para que o resultado da multiplicação seja calculado primeiro e "subido" para a operação de soma.

C) Porque o caractere * vem depois do caractere + na leitura da esquerda para a direita.

D) Trata-se de um erro do Lark; em compiladores, todas as operações devem estar no mesmo nível.

5. O que acontece se um parser Top-Down espera um token ID (identificador), mas recebe um token NUMBER?

A) Ele converte o número em texto automaticamente.

B) Ele ignora o erro e pula para a próxima linha.

C) Ele dispara um erro sintático, pois a "previsão" da regra superior não foi confirmada.

D) Ele reinicia a análise léxica para tentar encontrar o erro.

6. O uso de recursividade em regras gramaticais (ex: uma regra bloco que contém outros blocos) permite ao compilador:

A) Gastar menos memória RAM durante a compilação.

B) Processar apenas comandos simples, proibindo o aninhamento.

C) Representar estruturas complexas e aninhadas, como IFs dentro de IFs.

D) Aumentar a velocidade de digitação do programador.

7. Na Hierarquia de Chomsky, as gramáticas processadas por analisadores sintáticos comuns (como o Lark) são geralmente:

A) Regulares (Tipo 3).

B) Livres de Contexto (Tipo 2).

C) Sensíveis ao Contexto (Tipo 1).

D) Irrestritas (Tipo 0).

8. O símbolo | (pipe) em uma gramática Lark representa uma "Árvore de Decisão" porque:

A) Obriga o usuário a escolher apenas uma opção de comando por programa.

B) Permite que o parser escolha entre diferentes caminhos de derivação para um mesmo nó.

C) Divide o código em dois arquivos diferentes.

D) Indica que o código deve ser executado em paralelo.

9. Qual a função do símbolo ? antes de uma regra no Lark (ex: ?start)?

A) Tornar a regra opcional.

B) "Achatar" a árvore, removendo nós desnecessários que possuem apenas um filho.

C) Indicar que o compilador está em dúvida sobre aquela regra.

D) Transformar todos os tokens daquela regra em números reais.

10. A análise sintática é frequentemente chamada de "Análise Hierárquica". Isso se deve ao fato de:

A) Os programadores seniores definirem as regras para os juniores.

B) O código ser lido de cima para baixo pelo sistema operacional.

C) As estruturas de dados (árvores) organizarem o código em níveis de subordinação e precedência.

D) O compilador ser o software de maior hierarquia no computador.

II. Perguntas Reflexivas

1. **A Ordem dos Fatores:** Por que, em uma análise Top-Down, a sequência dos tokens é vital para o sucesso do parsing, enquanto na análise léxica a ordem das palavras muitas vezes não impede a identificação dos tokens individuais?

2. **Gramática vs. Realidade:** Imagine uma gramática que aceita IF condicao THEN acao. Se o programador esquecer o THEN, o parser falhará. Reflita sobre como as linguagens modernas tentam ser "mais flexíveis" (como Python que usa : em vez de THEN) e como isso impacta o desenho da árvore sintática.

III. Exercícios Práticos (LARK + Python)

Exercício 1: O Compilador de Receitas Crie uma gramática que aceite uma sequência de passos de uma receita. A ordem obrigatória deve ser: PEGAR <ingrediente>; MISTURAR; COZINHAR;. Se o usuário tentar COZINHAR antes de PEGAR, o parser deve gerar um erro. Visualize a árvore resultante de uma receita correta.

Exercício 2: Calculadora de Áreas Hierárquica Crie uma gramática que aceite o cálculo de áreas de formas geométricas. Ex: AREA QUADRADO 5; ou AREA CIRCULO 10;. O nó raiz deve ser start, que se ramifica em forma. Observe como a árvore de decisão se comporta ao trocar QUADRADO por CIRCULO.

Revisão – Unidade 1 até Unidade 8

Bloco A: Fundamentos e Gramáticas (Unidades 1 e 2)

1. Segundo Chomsky, a Classe 2 (Linguagens Livres de Contexto) é o "ponto ideal" para definir linguagens de programação. Qual a principal razão técnica para essa escolha?

A) Elas são processadas apenas por hardware específico.

B) Permitem definir estruturas recursivas e aninhadas, como blocos if dentro de if.

C) São as únicas que permitem o uso de comentários no código.

D) Elas eliminam a necessidade de uma CPU para a execução.

E) São menos potentes que as gramáticas regulares.

2. Uma gramática G é formalmente definida pela quádrupla G=(V, Σ, P, S). O que o conjunto Σ representa na prática de um compilador?

A) O conjunto de variáveis abstratas.

B) O símbolo inicial do programa.

C) Os símbolos reais (terminais/tokens) que compõem o código.

D) O manual de instruções do processador.

E) O algoritmo de otimização de código.

3. Sobre a Árvore de Derivação (Parse Tree), é correto afirmar que:

A) É apenas uma representação visual sem impacto na execução.

B) Ela dita a precedência de operações através de sua estrutura hierárquica.

C) Serve exclusivamente para identificar erros de digitação.

D) É gerada apenas por interpretadores, nunca por compiladores.

E) Substitui a necessidade de análise léxica.

4. O processo de substituir o não-terminal mais à esquerda em cada passo da análise é a base para analisadores:

A) Bottom-Up.

B) Interpretadores de baixo nível.

C) Top-Down (LL).

D) Otimizadores de registro.

E) Geradores de código objeto.

5. A ambiguidade em uma gramática é considerada um defeito de projeto porque:

A) Torna o código-fonte muito longo.

B) Permite duas ou mais árvores de derivação para a mesma sentença, gerando dúvida no compilador.

C) Impede o uso de números reais.

D) Obriga o uso da linguagem C.

E) Reduz a velocidade da internet.

Bloco B: Arquitetura e Fluxo (Unidades 2 e 3)

6. A divisão do compilador em Front-End e Back-End permite que:

A) O compilador rode sem memória RAM.

B) O programador não precise usar variáveis.

C) O código seja traduzido apenas linha por linha.

D) O hardware seja ignorado completamente.

E) O mesmo Back-End seja reutilizado para diferentes linguagens de origem.

7. Qual fase do Front-End é responsável por verificar o "significado" (ex: compatibilidade de tipos)?

A) Análise Léxica.

B) Análise Sintática.

C) Análise Semântica.

D) Geração de Código.

E) Pré-processamento.

8. O Kernel do Sistema Operacional interage com os programas através de:

A) Chamadas de Sistema (System Calls).

B) Tradução simultânea de áudio.

C) Garbage Collection.

D) Exclusão de arquivos temporários.

E) Redimensionamento de janelas.

9. A principal diferença entre um Compilador e um Interpretador é que o Compilador:

A) Executa o código linha por linha.

B) Gera um arquivo executável independente antes da execução.

C) Não realiza análise sintática.

D) É usado apenas em Python.

E) Para a execução no primeiro erro encontrado.

10. Linguagens como Java e C# utilizam um modelo híbrido que gera:

A) Código de máquina binário puro.

B) Apenas comentários.

C) Tabelas de verdade.

D) Bytecode para ser executado em uma máquina virtual.

E) Expressões regulares complexas.

Bloco C: Ferramentas e Automação (Unidades 4 e 7)

11. Por que ferramentas como Lex/Yacc e AntLR são utilizadas na criação de compiladores?

A) Para automatizar a criação de analisadores, reduzindo erros humanos e aumentando a produtividade.

B) Para desenhar a interface do usuário.

C) Para substituir o processador.

D) Para traduzir binário em código-fonte.

E) Para gerenciar o banco de dados do sistema.

12. O algoritmo LL(*) do AntLR é considerado inovador porque:

A) Só consegue olhar 1 token à frente.

B) Possui um lookahead dinâmico, podendo olhar quantos tokens forem necessários para decidir a regra.

C) Foi escrito inteiramente em Assembly.

D) Ignora a análise sintática.

E) É exclusivo para sistemas de 8 bits.

13. A biblioteca Lark (Python) utiliza qual notação para definir gramáticas?

A) XML.

B) JSON.

C) Markdown.

D) SQL.

E) EBNF (Extended Backus-Naur Form).

14. Na Hierarquia de Chomsky, os Autômatos Finitos reconhecem linguagens de qual tipo?

A) Tipo 0 (Irrestritas).

B) Tipo 1 (Sensíveis ao Contexto).

C) Tipo 2 (Livres de Contexto).

D) Tipo 3 (Regulares).

E) Tipo 4 (Linguagens Naturais).

15. O uso do Rust no Kernel do Linux é motivado por qual recurso do seu compilador?

A) Velocidade de compilação.

B) Borrow Checker (Verificador de Empréstimos) para segurança de memória.

C) Facilidade de uso para leigos.

D) Compatibilidade com Windows 95.

E) Inexistência de análise semântica.

Bloco D: Análise Léxica e Autômatos (Unidades 5 e 6)

16. Um Token é composto por:

A) Apenas o endereço de memória.

B) Nome do token (categoria) e valor do atributo.

C) Apenas o texto escrito pelo usuário.

D) O número da linha e da coluna.

E) Um código binário aleatório.

17. Qual a diferença entre Token e Lexema?

A) Token é a regra; Lexema é o compilador.

B) Não há diferença técnica entre eles.

C) Token é a categoria lógica; Lexema é a sequência real de caracteres no código.

D) Lexema é um erro; Token é um acerto.

E) Token é usado no Back-End; Lexema no Front-End.

18. O símbolo + em uma Expressão Regular como [0-9]+ indica:

A) Que o padrão deve ocorrer uma ou mais vezes.

A) Uma operação matemática de adição.

C) Que o padrão é opcional (zero ou uma vez).

D) Um erro de sintaxe.

E) O final de uma linha de código.

19. A regra do Maximal Munch (Maior Casamento) dita que o analisador léxico deve:

A) Parar no primeiro caractere que encontrar.

B) Consumir o maior número possível de caracteres que formem um token válido.

C) Preferir sempre os tokens mais curtos.

D) Ignorar todas as letras maiúsculas.

E) Substituir espaços por sublinhados.

20. Um Autômato Finito Determinístico (AFD) diferencia-se de um Não-Determinístico (AFN) por:

A) Possuir memória infinita.

B) Não aceitar números.

C) Permitir transições vazias (ϵ).

D) Ter exatamente uma transição de saída para cada símbolo em cada estado.

E) Ser mais lento na execução.

Bloco E: Análise Sintática Top-Down (Unidade 8)

21. O papel do Analisador Sintático é:

A) Verificar se os caracteres são válidos.

B) Alocar espaço físico no processador.

C) Verificar se a ordem dos tokens obedece às regras estruturais da linguagem.

D) Traduzir o código para inglês.

E) Verificar se o computador tem vírus.

22. Na análise descendente (Top-Down), o processo inicia-se por:

A) Pelos tokens (folhas).

B) Pelo símbolo inicial da gramática (raiz).

C) Pela declaração de variáveis globais.

D) Pelo fim do arquivo (EOF).

E) Pela análise semântica.

23. Em uma árvore sintática, a profundidade de um nó em relação a outro define:

A) O custo financeiro do programa.

B) A precedência (itens mais profundos costumam ser resolvidos primeiro).

C) A cor do texto no editor de código.

D) O número de linhas do programa.

E) A ordem alfabética das variáveis.

24. O símbolo ? antes de uma regra no Lark (ex: ?start) serve para:

A) Indicar uma pergunta ao usuário.

B) Marcar a regra como opcional.

C) "Achatar" a árvore, removendo nós desnecessários com apenas um filho.

D) Transformar o resultado em booleano.

E) Iniciar um processo de depuração automática.

25. Um erro sintático ocorre quando:

A) O programador esquece de declarar uma variável.

B) O usuário digita uma senha errada.

C) O computador fica sem bateria.

D) O arquivo é salvo com a extensão errada.

E) A sequência de tokens não corresponde à "previsão" feita pelas regras gramaticais.

Bloco F: Projeto Prático e Contexto (Unidade 7)

26. No projeto do Gestor de Memória, o erro "Segmentation Fault" ocorre quando:

A) O programador escreve "ALOC" em vez de "ALLOC".

B) O sistema tenta usar uma variável que não foi alocada ou já foi liberada no contexto atual.

C) O arquivo de código é muito grande.

D) O Lark não foi instalado corretamente.

E) O HD do servidor está cheio.

27. O que diferencia uma Gramática Sensível ao Contexto (GSC) de uma Livre de Contexto (GLC)?

A) Na GSC, a substituição de um símbolo depende dos símbolos ao seu redor (contexto).

B) A GSC não permite o uso de números.

C) A GLC é mais poderosa que a GSC.

D) A GLC só funciona em computadores quânticos.

E) Não há diferença técnica entre elas.

28. O componente Transformer do Lark é utilizado para:

A) Identificar os tokens iniciais.

B) Mudar a cor da árvore no console.

C) Percorrer a árvore sintática e aplicar lógica semântica ou contextual em Python.

D) Substituir o analisador léxico.

E) Criar gráficos 3D do código.

29. No cenário de compliance, por que a validação é semântica e não apenas sintática?

A) Porque envolve o uso de cores diferentes.

B) Porque a validade do número depende do valor de outro token (o estado).

C) Porque o SP e o RJ são estados geográficos.

D) Porque o Lark só entende números se forem semânticos.

E) Porque o ponto e vírgula é obrigatório.

30. A Tabela de Símbolos é uma estrutura central que armazena:

A) Apenas os erros encontrados.

B) O código binário final.

C) O histórico de navegação do programador.

D) Informações sobre identificadores (nomes, tipos, escopos) para todas as fases do compilador.

E) As senhas de acesso ao servidor.

Unidade 9: Análise Sintática Avançada (Bottom-Up)

Elementos Fundamentais da Gramática

Uma gramática formal é o conjunto de regras que define a estrutura da linguagem. Ela utiliza os seguintes elementos:

- **Token (ou Terminal):** É a menor unidade lógica da linguagem, vinda do analisador léxico. Exemplos: palavras reservadas (if, while), operadores (+, =), identificadores (nome_variavel) e pontuação (;, (). São chamados de "terminais" porque não podem ser expandidos em nada mais.
- **Não-Terminal:** Representa uma estrutura gramatical ou uma categoria sintática. São símbolos (geralmente escritos em maiúsculas ou entre < >) que agrupam outros símbolos. Exemplo: <EXPRESSAO>, <DECLARACAO_DE_VARIAVEL>.
- **Símbolo Inicial:** É o não-terminal que representa o programa inteiro ou a unidade máxima da gramática. Toda análise começa (ou termina, no caso do Bottom-Up) tentando alcançar este símbolo.
- **Produção (ou Regra):** É a regra de substituição, escrita como A → α, onde A é um não-terminal e α é uma sequência de terminais e/ou não-terminais.

Para descrever uma linguagem formal de maneira rigorosa, utilizamos o conceito de **Gramática Formal**, que é a "fórmula" ou o conjunto de regras que define como as sentenças válidas podem ser construídas.

Na Teoria da Computação e na Engenharia de Computação, a definição mais clássica é a proposta por Noam Chomsky. Uma gramática G é definida como uma quádrupla:

$$G = (V, \Sigma, P, S)$$

V (Variáveis ou Não-Terminais): É um conjunto finito de símbolos que representam estruturas intermediárias. Eles não aparecem na frase final, servindo apenas para organizar a hierarquia (ex: <FRASE>, <EXPRESSÃO>).

Σ (Terminais ou Tokens): É o alfabeto da linguagem. São os símbolos reais que o programador digita e que o analisador léxico identifica (ex: if, +, identificador, ;).

P (Regras de Produção): É o conjunto de instruções de substituição. Elas definem como você pode transformar um Não-Terminal em uma sequência de outros símbolos. A forma geral é: **α → β** *(Onde α é substituído por β)*

S (Símbolo Inicial): É o ponto de partida de toda a derivação. Em compiladores, geralmente representa o "Programa" completo.

Exemplo Prático de uma "Fórmula" Simples

Imagine uma linguagem que só aceita somas de números. A gramática G seria:

V: {E, N}
Σ: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}
P:

- E → E + N
- E → N
- N → 0 | 1 | 2 | ... | 9

S: E

Declaração	Significado
V: {E, N}	(Variáveis ou Não-Terminais) São símbolos abstratos que representam "categorias" ou "blocos" da linguagem. Eles não aparecem no resultado final (o código que o robô lê), mas servem para organizar a estrutura. **E (Expressão):** Representa a regra maior, ou seja, o que define uma conta de soma completa. **N (Número):** Representa a unidade básica de valor que será usada na conta.
Σ: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +}	São os "tokens" reais, os símbolos finais que o usuário realmente digita no teclado. Eles são chamados de terminais porque a derivação termina neles. Define que esta linguagem só entende os dígitos de **0 a 9** e o símbolo de adição **+**. Qualquer outro caractere (como - ou *) causaria um erro léxico aqui.
E → E + N	Esta regra permite a **recursividade**. Ela diz que uma Expressão pode ser formada por outra Expressão somada a um Número. É o que permite contas infinitas como 1 + 2 + 3...
E → N	Esta é a "regra de parada". Ela diz que uma Expressão também pode ser apenas um único Número sozinho.
N → 0 | 1 | 2 | ... | 9	O símbolo | significa **OU**. Diz que o símbolo abstrato N deve ser substituído por qualquer um dos dígitos de 0 a 9
S: E	Indica por onde o compilador deve começar a "pensar". Diz que tudo o que for analisado deve tentar se encaixar na regra de **E (Expressão)**. Se o analisador conseguir transformar os números digitados de volta para o símbolo E, o código é considerado válido.

9.1 Análise Ascendente (Bottom-Up)

A análise **Bottom-Up** (ou ascendente) funciona de forma inversa à Top-Down. O objetivo é construir a **Árvore de Sintaxe Abstrata (AST)** partindo das **folhas (tokens)** em direção à **raiz** (o símbolo inicial da gramática).

A **Árvore de Sintaxe Abstrata (AST - *Abstract Syntax Tree*)** é uma das estruturas de dados mais importantes dentro de um compilador. Ela é uma representação em árvore da estrutura sintática simplificada do código-fonte.

9.1.1. O que a torna "Abstrata"?

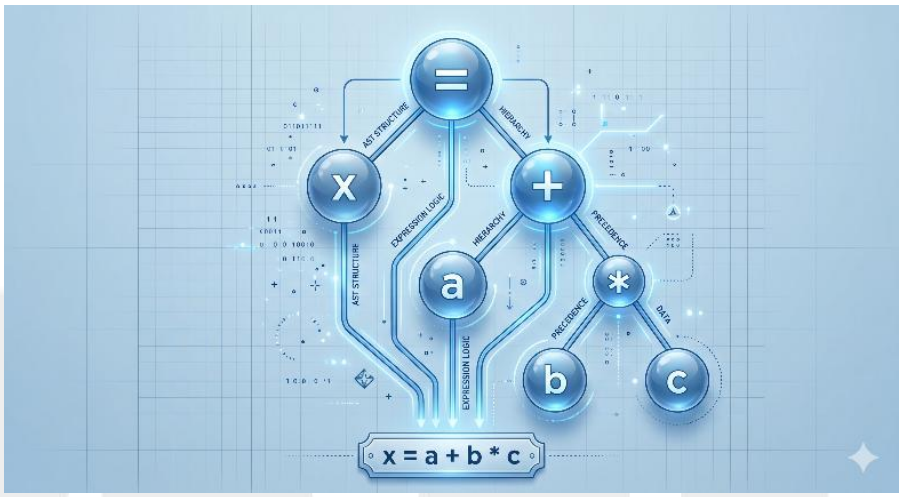
Diferente da Árvore de Derivação (Sintaxe Concreta), que contém todos os detalhes da gramática (como parênteses, pontos e vírgulas, palavras-chave como if ou then), a **AST foca apenas na essência lógica** do programa.

- **Na Árvore Concreta:** Aparecem todos os símbolos lidos pelo parser.
- **Na AST:** Aparecem apenas os operadores e operandos. Por exemplo, os parênteses em uma expressão (2 + 3) não precisam de um nó na árvore, pois a própria hierarquia da árvore já define que a operação dentro deles deve ser feita primeiro.

9.1.2. Estrutura e Composição

- **Nós Internos:** Representam os operadores ou construções de controle (como uma soma +, uma atribuição =, ou um laço while).
- **Folhas:** Representam os operandos ou identificadores (como números 10, 20 ou nomes de variáveis x, y).
- **Exemplo prático:** Para a expressão: **x = a + b * c**

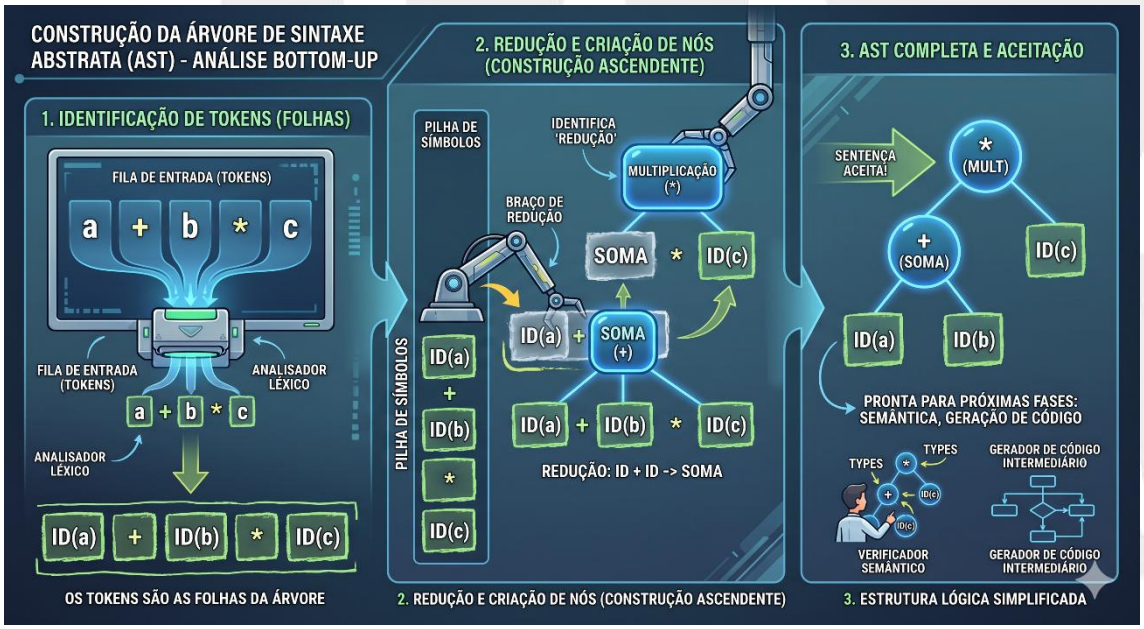
A AST teria o sinal de **=** como raiz. O filho esquerdo seria **x**. O filho direito seria o nó **+**. O nó **+** teria como **filho esquerdo a** e como **filho direito** o nó *****. Finalmente, o nó ***** teria **b** e **c** como folhas.



9.1.3. Por que ela é usada na Análise Sintática Avançada (Bottom-Up)?

Na análise **Bottom-Up**, a AST é construída conforme as **reduções** acontecem:

1. O analisador encontra os tokens (folhas).
2. Ao realizar uma **redução** (ex: transformar **ID + ID** em uma **Soma**), o compilador cria um **novo nó** na **AST**, ligando os termos que acabaram de ser reduzidos como filhos deste novo nó.
3. Ao final do processo, quando a sentença é aceita, a AST está completa e pronta para as próximas fases.



9.1.4. Importância para as fases seguintes

A AST é o **"contrato"** entre o Front-End e o Back-End do compilador:

- **Análise Semântica:** O compilador percorre a AST para verificar se os tipos de dados são compatíveis (ex: se você não está somando um número a uma string).
- **Geração de Código Intermediário:** É muito mais fácil gerar código a partir de uma árvore limpa (AST) do que do texto original ou de uma árvore de derivação poluída com símbolos gramaticais.

Análise Léxica (identificação de tokens), **Sintática** (montagem da árvore), **Semântica** (conversão de tipos)

A árvore sintática é definida pelas regras da gramática formal **G = (V, Σ, P, S).** Se a gramática diz que uma **Atribuição** é composta por **id = Expressão**, qualquer analisador, seja ele **Top-Down** ou **Bottom-Up**, é obrigado a seguir essa "planta".

- **Top-Down:** Começa no telhado (raiz) e tenta encontrar as fundações (tokens).
- **Bottom-Up:** Começa nas fundações (tokens) e tenta chegar ao telhado (raiz).

Independentemente de quem começou primeiro, o prédio (a árvore) construído será idêntico se eles seguirem o mesmo projeto.

9.2 Comparação de Estratégias

A escolha entre Top-Down e Bottom-Up depende da complexidade da linguagem e das ferramentas disponíveis.

Quando usar cada uma:

- **Top-Down (LL):** Ideal para linguagens simples, onde a estrutura pode ser prevista facilmente. É mais fácil de implementar manualmente (via Descida Recursiva).
- **Bottom-Up (LR):** É o padrão para linguagens de programação modernas (C++, Java, Python). Embora difícil de escrever manualmente, é implementado via geradores automáticos como **Yacc** ou **Bison**.

Exemplo Prático: Um programa que seja capaz de resolver o cálculo

```
X = (127 * 5) / 100
```

Definindo a gramática: G = (V, Σ, P, S)

1. Conjunto de Variáveis ou Não-Terminais (V)

São os símbolos que aparecem do lado esquerdo das setas e representam as categorias sintáticas.

V = {Programa, Atribuição, Expressão, Termo, Fator, Primário}

2. Conjunto de Terminais ou Alfabeto (Σ)

São os tokens finais (os "átomos" da linguagem) que o programador realmente digita.

Σ = {id, =, /, *, (,), número}

3. Conjunto de Regras de Produção (P)

Aqui listamos as substituições lógicas da sua gramática. "Termo" foi ajustado para seguir a lógica de precedência comum em compiladores:

1. Programa → Atribuição
2. Atribuição → id = Expressão
3. Expressão → Termo/Fator | Termo
4. Termo → Fator * Primário | Fator
5. Fator → (Expressão) | Primário
6. Primário → número

4. Símbolo Inicial (S)

É o ponto de partida de toda a análise sintática.

S = Programa

Estratégia 1: Análise Top-Down (Descendente)

Conceito: Parte-se do símbolo inicial da gramática (raiz) e tenta-se "expandir" as regras para corresponder aos tokens encontrados na entrada (folhas), movendo-se de cima para baixo. É uma abordagem preditiva.

Tokens de entrada: id(X) = (número(127) * número(5)) / número(100)

Descrição Textual da Construção:

1. **Início na Raiz:** O analisador começa com o símbolo inicial: <Programa>.
2. **Expansão do Programa:** Para corresponder à entrada, <Programa> é expandido usando a Regra 1 para <Atribuição>.
3. **Expansão da Atribuição:** <Atribuição> é expandida usando a Regra 2. A árvore agora tem a estrutura: **<Atribuição> → id(X) = <Expressão>.**
    - *Símbolos id(X) e = são reconhecidos na entrada.*
4. **Expansão da Expressão:** O analisador agora precisa expandir <Expressão> para corresponder ao restante da entrada, começando com (. Olhando para a gramática, <Expressão> deve se expandir para <Termo>, e <Termo> deve se expandir para <Fator>, que por fim se expande para (<Expressão>) usando a Regra 5.
    - *A árvore agora tem um "subgalho" que reconhece (.*
5. **Análise dentro dos Parênteses:** Agora, o analisador recomeça o processo de expansão para a <Expressão> que está *dentro* dos parênteses, visando encontrar 127 * 5.
    - <Expressão> se expande para <Termo>.
    - <Termo> se expande para <Termo> * <Primário> (para acomodar a multiplicação).
    - O <Termo> à esquerda se expande até <Primário>, que se expande para o número(127). *Reconhecido.*
    - O * é *reconhecido*.
    - O <Primário> à direita se expande para o número(5). *Reconhecido.*
6. **Fechamento dos Parênteses:** O analisador volta à expansão da Regra 5 e reconhece o símbolo).
7. **Análise do Restante da Expressão:** O analisador volta para a <Expressão> principal (Fase 4). O que foi analisado até agora foi apenas a parte antes do /.

<Termo> (que derivou os parênteses) deve agora se expandir para <Termo> / <Fator> (Regra 3, adaptada para a hierarquia correta).

- O / é *reconhecido*.
- <Fator> se expande até <Primário>, que se expande para número(100). *Reconhecido.*

8. **Finalização:** Todos os tokens foram consumidos e a árvore foi totalmente conectada de volta à raiz <Programa>.

Estratégia 2: Análise Bottom-Up (Ascendente)

Conceito: Parte-se dos tokens encontrados na entrada (folhas) e tenta-se "reduzir" esses símbolos para os não-terminais da gramática, movendo-se de baixo para cima até chegar ao símbolo inicial (raiz). É uma abordagem baseada em empilhar e reduzir.

Tokens de entrada: id(X) = (número(127) * número(5)) / número(100)

Descrição Textual da Construção:

1. **Empilhamento Inicial:** O analisador começa empilhando os primeiros tokens: id(X), =, (. Nada pode ser reduzido ainda.
2. **Identificação de Primário:** O token número(127) é empilhado. O analisador reconhece que número é o lado direito da Regra 6.
   - **Ação:** Cria um nó <Primário> e liga o número(127) a ele. (O 127 na pilha é substituído por <Primário>).
3. **Reduções Sequenciais:**<Primário> pode ser reduzido a <Fator> (Regra 5), e <Fator> a <Termo> (Regra 4).
   - **Ação:** Cria nós intermediários <Fator> e <Termo>, formando o galho do 127.
4. **Processamento da Multiplicação:** O token * é empilhado. O token número(5) é empilhado.
   - **Ação:** O número(5) é reduzido a <Primário> (Regra 6).
   - O analisador agora vê na pilha: id(X), =, (, <Termo>, *, <Primário>. Ele reconhece isso como o lado direito da Regra 4.
   - **Ação:** Cria um nó <Termo>, que se torna o "pai" dos nós <Termo>, * e <Primário> já criados para 127 * 5.
5. **Fechamento dos Parênteses:** O token) é empilhado. O analisador agora vê na pilha: id(X), =, (, <Termo>,). Isso corresponde à Regra 5.

- **Ação:** Cria um nó <Fator> que liga os nós (, <Termo>, e) como seus filhos.
- A estrutura (127 * 5) foi consolidada em um único nó <Fator>.

6. **Processamento da Divisão:** O nó <Fator> é reduzido a <Termo> (Regra 4).
   - O token / é empilhado.
   - O token número(100) é empilhado e reduzido até <Primário>.
   - A pilha agora contém: id(X), =, <Termo>, /, <Primário>.
7. **Consolidação da Expressão:** O analisador reconhece a estrutura de divisão (Regra 3).
   - **Ação:** Cria um nó <Expressão> que liga os nós <Termo>, /, e <Primário> como filhos.
8. **Finalização:** A pilha contém: id(X), =, <Expressão>. Isso corresponde à Regra 2.
   - **Ação:** Cria o nó <Atribuição>, ligando id(X), = e <Expressão>.
   - <Atribuição> é reduzido a <Programa> (Regra 1). *Raiz alcançada.*

Resumo das Diferenças

Característica	Top-Down	Bottom-Up
Ponto de Partida	Raiz (<Programa>)	Folhas (id, =, número, etc.)
Ação Principal	Expansão de regras de cima para baixo.	Redução de tokens de baixo para cima.
Preditividade	Tenta prever a estrutura antes de ver os tokens.	Espera ver os tokens antes de decidir a estrutura.
Hierarquia	Constrói os nós "pais" antes dos "filhos".	Constrói os nós "filhos" antes dos "pais".
Precedência	Definida pela ordem de expansão das regras.	Definida pela ordem em que os nós são reduzidos (o que é reduzido antes fica mais fundo).

Por que o Bottom-Up é mais poderoso?

1. **Recursão à Esquerda:** Analisadores ascendentes lidam naturalmente com recursão à esquerda (E → E + T), que causa loops infinitos em analisadores descendentes.

2. **Atraso na Decisão:** O analisador ascendente "vê" o corpo da produção antes de decidir a qual não-terminal ela pertence. Isso permite que ele reconheça uma gama muito maior de gramáticas (Gramáticas LR).
3. **Detecção de Erros:** Geralmente detectam erros sintáticos o mais cedo possível, assim que o token lido não pode fazer parte de nenhum *handle* válido.

I. Questões de Múltipla Escolha

QUESTÃO 1 - A Árvore de Sintaxe Abstrata (AST) é uma representação intermediária fundamental que simplifica a estrutura do código-fonte para as fases posteriores do compilador. Diferente da Árvore de Derivação (Sintaxe Concreta), a AST foca na essência lógica do programa.

Avalie as afirmações a seguir sobre a AST:

I. A AST omite detalhes gramaticais como parênteses obrigatórios e delimitadores (ponto e vírgula), pois a hierarquia da árvore já define a precedência e a separação de instruções.

II. Na análise bottom-up, a AST é construída de forma incremental: a cada redução realizada, um novo nó é criado ligando os termos reduzidos como seus filhos.

III. A AST é idêntica à Árvore de Derivação, diferenciando-se apenas pelo algoritmo utilizado para sua geração.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e II, apenas.

D) II e III, apenas.

E) I, II e III.

QUESTÃO 2 - A estratégia de análise Bottom-Up é frequentemente considerada mais poderosa que a Top-Down (descendente) para linguagens de programação modernas. Um dos principais motivos técnicos para essa superioridade é a forma como o algoritmo trata certas construções gramaticais que seriam problemáticas para algoritmos como o LL(1).

Qual característica gramatical é suportada nativamente pela análise Bottom-Up sem a necessidade de transformações complexas na gramática?

A) Ambiguidade inerente.

B) Fatoração à esquerda.

C) Recursão à esquerda.

D) Linguagens de Tipo 0 (Irrestritas).

E) Ausência de símbolos terminais.

QUESTÃO 3 - As ferramentas modernas de geração de parsers, como Yacc, Bison e o módulo LALR do Lark (utilizado em nossos laboratórios), baseiam-se na teoria de analisadores LR. Esses analisadores são capazes de reconhecer praticamente todas as construções de linguagens de programação e detectar erros sintáticos assim que o primeiro token inválido é lido.

Sobre a análise LR, assinale a opção correta:

A) Ela reconstrói a derivação mais à esquerda (leftmost) do código-fonte.

B) Ela utiliza exclusivamente o conjunto FIRST para decidir reduções.

C) Ela é menos eficiente que a análise descendente por exigir retrocesso (backtracking).

D) Ela constrói a árvore sintática partindo das folhas em direção à raiz.

E) Ela exige que a gramática seja convertida para a Forma Normal de Greibach.

Questões Discursivas

QUESTÃO 6 - Explique a relação entre as ações de Redução no algoritmo Shift-Reduce e a construção da Árvore de Sintaxe Abstrata (AST) na análise ascendente. Como a AST facilita o trabalho do Analisador Semântico em comparação com o código-fonte bruto?

QUESTÃO 7 - Diferencie, do ponto de vista da Engenharia de Compiladores, as estratégias Top-Down e Bottom-Up. Em sua resposta, aborde por que o uso de geradores automáticos (como o Lark em modo LALR) é preferível para a implementação de analisadores ascendentes em vez da codificação manual por descida recursiva. xx

Unidade 10: Projeto Prático II – Gerador Sintático

Nesta etapa, utilizaremos a biblioteca **Lark** (Python) pela sua versatilidade em suportar tanto análise *Top-Down* quanto *Bottom-Up*, facilitando a visualização da Árvore de Sintaxe Abstrata (AST).

10.1 Construção da Gramática: Definindo as Regras de Estrutura

Construir uma gramática é definir a "Constituição" da sua linguagem. Se a Análise Léxica (Unidade 5) lidava com as peças do LEGO, a Análise Sintática lida com o **manual de montagem**.

Como declarar uma Função?

Vamos definir uma regra para uma linguagem hipotética onde uma função deve ter: uma palavra-chave def, um nome, parâmetros entre parênteses e um bloco de código entre chaves.

Exemplo de Gramática em notação EBNF (Lark):

```
// Regra de alto nível
start: funcao

// Definição da estrutura da função
funcao: "def" nome "(" parametros ")" bloco

// Sub-regras
parametros: (nome ("," nome)*)?
bloco: "{" instrucao* "}"
instrucao: nome "=" expressao ";"

// Definições léxicas (Tokens)
nome: /[a-zA-Z_][a-zA-Z0-9_]*/
expressao: /[0-9]+/

%import common.WS
%ignore WS
```

Explicação Detalhada:

1. **start: funcao**: Define que o ponto de entrada do nosso analisador é a regra de função.
2. **A Regra funcao**: Estabelece a ordem obrigatória. Se o programador esquecer o def ou os parênteses, o parser acusará erro.
3. **O Operador * e ?**:
    - parametros? significa que os parâmetros são opcionais.

- instrucao* significa que o bloco pode ter zero ou várias instruções.

4. **A Hierarquia**: Note que a funcao é "pai" do bloco, que por sua vez é "pai" das instrucoes. Isso formará a nossa árvore.

10.2 Integração: Unindo o Analisador Léxico ao Sintático

A integração é o processo onde o fluxo de caracteres vira um fluxo de tokens, que por sua vez vira uma árvore. No Lark, essa união é nativa, mas entender o que acontece "sob o capô" é vital para a Engenharia.

O Fluxo de Dados

1. **Entrada Bruta:** "def soma(a, b) { x = 10; }"
2. **Léxico (Scanner):** Identifica que def é uma palavra reservada, soma é um ID, (é um símbolo, etc.
3. **Sintático (Parser):** Recebe esses tokens e tenta encaixá-los nas regras da Unidade 10.1.

Exemplo Prático de Integração em Python:

```python
from lark import Lark

# Nossa gramática integrada
minha_gramatica = """
    start: funcao
    funcao: "def" ID "(" parametros ")" bloco
    parametros: (ID ("," ID)*)?
    bloco: "{" instrucao* "}"
    instrucao: ID "=" NUMBER ";"

    ID: /[a-zA-Z_][a-zA-Z0-9_]*/
    NUMBER: /[0-9]+/

    %import common.WS
    %ignore WS
"""

# Criando o Parser (Usando LALR - Bottom-Up para maior performance)
parser = Lark(minha_gramatica, parser='lalr')

# Código a ser testado
codigo_fonte = "def calcular(x, y) { resultado = 100; }"

try:
    arvore = parser.parse(codigo_fonte)
    print("Sucesso! Árvore Sintática Gerada:")
    print(arvore.pretty())
except Exception as e:
    print(f"Erro de Sintaxe: {e}")
```

declaração_linguagem.py

Por que integrar é desafiador?

O maior erro na integração é a **ambiguidade**. Se a sua gramática permitir que o mesmo código seja interpretado de duas formas (ex: um if sem um else claro), o integrador falhará.

Nesta fase, o analisador sintático atua como um "filtro de qualidade": ele só permite que a próxima fase (Análise Semântica) receba estruturas que façam sentido gramatical.

I. Questões de Múltipla Escolha

Questão 1: Na integração entre o analisador léxico e o sintático, qual é o papel primordial do léxico em relação ao parser?

A) O léxico decide a ordem das operações matemáticas.

B) O léxico simplifica a entrada, transformando sequências de caracteres em unidades lógicas (tokens) para que o parser trabalhe sobre símbolos e não caracteres individuais.

C) O léxico substitui a árvore sintática por uma tabela de símbolos.

D) O léxico é responsável por verificar se uma variável foi declarada.

E) Não há integração; as duas fases ocorrem de forma isolada em arquivos diferentes.

Questão 2: Ao definir a regra bloco: "{" instrucao* "}", o uso do metacaractere * indica:

A) Que o bloco deve obrigatoriamente ter uma instrução.

B) Que o bloco é um comentário.

C) Que o bloco pode conter zero ou mais repetições da regra instrucao.

D) Que a instrução é um ponteiro de memória.

E) Que o compilador deve multiplicar as instruções.

II. Questões Discursivas

Questão 3: Explique como a definição de uma gramática para declaração de funções ajuda a prevenir erros de execução em uma linguagem de programação.

Questão 4: Descreva o que ocorre em um analisador sintático integrado quando ele recebe um token que não segue a sequência esperada pela regra de produção atual (ex: um número onde se esperava um parêntese).

Unidade 11: Análise Semântica – O Significado

11.1 Verificação de Coerência: O código pode estar escrito certo, mas faz sentido?

Dizemos que uma linguagem tem **Sintaxe Correta** quando ela segue as regras da gramática, mas isso não garante que ela funcione.

Analogia com a Língua Portuguesa:

- *Sintaticamente correto:* "O computador bebeu o café da manhã." (Sujeito + Verbo + Objeto).
- *Semanticamente incorreto:* Computadores não bebem café. Não faz sentido.

No mundo da programação, a Análise Semântica verifica regras que as Gramáticas Livres de Contexto não conseguem capturar sozinha. As principais verificações são:

1. **Verificação de Tipos (Type Checking):** Você está tentando somar um texto com um número? Ex: x = "Robson" + 10;. Sintaticamente está perfeito (ID = STRING + NUM), mas semanticamente é um erro na maioria das linguagens.
2. **Verificação de Fluxo de Controle:** Existe um break fora de um laço de repetição?
3. **Verificação de Escopo:** Você está tentando usar uma variável que foi declarada dentro de uma função diferente?
4. **Unicidade:** Você declarou duas variáveis com o mesmo nome no mesmo bloco?

11.2 Tabela de Símbolos: O "caderno de anotações" do compilador

A **Tabela de Símbolos** é a estrutura de dados central que o compilador utiliza para "lembrar" de tudo o que o programador declarou. Sem ela, a análise semântica seria impossível.

O que ela armazena?

Para cada identificador (nome de variável, função ou classe), a tabela guarda:

- **Nome:** O lexema original (ex: salario_base).
- **Tipo:** int, float, string, bool.
- **Escopo:** Em qual bloco ela "vive" (Global, Função A, Loop B).
- **Endereço:** Onde ela será guardada na memória (fase posterior).

Exemplo Prático de Tabela de Símbolos: considere o código python.

```
x = 10
def calcular(y):
    return x + y
```

Tabela de Símbolos Simplificada:

Identificador	Categoria	Tipo	Escopo
x	Variável	int	Global
calcular	Função	-> int	Global
y	Parâmetro	int	Local (calcular)

Integração no Lark (Python)

Para realizar a análise semântica no nosso projeto, utilizamos o **Transformer**. Ele percorre a árvore gerada na Unidade 10 e consulta uma Tabela de Símbolos (que podemos implementar como um dicionário Python).

I. Questões de Múltipla Escolha

Questão 1: Considere a instrução `float x = "texto";`. Do ponto de vista de um compilador, essa instrução apresenta:

A) Um erro léxico, pois "texto" não é um número.

B) Um erro sintático, pois a ordem dos tokens está invertida.

C) Um erro semântico, pois há uma incompatibilidade de tipos na atribuição.

D) Sucesso total, pois a sintaxe está correta.

E) Um erro de otimização de código.

Questão 2: A Tabela de Símbolos é preenchida e consultada em diversas fases. Qual a importância do campo "Escopo" nesta tabela?

A) Definir a cor da variável no editor de texto.

B) Garantir que o programa seja traduzido para inglês.

C) Permitir que o compilador diferencie variáveis com o mesmo nome, mas declaradas em contextos diferentes (ex: uma variável local vs uma global).

D) Aumentar a velocidade do processador.

E) Excluir variáveis que não foram usadas.

II. Questões Discursivas

Questão 3: Por que as Gramáticas Livres de Contexto (vistas na Unidade 8) não são suficientes para garantir que um programa seja executado sem erros de lógica de tipos?

Questão 4: Imagine que você está projetando um compilador e o usuário tenta chamar uma função que ainda não foi declarada. Explique como a Tabela de Símbolos ajudaria o compilador a identificar esse erro.

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

12.1 Por que um código intermediário?

O grande desafio no desenvolvimento de compiladores clássicos reside na variedade de origens e destinos. Se quisermos construir compiladores para **M** linguagens de programação diferentes (como Python, Rust, Java e C) voltados para **N** arquiteturas de processadores distintas (como x86_64, ARM, RISC-V e MIPS), a codificação direta exigiria o desenvolvimento de **M X N** compiladores independentes.

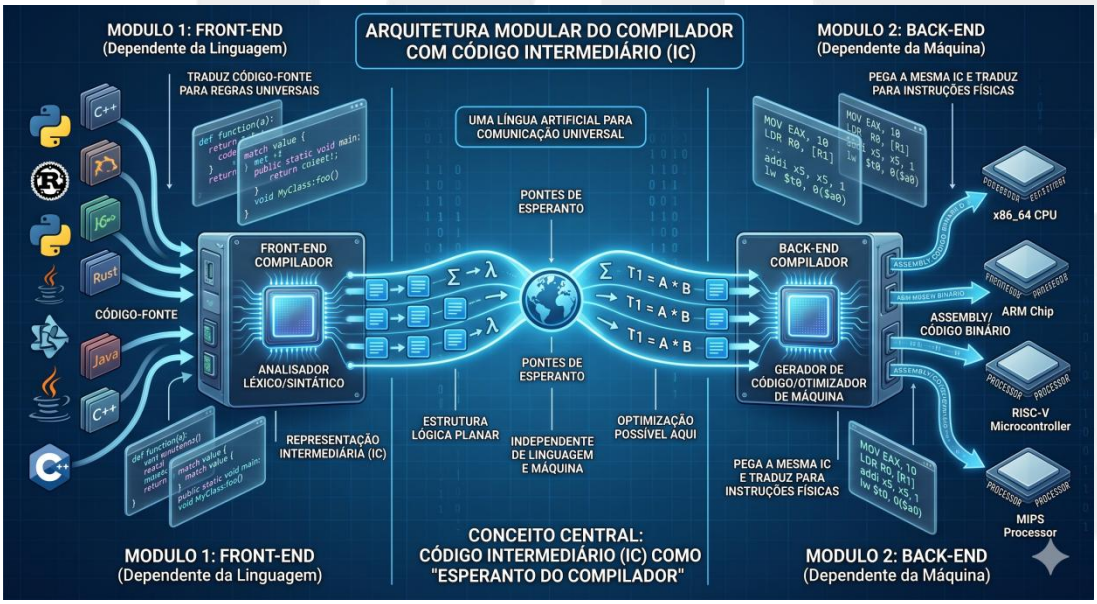
Para mitigar esse problema matemático e computacional, a engenharia de sistemas introduziu o conceito de **Código Intermediário** (IC - *Intermediate Code*), frequentemente comparado ao *Esperanto* — uma língua artificial criada para permitir a comunicação universal.

Com essa abordagem, dividimos o compilador de forma modular:

1. **Front-End (Dependente da Linguagem):** Traduz o código-fonte de uma linguagem específica para a representação intermediária.

2. **Back-End (Dependente da Máquina):** Pega essa mesma representação intermediária e a traduz para as instruções físicas da CPU de destino (Assembly/Código Binário).

Dessa forma, o custo de desenvolvimento cai drasticamente de M X N para **M + N**. Se uma nova arquitetura de hardware surgir no mercado, precisamos programar apenas um novo *Back-End* que leia o código intermediário universal.



Formas Comuns de Código Intermediário

- **Código de Três Endereços (TAC - *Three-Address Code*):** É uma representação linear muito parecida com o Assembly, onde cada instrução possui, no máximo, um operador e três operandos (dois argumentos e um destino).

Exemplo de expressão: `x = a + b * c` *Tradução em TAC:*

```
t1 = b * c
t2 = a + t1
x = t2
```

12.2 Verificação de Tipos e Fluxo Avançada

Embora a Análise Semântica (Unidade 11) já tenha validado os aspectos iniciais de tipos de dados com o auxílio da Tabela de Símbolos, a fase de Código Intermediário atua de forma refinada, estruturando as regras lógicas que controlam o comportamento das estruturas de decisão e repetição, como o `if` e o `while`.

I. Verificação e Coerção de Tipos na Geração

Durante a quebra de expressões complexas em Código de Três Endereços, o compilador insere conversões implícitas (coerção de tipos), se a linguagem permitir. Se uma operação em TAC envolver um número inteiro e um número flutuante (`t1 = int_var + float_var`), o compilador gera uma instrução intermediária explícita de conversão (`t2 = int_to_float(int_var)`) antes de realizar a operação aritmética.

II. Linearização do Fluxo de Controle (`if` e `while`)

O processador de um computador executa instruções sequencialmente, uma linha após a outra, a menos que encontre um desvio de fluxo (*jump*). O papel da representação intermediária é traduzir as estruturas estruturadas de alto nível em blocos lógicos planos e lineares usando rótulos (*labels*) e desvios condicionais.

- A Estrutura **`if-else`**:

Python

```
if (x > 0):
    y = 1
else:
    y = 2
```

Equivalência em Código Intermediário (TAC):

```
if x <= 0 goto L1    # Desvio condicional (Inversão da lógica)
y = 1
goto L2              # Salta o bloco else
L1:
y = 2
L2:
```

- A Estrutura **`while`**: O laço de repetição exige que o fluxo retorne a um ponto anterior do código para reavaliar uma condição.

Python

```
while (x < 10):
    x = x + 1
```

Equivalência em Código Intermediário (TAC):

```
L1:
if x >= 10 goto L2   # Se a condição falhar, sai do loop
x = x + 1
goto L1              # Retorna para reavaliar a condição
L2:
```

I. Questões de Múltipla Escolha

QUESTÃO 1 - O desenvolvimento de compiladores modernos adota uma arquitetura desacoplada onde o *Front-End* e o *Back-End* comunicam-se exclusivamente por meio de uma Representação Intermediária (IR). Considere uma empresa de tecnologia que decida criar um ecossistema de desenvolvimento de software composto por 4 linguagens de programação proprietárias, cujos programas gerados devem ser capazes de rodar em 3 sistemas embarcados com arquiteturas de hardware completamente distintas.

Adotando a estratégia de representação intermediária única baseada em Código de Três Endereços (TAC), quantos módulos de tradução isolados (considerando *Front-Ends* e *Back-Ends*) os engenheiros precisarão programar para garantir o funcionamento de todo o ecossistema?

A) 7 módulos.

B) 12 módulos.

C) 16 módulos.

D) 6 módulos.

E) 64 módulos.

QUESTÃO 2 O Código de Três Endereços (TAC) é uma forma linearizado de representação intermediária que se assemelha às linguagens de montagem, mas mantém independência de registradores específicos de hardware. Na tradução de comandos de alto nível de controle de fluxo, como `while` e `if-then-else`, o compilador utiliza instruções de desvio condicional e incondicional associadas a rótulos (*labels*).

Considere o seguinte trecho de código intermediário:

```
L1:
if a >= b goto L2
a = a + 1
goto L1
L2:
```

A estrutura de alto nível correspondente a esse comportamento lógico é um bloco de:

A) Atribuição simples condicional.

B) Laço de repetição do tipo *while*.

C) Estrutura condicional do tipo *if-then-else* sem aninhamento.

D) Declaração de escopo de função recursiva.

E) Tratamento de exceções com interrupção.

II. Questões Discursivas

QUESTÃO 3 (Discursiva) Explique por que uma expressão matemática complexa como `resultado = (a + b) * (c - d)` não pode ser enviada diretamente para a geração de código final sem antes ser decomposta em instruções de Código de Três Endereços (TAC). Justifique sua resposta abordando as limitações físicas das CPUs tradicionais.

QUESTÃO 4 (Discursiva) A otimização de código é uma das etapas mais importantes que ocorrem na camada intermediária do compilador. Descreva o conceito de "Eliminação de Código Morto" (*Dead Code Elimination*) aplicada sobre representações intermediárias e explique como o compilador pode identificar que uma instrução dentro de um bloco `if` nunca será executada.

Unidade 13: Onde o Código Vive – Ambientes de Execução

13.1 Organização da Memória: Diferença entre Pilha (Stack) e Heap

Quando um programa é carregado pelo Sistema Operacional, a memória RAM alocada para ele é dividida de forma lógica em diferentes segmentos. Para a Engenharia de Computação, compreender a distinção entre a **Pilha (*Stack*)** e o **Heap** é vital para evitar vazamentos de memória (*memory leaks*) e otimizar o desempenho do software.

1. A Pilha (Stack)

A Pilha é uma estrutura de memória extremamente rápida e organizada de forma sequencial, funcionando sob a lógica **LIFO** (*Last In, First Out* — O último a entrar é o primeiro a sair).

- **Como funciona:** Cada vez que uma função é chamada, o compilador cria um "Quadro de Ativação" (*Stack Frame*) no topo da pilha. Esse quadro armazena todas as variáveis locais, os parâmetros recebidos pela função e o endereço de retorno (para onde o programa deve voltar quando a função terminar).

- **Gerenciamento:** É **automático**. Quando a função chega ao fim, seu *Stack Frame* é destruído imediatamente, liberando o espaço de memória.

- **Características:** Possui tamanho limitado (podendo gerar o erro *Stack Overflow* se houver recursão infinita) e os dados guardados nela precisam ter tamanho fixo e conhecido em tempo de compilação.

2. O Heap

O Heap é um grande segmento de memória compartilhada e desorganizada, usado para alocação **dinâmica** de dados em tempo de execução.

- **Como funciona:** Quando o programa precisa criar um objeto, uma lista ou uma estrutura cujo tamanho pode mudar (por exemplo, dados vindos de uma API), ele solicita um espaço no Heap.

- **Gerenciamento:** O sistema operacional busca um bloco livre no Heap que caiba aquele dado, reserva o espaço e devolve um **ponteiro** (um endereço de memória). Esse ponteiro, por ter tamanho fixo, fica guardado na Pilha.

- **Características:** Muito maior que a Pilha, mas o acesso aos dados é mais lento porque exige a resolução de ponteiros.

Python

```
def processar_dados():
    X = 10                  # Guardado na Pilha (Stack)
    lista = [1, 2, 3, 4]    # O objeto lista vai para o Heap; a
variável 'lista' (ponteiro) fica na Pilha.
```

13.2 Coleta de Lixo (Garbage Collection)

Nas linguagens de baixo nível tradicionais, como C e C++, o programador é o responsável absoluto por gerenciar o Heap. Se ele alocar memória com `malloc()` ou `new`, ele deve obrigatoriamente liberá-la com `free()` ou `delete`. Esquecer de liberar gera o *Memory Leak* (o programa consome toda a RAM da máquina até travar).

Para solucionar esse problema e aumentar a segurança do software, linguagens modernas como Java, Python, C# e Go utilizam um mecanismo integrado chamado **Coletor de Lixo (*Garbage Collector* - GC)**. O papel do GC é rastrear o Heap automaticamente e desalocar a memória que não está mais sendo usada.

Algoritmos Comuns de Coleta de Lixo:

1. Contagem de Referências (Usado no CPython)

Cada objeto criado no Heap possui um contador interno que registra quantas variáveis apontam para ele.

- Quando uma variável começa a apontar para o objeto, o contador aumenta (+1).

- Quando a variável sai de escopo ou passa a apontar para outra coisa, o contador diminui (-1).
- Se o contador chegar a **zero**, o objeto torna-se inacessível e a memória é liberada instantaneamente.
- *Problema:* Não consegue resolver nativamente referências cíclicas (objeto A aponta para o objeto B, e B aponta para A).

2. Rastreamento por Varredura (Mark-and-Sweep - Usado na JVM do Java)

O GC interrompe periodicamente a execução do programa principal (evento conhecido como *Stop-the-World*) para realizar duas fases:

1. **Marcação (Mark):** Partindo de raízes conhecidas (como variáveis ativas na Pilha e variáveis globais), o GC navega pelo Heap marcando todos os objetos que são alcançáveis.
2. **Varredura (Sweep):** O GC percorre todo o Heap. Qualquer objeto que não tenha sido marcado na fase anterior é considerado "lixo" e sua memória é liberada.

I. Questões de Múltipla Escolha

QUESTÃO 1 - Durante a execução de um sistema de software desenvolvido em uma linguagem de programação com gerenciamento de memória em tempo de execução (*runtime*), ocorrem chamadas consecutivas de funções de cálculo estatístico. Cada função declara variáveis locais de controle numérico e instancia grandes matrizes dinâmicas com dados lidos de um banco de dados externo.

Considerando os conceitos de organização de memória, os parâmetros de controle e as matrizes dinâmicas serão armazenados, respectivamente, na:

A) Pilha (*Stack*) e na Pilha (*Stack*), devido à necessidade de desalocação rápida.

B) Pilha (*Stack*) e no Heap, visto que o tamanho das matrizes é determinado dinamicamente em tempo de execução.

C) Memória Cache e no Heap, para otimizar os desvios de barramento do processador.

D) Tabela de Símbolos e no Heap, permitindo o rastreamento semântico.

E) Heap e na Pilha (*Stack*), obedecendo à estrutura de persistência volátil.

QUESTÃO 2 - O mecanismo de Coleta de Lixo por Rastreamento (*Mark-and-Sweep*) é amplamente utilizado em máquinas virtuais para evitar vazamentos de memória sem transferir o ônus da desalocação ao desenvolvedor. Contudo, esse algoritmo impõe um custo de processamento ao sistema operacional devido à necessidade de varredura de grafos de objetos.

A principal desvantagem operacional imediata ao se disparar a fase de marcação (*Mark*) desse algoritmo em sistemas de tempo real é:

A) A corrupção dos ponteiros locais armazenados na Pilha sintática.

B) A necessidade de reinicializar as variáveis globais do sistema.

C) A pausa temporária na execução das *threads* da aplicação (*Stop-the-World*).

D) O aumento do consumo de energia devido ao uso exclusivo da memória estática.

E) A transformação automática de linguagens interpretadas em código compilado nativo.

II. Questões Discursivas

QUESTÃO 3 (Discursiva) Explique o fenômeno do *Stack Overflow* (Estouro de Pilha). Em sua resposta, descreva como a chamada sucessiva de uma função recursiva sem uma condição de parada adequada afeta a estrutura de *Stack Frames* na memória do programa.

QUESTÃO 4 (Discursiva) Linguagens como Rust adotam uma abordagem inovadora chamada "Sistema de Posse" (*Ownership*) para gerenciar a memória no Heap em tempo de compilação, abrindo mão tanto do gerenciamento manual (como em C) quanto do uso de um *Garbage Collector* ativo (como em Java).

Discorra sobre as vantagens de desempenho e consumo de hardware que uma linguagem ganha ao eliminar a necessidade de um *Garbage Collector* em seu ambiente de execução.

Unidade 14: Geração e Otimização de Código

14.1 Linguagem Objeto (Assembly/Binário): O Destino Final

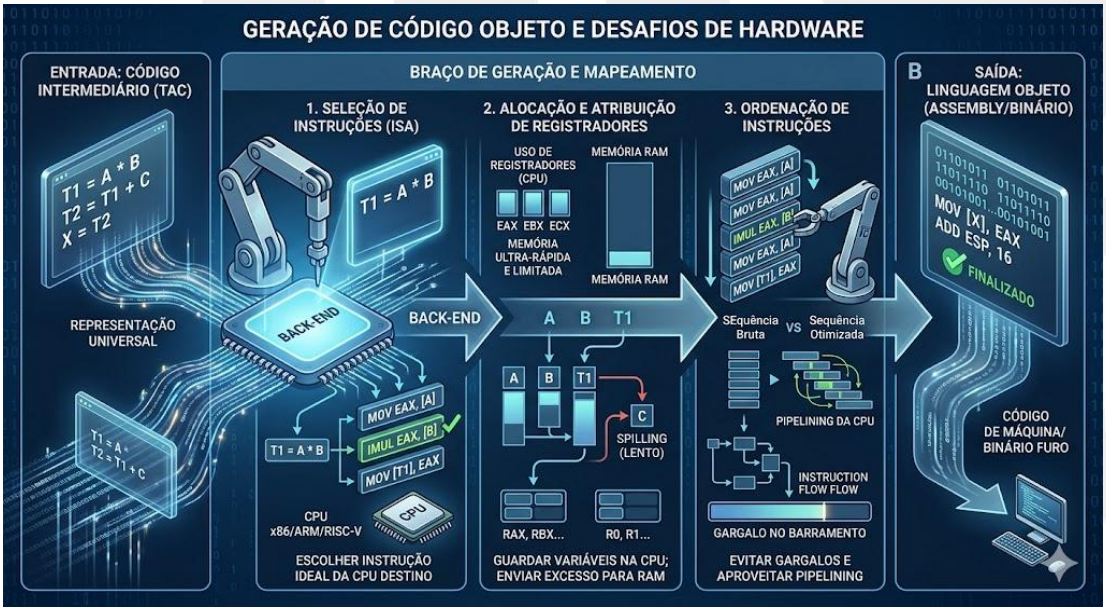
A fase de **Geração de Código** é o ato final do *Back-End* do compilador. Ela recebe a representação intermediária (como o Código de Três Endereços - TAC, que estudamos na Unidade 12) e a mapeia para a **Linguagem Objeto**, que pode ser o código de montagem (*Assembly*) de uma arquitetura específica ou o código de máquina binário puro (**0**s e **1**s).

Para realizar essa tradução com máxima eficiência, o gerador de código precisa resolver três problemas complexos de engenharia de hardware:

1. **Seleção de Instruções:** Escolher quais instruções da CPU destino são as mais adequadas e rápidas para realizar uma operação matemática ou lógica da linguagem intermediária.

2. **Alocação e Atribuição de Registradores:** Decidir quais variáveis do programa ficarão guardadas nos registradores da CPU (que são memórias ultrarrápidas dentro do próprio processador, mas de quantidade extremamente limitada, como **RAX**, **RBX** em x86, ou **R0**, **R1** em ARM). As variáveis que não couberem nos registradores precisam ser enviadas para a memória RAM (processo lento conhecido como *Spilling*).

3. **Ordenação de Instruções:** Determinar a sequência exata de execução para evitar gargalos no barramento físico e aproveitar recursos como o *pipelining* do processador.



Exemplo Prático de Mapeamento:

Considere a instrução intermediária de atribuição: `x = y + 5`

Mapeamento típico para o **Assembly x86_64**:

Snippet de código

```
MOV EAX, [y] ; Carrega o valor da variável 'y' da RAM para o registrador EAX
ADD EAX, 5   ; Soma o valor imediato 5 ao conteúdo do registrador EAX
MOV [x], EAX ; Copia o resultado de EAX de volta para o endereço da variável
'x' na RAM
```

14.2 Otimização: O Programa Mais Rápido Sem Mudar o Código

A **Otimização de Código** pode acontecer tanto na camada intermediária quanto na geração final. Seu objetivo principal é diminuir o tempo de execução do programa (*tempo*) e/ou reduzir o espaço que ele ocupa na memória (*espaço*), mantendo o comportamento original exatamente idêntico.

Aqui estão as técnicas clássicas de otimização aplicadas de forma transparente pelos compiladores modernos:

I. Dobradura e Propagação de Constantes (Constant Folding and Propagation)

Se o programador escreve operações envolvendo valores fixos, o compilador calcula o resultado em tempo de compilação para evitar que a CPU gaste ciclos calculando isso em tempo de execução.

- *Código original:*

Python

```
taxa = 0.15
tempo_segundos = 60 * 60 * 24
```

- *Código otimizado:*

Python

```
taxa = 0.15
tempo_segundos = 86400  # O cálculo (60*60*24) foi resolvido de forma
estática
```

II. Eliminação de Subexpressões Comuns (Common Subexpression Elimination - CSE)

Se uma mesma expressão matemática é calculada mais de uma vez em sequência, o compilador calcula uma vez, guarda o resultado em um registrador temporário e o reutiliza.

- *Código original:*

Python

```
x = (a + b) * c
y = d + (a + b)
```

- *Código otimizado:*

Python

```
t1 = a + b
x = t1 * c
y = d + t1
```

II. Otimizações de Laço: Redução de Força e Movimento de Código

- Movimento de Código (*Loop-Invariant Code Motion*): Se uma expressão dentro de um `while` ou `for` resulta sempre no mesmo valor em todas as iterações, o compilador a move para *fora* do laço.

- Redução de Força (*Strength Reduction*): Substitui operações matematicamente pesadas por equivalentes mais leves. Multiplicações dentro de loops podem ser substituídas por somas consecutivas, ou multiplicações por potências de 2 podem ser trocadas por deslocamentos de bits (*Bitwise Shift*), que rodam instantaneamente no hardware.

14.3 Visão Geral do Ciclo de Vida: Do Teclado ao Processador

Para consolidar o aprendizado de toda a disciplina, podemos revisar o fluxo completo que transforma um texto digitado pelo desenvolvedor no arquivo binário que roda na CPU:

1. **Código-Fonte:** O programador digita o texto puro no arquivo (ex: `resultado = x + 10;`).

2. **Análise Léxica (Unidade 5):** Transforma os caracteres brutos em uma sequência linear de **Tokens** (`[ID, resultado]`, `[OP_ATRIB, =]`, `[ID, x]`, `[OP_SOMA, +]`, `[NUM, 10]`, `[PONT, ;]`).

3. **Análise Sintática (Unidades 8, 9, 10):** Agrupa os tokens em uma estrutura hierárquica baseada nas regras da gramática formal, gerando a **Árvore de Sintaxe Abstrata (AST)**.

4. **Análise Semântica (Unidade 11):** Consulta a **Tabela de Símbolos** para validar o significado das instruções (tipos, escopos e declarações).

5. **Geração de Código Intermediário (Unidade 12):** Traduz a árvore em uma linguagem plana universal e linear, como o Código de Três Endereços (TAC).

6. **Otimização:** Aplica as transformações matemáticas para tornar o fluxo do TAC o mais enxuto possível.

7. **Geração de Código Objeto (Unidade 14):** Aloca registradores, escolhe as instruções da arquitetura real (como x86 ou ARM) e cospe as instruções físicas binárias.

8. **Ambiente de Execução (Unidade 13):** O programa roda na RAM, gerenciando dinamicamente seus espaços através da **Pilha** e do **Heap**.

I. Questões de Múltipla Escolha

QUESTÃO 1 - No *Back-End* de um compilador, a fase de geração de código objeto lida diretamente com as restrições físicas da arquitetura do processador de destino. Uma das tarefas mais críticas dessa fase consiste em decidir quais variáveis lógicas do programa de alto nível serão mapeadas para os registradores internos da CPU e quais precisarão ser armazenadas na memória RAM principal.

Essa tarefa de otimização de recursos físicos é denominada:

A) Seleção e Ordenação de Instruções Semânticas.

B) Alocação e Atribuição de Registradores.

C) Linearização de Código de Três Endereços.

D) Dobradura de Constantes Estáticas.

E) Verificação de Tipos em Tempo de Execução.

QUESTÃO 2 - Considere que um otimizador de código analise o seguinte laço de repetição escrito em uma linguagem de alto nível:

Python

```
for i in range(1000):
    vetor[i] = i * (largura_tela / 2)
```

Ao aplicar a técnica de **Movimento de Código (*Loop-Invariant Code Motion*)**, qual ação o compilador executará de forma transparente para aumentar o desempenho do programa?

A) Removerá o laço `for`, substituindo-o por 1000 linhas repetidas de atribuição.

B) Calculará o valor da subexpressão `(largura_tela / 2)` uma única vez *antes* do início do laço, armazenando o resultado em uma variável temporária.

C) Alterará o tipo da variável `i` de inteiro para flutuante no Heap.

D) Substituirá a operação de multiplicação por um deslocamento de bits à esquerda.

E) Ignorará a execução do laço se o vetor não tiver sido alocado previamente na Pilha.

II. Questões Discursivas

QUESTÃO 3 - Imagine que você possui um código intermediário linear contendo a instrução `y = x * 8`. Explique como o gerador de código objeto do compilador pode aplicar a técnica de **Redução de Força (*Strength Reduction*)** ao traduzir essa instrução para o Assembly de um microcontrolador, justificando o ganho de desempenho sob a ótica do hardware.

QUESTÃO 4 - A arquitetura modular de compiladores modernos que utiliza uma representação intermediária independente de máquina permite que técnicas de otimização sejam aplicadas em diferentes momentos. Diferencie os objetivos e o escopo das otimizações realizadas na camada **independente de máquina** daquelas realizadas na camada **dependente de máquina** (geração final).

Revisão Para Prova

Bloco 1: Front-End, Linguagens Formais e Análise Léxica (Unidades 1 a 6)

Questão 1 - Na classificação da Hierarquia de Chomsky, as gramáticas que definem a estrutura da maioria das linguagens de programação e permitem o aninhamento recursivo de estruturas lógicas (como blocos condicionais e laços) pertencem à qual categoria?

A) Tipo 0 – Linguagens Irrestritas.

B) Tipo 1 – Linguagens Sensíveis ao Contexto.

C) Tipo 2 – Linguagens Livres de Contexto.

D) Tipo 3 – Linguagens Regulares.

E) Tipo 4 – Linguagens Naturais.

Questão 2 - Uma gramática formal é matematicamente definida pela quádrupla G = (V, Σ, P, S). Qual elemento desse conjunto representa os símbolos reais e finais que compõem o código-fonte escrito pelo programador?

A) Símbolo Inicial (S).

B) Variáveis não-terminais (V).

C) Regras de Produção (P).

D) Alfabeto de Terminais (Σ).

E) Autômatos de Controle (A).

Questão 3 - Qual é a principal consequência negativa de se projetar uma gramática ambígua para uma linguagem de programação?

A) O aumento excessivo do tamanho do arquivo binário final.

B) A possibilidade de gerar duas ou mais árvores de derivação diferentes para uma mesma sentença, gerando indeterminação para o compilador.

C) A impossibilidade de declarar variáveis globais no escopo do programa.

D) A obrigatoriedade de utilizar o algoritmo de retrocesso (backtracking) na análise léxica.

E) A rejeição imediata de todas as expressões matemáticas que utilizam parênteses.

Questão 4 - A arquitetura de um compilador clássico é dividida entre Front-End e Back-End. Qual é a principal vantagem estratégica dessa divisão modular na engenharia de software?

A) Permitir a execução do compilador em sistemas operacionais sem memória RAM.

B) Eliminar a necessidade de validar o significado semântico das variáveis.

C) Possibilitar o reaproveitamento do mesmo Back-End para traduzir diferentes linguagens de origem para uma mesma arquitetura de hardware.

D) Isolar completamente o código de máquina de qualquer representação intermediária.

E) Forçar o código-fonte a ser interpretado linha por linha em tempo de execução.

Questão 5 - Como o Kernel do Sistema Operacional interage com os programas em execução no espaço de usuário para gerenciar recursos de hardware?

A) Por meio de Expressões Regulares de controle.

B) Através de chamadas de sistema (System Calls).

C) Utilizando tabelas de conversão sintática exclusivas do Lark.

D) Por meio de compilação híbrida em tempo de execução (JIT).

E) Impondo a verificação estática de tipos no Heap.

Questão 6 - Diferente de um compilador puro, o que caracteriza a operação de um Interpretador clássico?

A) Ele traduz o código inteiro para binário antes de iniciar a execução do software.

B) Ele dispensa as fases de análise léxica e tabela de símbolos.

C) Ele analisa e executa as instruções do código-fonte de forma sequencial e direta, sem gerar um arquivo executável independente.

D) Ele armazena todas as variáveis exclusivamente no segmento de dados estáticos do Kernel.

E) Ele restringe a sintaxe da linguagem apenas a estruturas de Tipo 3 de Chomsky.

Questão 7 - Linguagens modernas como Java e C# utilizam um modelo híbrido de tradução. O que o compilador dessas linguagens gera para garantir a portabilidade entre diferentes plataformas?

A) Código de máquina binário puro para o processador x86_64.

B) Bytecode intermediário para ser executado por uma máquina virtual.

C) Tabelas de transição de estados para autômatos não-determinísticos.

D) Arquivos de texto com macros em notação EBNF.

E) Scripts planos contendo apenas árvores de derivação concreta.

Questão 8 - No contexto da análise léxica, qual é a definição técnica correta de um Lexema?

A) É a categoria abstrata à qual pertence uma palavra reservada.

B) É a sequência real de caracteres extraída do código-fonte que corresponde a um padrão de token.

C) É o índice de memória alocado na tabela de símbolos para uma função.

D) É a regra de produção utilizada para expandir o símbolo inicial.

E) É o erro gerado quando um operador matemático é colocado na posição errada.

Questão 9 - Se um analisador léxico baseado em Expressões Regulares utiliza o padrão [0-9]+, o metacaractere + indica que o dígito numérico deve ocorrer:

A) Zero ou uma única vez (opcional).

B) Exatamente dez vezes consecutivas.

C) Uma ou mais vezes consecutivas.

D) Apenas se houver um operador de adição logo em seguida.

E) Exclusivamente no início da linha de código.

Questão 10 - A regra do Maximal Munch (ou Maior Casamento) dita o comportamento do analisador léxico quando há sobreposição de padrões. Segundo essa regra, o scanner deve:

A) Consumir a menor sequência possível de caracteres que valide um token.

B) Interromper a leitura assim que encontrar a primeira letra maiúscula.

C) Consumir a maior sequência possível de caracteres na entrada que corresponda a um token válido.

D) Ignorar os operadores aritméticos em favor de identificadores textuais.

E) Rejeitar lexemas que possuam mais de três caracteres de comprimento.

Bloco 2: Automação e Análise Sintática (Unidades 7 a 10)

Questão 11 - Por que ferramentas de automação como Lex, Yacc, AntLR e a biblioteca Lark são amplamente preferidas no desenvolvimento de compiladores profissionais?

A) Porque elas substituem a necessidade de o hardware executar o código binário.

B) Porque eliminam a fase de geração de código objeto, gerando o executável diretamente da sintaxe.

C) Porque automatizam a criação de analisadores complexos a partir de especificações gramaticais formais, minimizando erros manuais.

D) Porque convertem linguagens de alto nível diretamente em linguagens naturais.

E) Porque limitam o uso de memória RAM apenas ao segmento da Pilha.

Questão 12 - O algoritmo LL(*) adotado por ferramentas como o AntLR apresenta qual inovação em relação aos analisadores LL(1) tradicionais?

A) Ele analisa o código de trás para frente, partindo do final do arquivo.

B) Ele possui um lookahead dinâmico, sendo capaz de olhar quantos tokens forem necessários à frente na fita para decidir a regra correta.

C) Ele dispensa completamente o uso de símbolos terminais na gramática.

D) Ele restringe a análise apenas a gramáticas regulares (Tipo 3).

E) Ele exige a conversão de toda a gramática para tabelas de ação e desvio estáticas.

Questão 13 - A biblioteca Lark, implementada em Python, adota nativamente qual notação padrão para que os desenvolvedores definam as regras de suas gramáticas?

A) Notação JSON estruturada.

B) Esquemas XML extensíveis.

C) Notação EBNF (Extended Backus-Naur Form).

D) Consultas relacionais SQL.

E) Diagramas de transição de texto plano.

Questão 14 - Qual tipo de máquina abstrata é teoricamente utilizado para reconhecer e validar as linguagens regulares (Tipo 3) na fase de análise léxica?

A) Autômatos de Pilha Não-Determinísticos.

B) Máquinas de Turing Universais.

C) Autômatos Finitos (Determinísticos ou Não-Determinísticos).

D) Sistemas de Reduções Iterativas LR.

E) Grafos de Árvores de Sintaxe Abstrata.

Questão 15 - A análise sintática descendente (Top-Down) constrói a árvore de derivação realizando qual movimento lógico estrutural?

A) Parte dos tokens individuais (folhas) e realiza reduções sucessivas até alcançar a raiz.

B) Inicia-se na raiz (símbolo inicial da gramática) e expande as regras de cima para baixo até corresponder aos tokens da fita.

C) Varre o código aleatoriamente utilizando funções de hash.

D) Constrói os nós filhos antes de instanciar os nós pais na memória.

E) Executa a verificação semântica antes de ler a fita de tokens.

Questão 16 - O que indica um erro sintático durante a execução de um parser estruturado?

A) Que o programador tentou dividir um número por zero.

B) Que uma variável foi utilizada sem ter sido previamente declarada.

C) Que a sequência de tokens lida na entrada quebrou a ordem e a estrutura esperadas pelas regras gramaticais da linguagem.

D) Que o arquivo de código-fonte possui caracteres inválidos no alfabeto.

E) Que houve estouro de capacidade física no segmento de memória Heap.

Questão 17 - Na análise sintática ascendente (Bottom-Up), o que representa o conceito técnico de uma Alça (Handle)?

A) O próximo token que está esperando na fita de entrada para ser lido (lookahead).

B) Uma subcadeia de símbolos no topo da pilha que coincide perfeitamente com o lado direito de uma produção gramatical válida para redução.

C) O ponteiro de memória que conecta a Tabela de Símbolos ao gerador de código.

D) Um erro de sintaxe associado a um conflito irrecuperável de deslocamento.

E) O nó raiz de uma Árvore de Sintaxe Abstrata completa.

Questão 18 - Durante a operação de um analisador sintático do tipo Shift-Reduce, a ação de Reduce (Reduzir) consiste em:

A) Ler o próximo token da fita de entrada e empilhá-lo na pilha de símbolos.

B) Diminuir o tamanho das variáveis inteiras para economizar espaço no Heap.

C) Retirar do topo da pilha a sequência de símbolos que forma o corpo de uma regra e substituí-la pelo seu não-terminal correspondente.

D) Descartar os tokens de comentários e espaços em branco da fita.

E) Encerrar a compilação emitindo um aviso de sucesso sintático.

Questão 19 - O analisador sintático ascendente entra em um estado de "Conflito Shift-Reduce" quando:

A) A pilha de símbolos fica totalmente vazia antes do fim do arquivo.

B) Ele não consegue determinar matematicamente se deve empilhar o próximo token ou se deve reduzir os símbolos já presentes no topo da pilha.

C) Duas regras de produção diferentes possuem exatamente o mesmo não-terminal do lado esquerdo.

D) O analisador léxico entrega um token com atributo semântico corrompido.

E) Ele atinge o símbolo inicial da gramática sem consumir toda a entrada.

Questão 20 - Diferente da árvore de derivação concreta, a Árvore de Sintaxe Abstrata (AST) caracteriza-se por:

A) Conter todos os delimitadores textuais, parênteses e pontos e vírgulas lidos no código.

B) Ser gerada exclusivamente por interpretadores híbridos baseados em tabelas de ação.

C) Omitir detalhes e ruídos puramente gramaticais, focando estritamente na essência e na estrutura lógica dos operadores e operandos.

D) Armazenar os endereços físicos de registradores como RAX e RBX.

E) Ser construída de cima para baixo ignorando as regras de produção.

Bloco 3: Análise Semântica e Código Intermediário (Unidades 11 e 12)

Questão 21 - Qual é o objetivo primordial da fase de Análise Semântica em um compilador?

A) Validar se os caracteres lidos pertencem ao alfabeto da linguagem.

B) Garantir que a ordem dos tokens obedeça à estrutura hierárquica da gramática Livre de Contexto.

C) Verificar a coerência lógica e o significado das instruções do programa, inspecionando regras de escopo, unicidade e compatibilidade de tipos.

D) Traduzir as expressões matemáticas lineares diretamente para código binário purificado.

E) Gerenciar os limites físicos de tamanho das matrizes alocadas na Pilha.

Questão 22 - A Tabela de Símbolos é uma estrutura de dados central no funcionamento do compilador. Quais informações básicas ela deve reter sobre os identificadores mapeados no programa?

A) Apenas o nome do lexema e a linha onde ocorreu o erro léxico.

B) O nome, a categoria (variável, função, parâmetro), o tipo de dado e o escopo associado.

C) As regras de produção EBNF convertidas em formato binário.

D) Os gráficos de fluxo de controle linearizados para o laço while.

E) O histórico de modificações do código-fonte feitas pelo desenvolvedor.

Questão 23 - Se um compilador analisa a instrução int valor = "fmu"; e aponta uma falha, estamos diante de um erro de qual natureza?

A) Erro Léxico, porque a palavra "fmu" contém caracteres proibidos.

B) Erro Sintático, pois a ordem dos elementos da atribuição está invertida.

C) Erro Semântico, devido à incompatibilidade de tipos (Type Checking) entre um inteiro e uma cadeia de caracteres.

D) Erro de Código Intermediário, decorrente de uma falha de desvio condicional.

E) Erro de Otimização de Código por falta de dobradura de constantes.

Questão 24 - A introdução de um Código Intermediário universal (como o Código de Três Endereços - TAC) reduz o esforço de engenharia para criar compiladores para M linguagens voltadas a N arquiteturas de hardware de:

A) M X N para M + N.

B) M^N para log(M).

C) M + N para uma constante fixa de 2 módulos.

D) Uma escala exponencial para uma escala linear de Tipo 0.

E) M \X N para zero módulos adaptáveis.

Questão 25 - Uma instrução padrão em Código de Três Endereços (TAC) possui qual restrição estrutural?

A) Deve conter no mínimo cinco operandos e três operadores lógicos estruturados.

B) Pode processar apenas variáveis declaradas no escopo global do Kernel.

C) Contém, no máximo, um único operador e três operandos (dois argumentos e um destino).

D) Exige a tradução prévia de todas as constantes textuais em ponteiros do Heap.

E) Deve obrigatoriamente invocar uma chamada de sistema (System Call).

Bloco 4: Ambientes de Execução, Geração e Otimização de Código (Unidades 13 e 14)

Questão 26 - No ambiente de execução (Runtime), as variáveis locais de uma função e seus parâmetros de controle são alocados em qual segmento de memória?

A) No Heap, por meio de alocação dinâmica via ponteiros flexíveis.

B) Na Pilha (Stack), organizada sob a lógica LIFO e gerenciada de forma automática.

C) Na Tabela de Símbolos estática do Back-End.

D) No segmento de Código Morto otimizado.

E) Exclusivamente nos registradores de controle do Kernel do sistema operacional.

Questão 27 - O erro crítico de execução conhecido como Stack Overflow (Estouro de Pilha) é comumente provocado por:

A) Alocação excessiva de grandes objetos dinâmicos e matrizes flexíveis no Heap.

B) Falha nos contadores do algoritmo de Coleta de Lixo por contagem de referências.

C) Chamadas recursivas sucessivas de funções sem uma condição de parada adequada, esgotando o limite físico reservado para os Stack Frames.

D) Uso incorreto da técnica de otimização por dobradura de constantes aritméticas.

E) Tentativa de ler um token que não pertence ao alfabeto de terminais da gramática.

Questão 28 - Como opera o algoritmo de Coleta de Lixo por Rastreamento do tipo Mark-and-Sweep (Marcação e Varredura)?

A) Ele destrói as variáveis da Pilha sintática assim que a função chega ao seu fim de bloco.

B) Ele altera o código-fonte em tempo de compilação substituindo loops por instruções planas TAC.

C) Ele mapeia os objetos alcançáveis no Heap a partir de raízes ativas e, em seguida, libera os blocos de memória dos objetos que não receberam marcação.

D) Ele incrementa um contador numérico individual toda vez que um ponteiro muda de endereço.

E) Ele força o processador a realizar desvios condicionais invertidos para limpar a memória RAM.

Questão 29 - Qual é o objetivo técnico da aplicação da técnica de otimização de código chamada Loop-Invariant Code Motion (Movimento de Código em Laços)?

A) Mover expressões cujo resultado é constante e independente das iterações para fora do laço, evitando cálculos redundantes a cada repetição.

B) Substituir comandos de repetição for por blocos de decisão aninhados if-else.

C) Forçar a alocação de todas as variáveis do laço dentro da memória estática de cache.

D) Multiplicar o número de iterações do loop para preencher o pipeline da CPU de destino.

E) Eliminar as instruções que alteram o valor do indexador do laço de repetição.

Questão 30 - A técnica de otimização por Redução de Força (Strength Reduction) atua sobre o código gerado realizando qual substituição estratégica no hardware?

A) Substitui variáveis locais por variáveis globais de escopo amplo.

B) Troca instruções matematicamente pesadas e lentas por equivalentes físicos mais leves e rápidos (como trocar uma multiplicação por potência de 2 por um deslocamento de bits).

C) Reduz a quantidade de nós filhos presentes na Árvore de Sintaxe Abstrata (AST).

D) Elimina os blocos condicionais que nunca serão executados pelo fluxo de controle.

E) Diminui o tamanho físico dos chips de memória RAM alocados pelo Kernel.

Bibliografia Básica

AHO, Alfred V., LAM, Mônica S., SETHI, Ravi, ULLMAN, Jeffrey D., Compiladores: Princípios, Técnicas e Ferramentas [recurso eletrônico, Biblioteca Virtual]. São Paulo: Pearson Addison-Wesley, 2008.

MENEZES, Paulo F. B., Linguagens Formais e Autômatos [recurso eletrônico, Minha Biblioteca]. Porto Alegre, Grupo A, 2011.

DIVERIO, Tiaraju A., Teoria da Computação: Máquinas universais e computabilidade [recurso eletrônico, Minha Biblioteca]. Porto Alegre, Grupo A, 2011.

Bibliografia Complementar

DEITEL, P. J., C: como programar [recurso eletrônico, Biblioteca Virtual]. São Paulo: Pearson Prentice Hall, 2011.

SIPSER, Michael., Introdução à teoria da computação [recurso eletrônico, Minha Biblioteca]. São Paulo: Cengage Learning, 2007.

ASCENCIO, Ana F. Gomes, Estruturas de Dados: algoritmos, análise da complexidade e implementações em Java e C/C++ [recurso eletrônico, Biblioteca Virtual]. São Paulo: Pearson Prentice Hall, 2010.

DEITEL, Paul., Java: como programar [recurso eletrônico, Biblioteca Virtual]. São Paulo: Pearson Education do Brasil, 2017.

TOSCANI, Laira Vieira., Complexidade de Algoritmos [recurso eletrônico, Minha Biblioteca]. Porto Alegre, Bookman, 2012. 3a ed.

Periódicos Científicos

Advances in Software Engineering - ISSN 1687-8655
Computación y Sistemas - ISSN 1405-5546
Computational & Applied Mathematics - ISSN 1807-0302
Journal of the Brazilian Computer Society - ISSN 0104-6500