UNIP
UNIVERSIDADE PAULISTA
Interativa

Unidade I

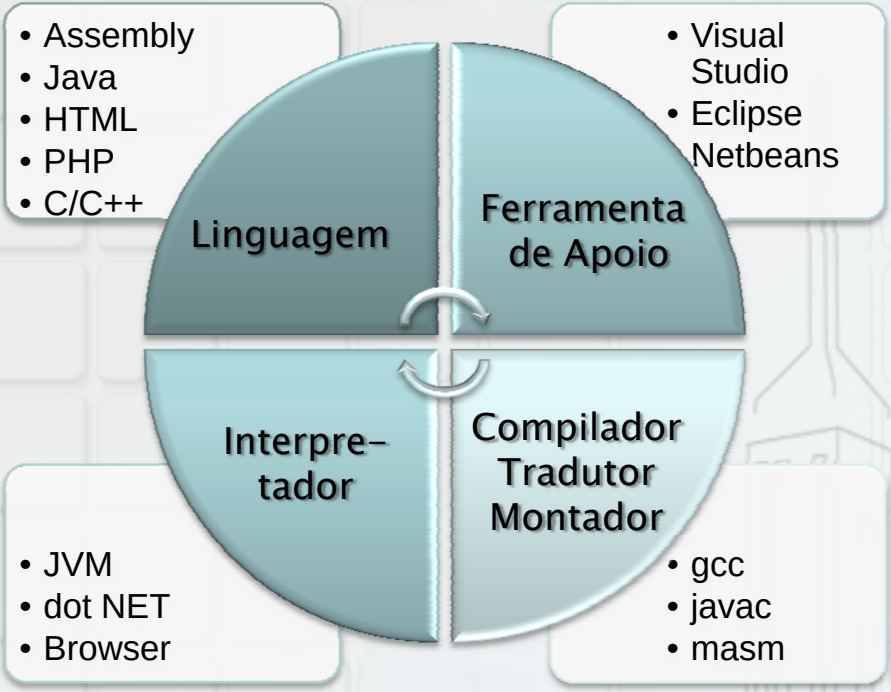
COMPILADORES E COMPUTABILIDADE

Prof. Leandro Fernandes

Roteiro

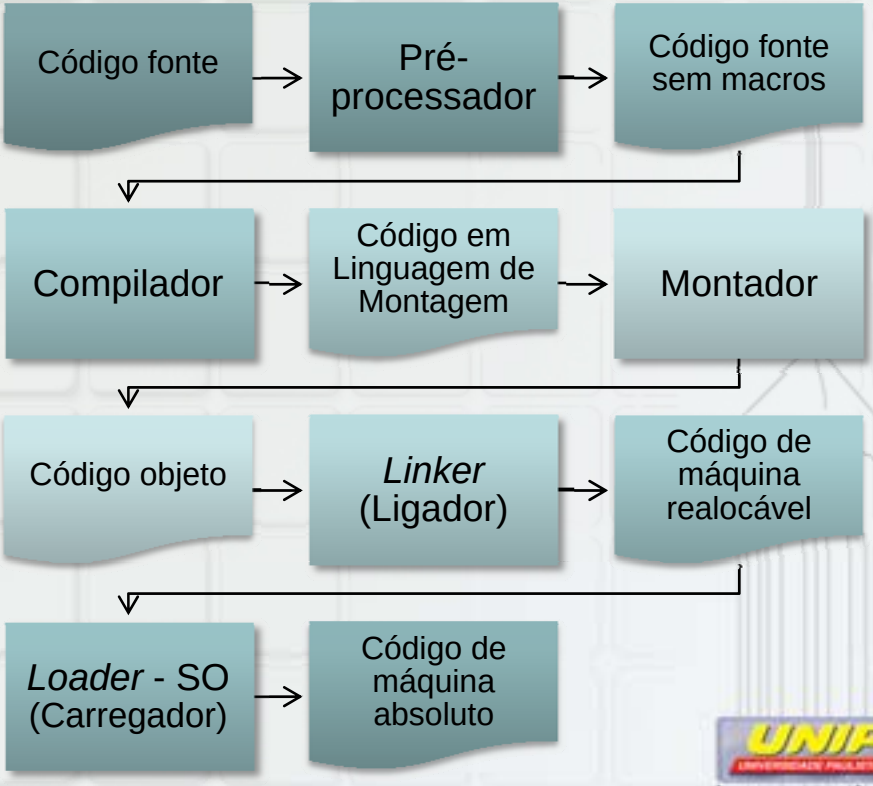
- Contextualização e conceitos básicos
  - O que acontece desde a codificação até a execução de um programa?
- O modelo de análise e síntese
- Análise léxica, a 1ª etapa!
  - O emprego das gramáticas regulares e autômatos finitos
- O processo de análise sintática.
  - Tipos de analisadores e recuperação de erros
  - Analisadores descendentes.
  - Descendentes recursivos e LL(1)

Linguagens, ferramentas e tradutores

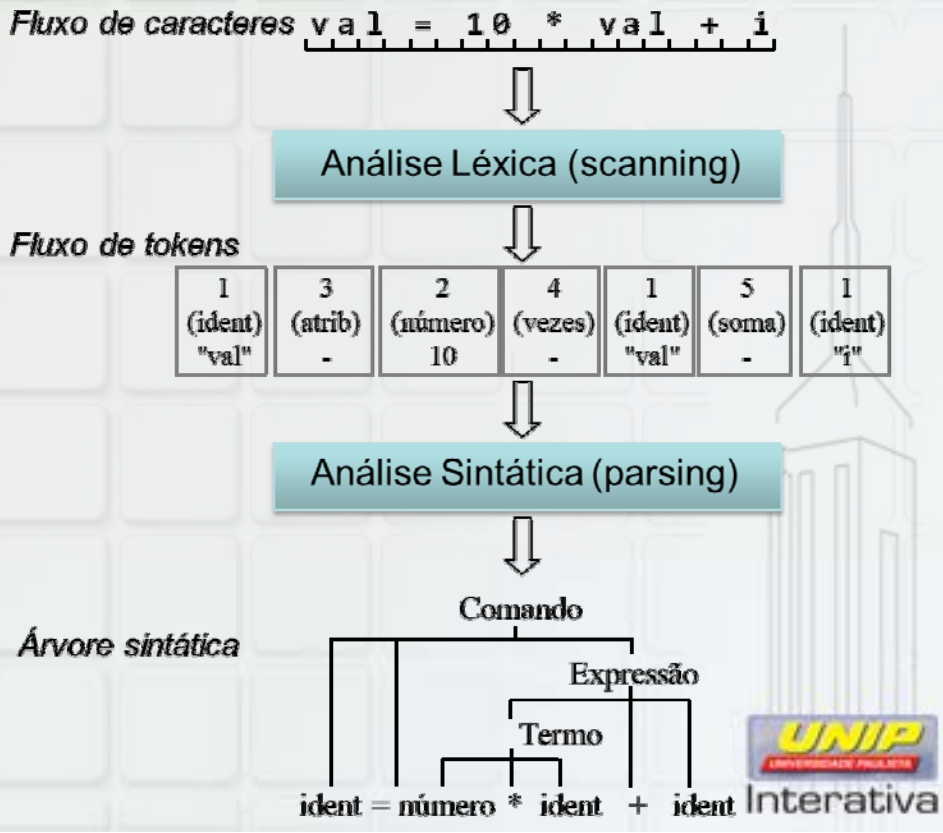


UNIP
Interativa

Da codificação a execução

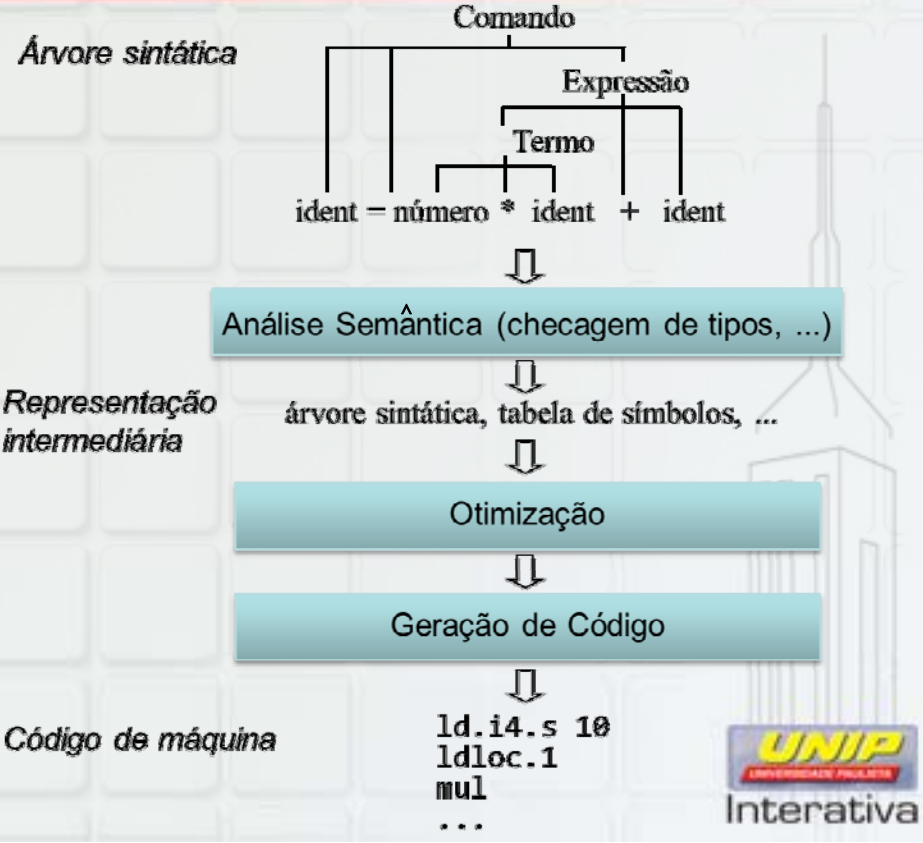


Estrutura dinâmica de um compilador



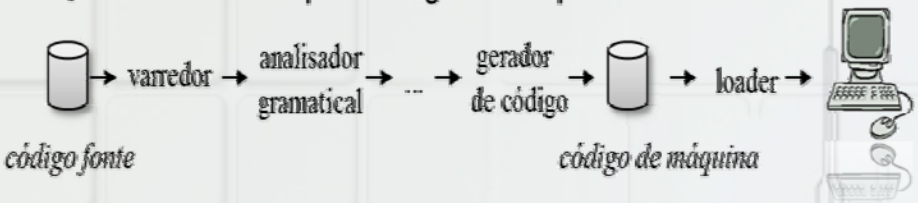
UNIP Interativa

Estrutura dinâmica de um compilador (cont.)

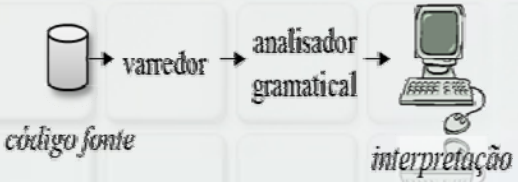


Compiladores vs. interpretadores

Compilador traduz para código de máquina



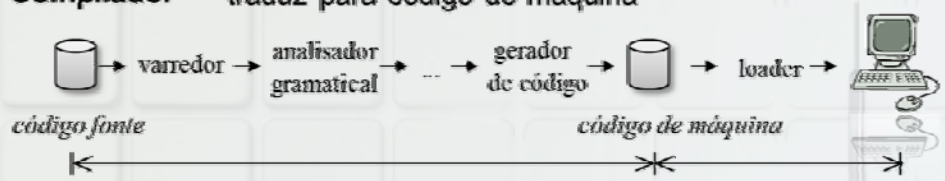
Interpretador executa o código fonte "diretamente"



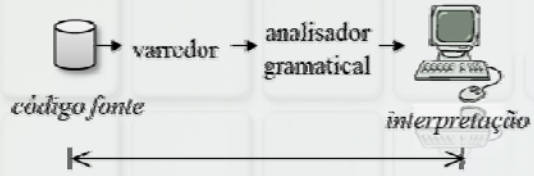
- comandos em um laço são lidos e analisados gramaticalmente a cada iteração

Compiladores vs. interpretadores

Compilador traduz para código de máquina

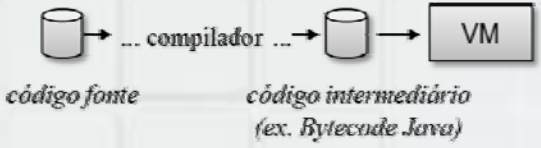


Interpretador executa o código fonte "diretamente"



- comandos em um laço são lidos e analisados gramaticalmente a cada iteração

Variação: interpretação de um código intermediário

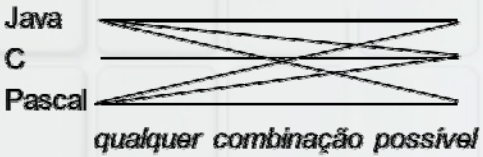


- código fonte é traduzido para o código de uma *máquina virtual*
- A máquina interpreta o código simulando a máquina real

UNIP Interativa

Compiladores de duas passagens (modelo de análise e síntese)

dependente da linguagem **dependente da máquina**



Vantagens

- melhor portabilidade
- possibilidade de muitas combinações entre front-ends e back-ends
- otimizações são mais fáceis na representação intermediária do que no código fonte

Desvantagens

- lenta
- necessita mais memória

UNIP Interativa

Interatividade

Qual tipo de *software* tradutor deve ser utilizado para programas em geral, quando a velocidade de execução é uma exigência de alta prioridade?

a) **Compiladores.**

b) **Interpretadores.**

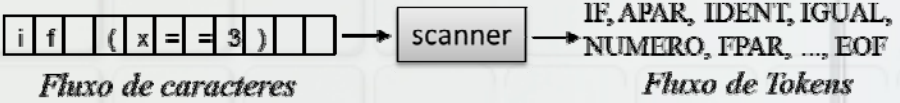
c) **Tradutores híbridos.**

d) **Macroprocessadores.**

e) **Interpretadores de macroinstruções.**

Analise léxica (tokenização ou *scanning*)

- **Produzir símbolos terminais:**



- **Ignorar e descartar símbolos irrelevantes:**
  - **espaços em branco;**
  - **caracteres de tabulação;**
  - **caracteres de controle (CR e LF);**
  - **Comentários.**
- ***Tokens* possuem uma estrutura sintática:**
  - ***identif*:= letra { letra | dígito }**
  - **número:= dígito { dígito }**
  - **if := “i“ ”f”**
  - **igual := "=““=“ ...**

UNIP Interativa

Por que o analisador léxico não é uma parte do analisador sintático?

- **Isso deixaria o analisador sintático mais complicado de ser construído.**
- **Dificulta a distinção entre palavras reservadas e identificadores.**

```
Statement   =  ident "=" Expr ";"
            |  "if" "(" Expr ")" ... .
```

- **Precisaria ser reescrito na forma:**

```
Statement   =  "i" (   "f" "(" Expr ")" ...
                   |   notF {letter | digit} "=" Expr ";"
                   )
            |  notI {letter | digit} "=" Expr ";".
```

Por que o analisador léxico não é uma parte do analisador sintático?

- O *scanning* deve eliminar brancos, tabulações, fins de linha e comentários.
- Esses caracteres podem ocorrer em qualquer lugar do código, levando a gramáticas muito complexas.

```
Statement    =  "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

- *Tokens* podem ser descritos por linguagens regulares:
  - mais simples e mais eficientes que as gramáticas livres de contexto.

Usando uma gramática regular para representar lexemas

- **Um gramática é dita regular se suas produções são na forma:**

A = a	a, b ∈ Símbolos Terminais
A = b B	A, B ∈ Símbolos Não terminais

- **Ex.: gramática de nomes.**

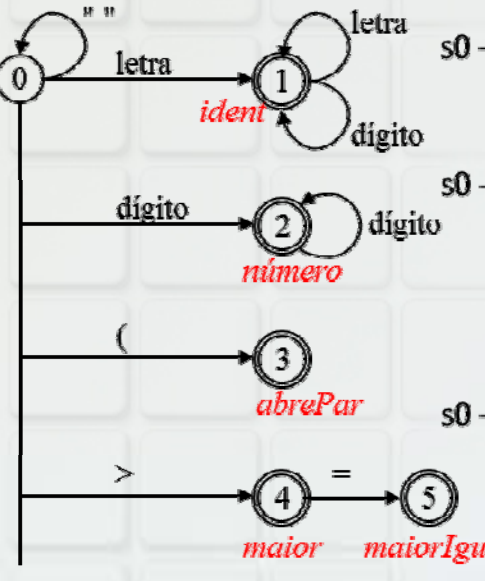
```
Identif = letra
        | letra Restante
Restante = letra
         | dígito
         | letra Restante
         | dígito Restante
```

- **Definição alternativa:**

```
Ident = letra { letra | dígito }
```

O *scanner* como um AFD (Autômato Finito Determinístico)

- **Exemplo para a entrada: max >= 30**



s0 → s1 — sem transições " " em s1
reconhecido *Ident*

s0 → s5 — ignora os brancos do início
não para em s4
sem transições " " em s5
reconhecido *maiorIgual*

s0 → s2 — ignora os brancos do início
sem transições " " em s2
reconhecido *número*

UNIP Interativa

Portanto, a análise léxica deverá:

- esquadrinhar o código fonte, símbolo a símbolo, compondo *tokens*, e classificá-los;
- eliminar elementos desnecessários ao processo de compilação;
- reconhecer e validar números inteiros e reais;
- reconhecer e validar os elementos utilizados como identificadores;
- prover recursos para que se projete um mecanismo de controle de erros mais amigável;
- interagir com o sistema de arquivos.

Interatividade

Dentre os diferentes tipos de mensagens de erro que podem ser reportados por um compilador, quais dentre as apresentadas abaixo são de natureza léxica?

a) **Identificador não declarado.**

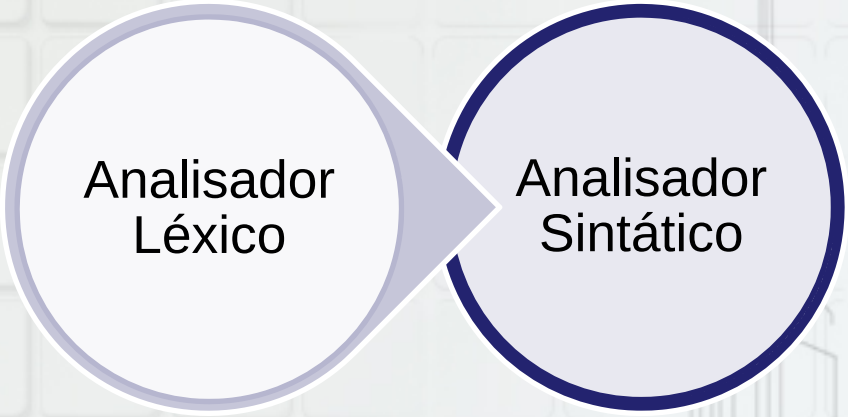
b) **Esperado fim de comentário.**

c) **Esperado símbolo X, porém encontrado símbolo Y.**

d) **Número de parâmetros insuficiente (durante a chamada de uma sub-rotina).**

e) **Tipo misturado (durante uma atribuição).**

Análise sintática:
2ª etapa do processo de análise



- ***Tokens***

- **Árvore sintática**
- **Tabela de símbolos**

UNIP
Interativa

A análise sintática deve:

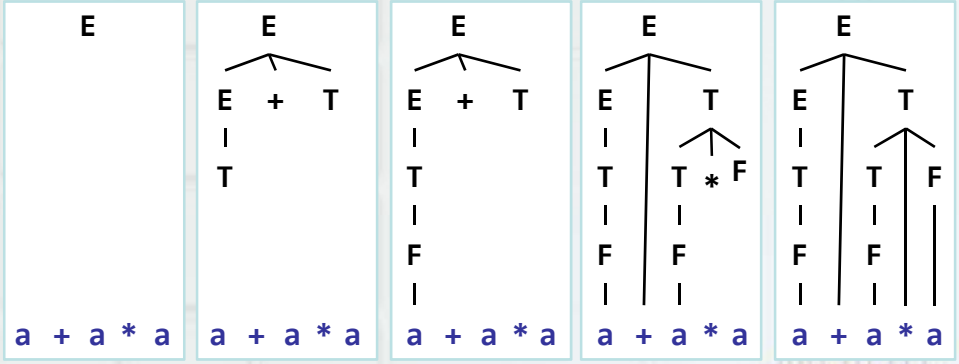
- Comprovar que a sequência de *tokens* cumpre as regras sintáticas da linguagem:
  - identificar erros de sintaxe.
- Compor a estrutura hierárquica dos comandos e expressões:
  - A / B * C (A/B) * C em Fortran
    A / (B*C) em APL
- Recuperação de erros de sintaxe.
  - Importante: não retardar, de forma significativa, o processamento de programas corretos.

Especificando a linguagem por meio de Gramáticas Livres de Contexto

- Gramáticas regulares não podem lidar com estruturas aninhadas ou com recursões centrais:
  - Expr ➊... "(" Expr ")" ...
  - Cmd ➊ "do" Cmd "while" "(" Expr ")“
- Solução: Gramáticas Livres de Contexto.
- Vantagens na utilização de gramáticas:
  - especificações sintáticas;
  - permite uso de geradores automáticos;
  - o processo de construção pode levar à identificação de ambiguidades;
  - facilidade em ampliar/modificar a linguagem.

Tipos de analisadores sintáticos

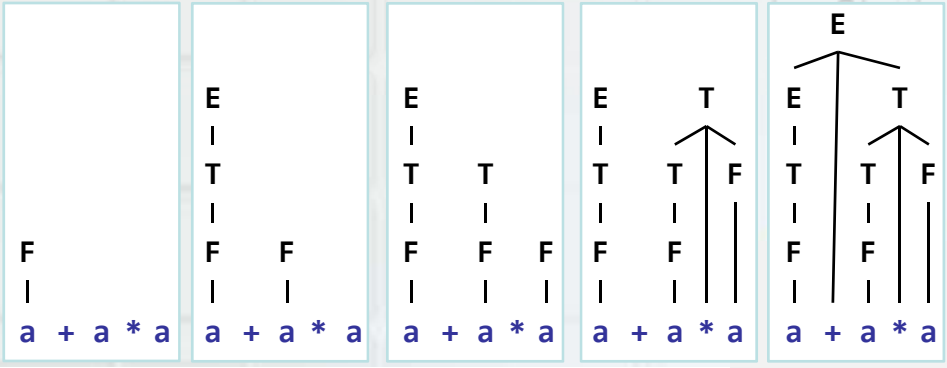
- **Métodos descendentes (*Top Down*):**
- **Constroem a árvore sintática de cima para baixo (da raiz para as folhas), ou seja, do símbolo inicial da gramática para a sentença:**
  - **Analisadores descendentes recursivos;**
  - **Analisadores LL(k).**



UNIP
Interativa

Tipos de analisadores sintáticos

- **Métodos ascendentes (*Bottom-up*):**
- **Constroem a árvore sintática de baixo para cima (das folhas para a raiz), ou seja, reduz os símbolo da sentença até alcançar o símbolo inicial da gramática:**
  - **Analisadores LR;**
  - **Analisadores LALR.**



Tipos de analisadores sintáticos

- Tanto em analisadores ascendentes quanto descendentes a entrada sempre é examinada da esquerda para a direita, um símbolo por vez.
- Muitos compiladores são dirigidos pela sintaxe (*parsen driver*), isto é, o analisador sintático chama o analisador léxico.
- Ferramentas para geração automática de analisadores sintáticos baseiam-se na gramática.
  - Ex.: Yacc, Bison e ANTLR.

Recuperação de erros

- **Modo Pânico ou Desespero:**
  - **para imediatamente; ou**
  - **descarta símbolos até que seja encontrado um *token* de sincronização.**
- **Recuperação de frases:**
  - **tenta realizar uma correção local, substituindo alguns elementos que permitam à análise prosseguir.**
    - **Ex.: substituir uma vírgula inadequada por um ponto e vírgula, remover um “:” excedente.**

Recuperação de erros (cont.)

- **Produções de erro:**
  - **aumenta-se a gramática, incluindo regras de forma a acomodar os erros mais comuns.**
- **Correção global:**
  - **um algoritmo escolhe a sequência mínima de mudanças necessárias para se obter a correção.**
    - **Ex.: dada uma cadeia x, o *parser* procura árvores gramaticais que permitam transformar x em y (cadeia correta) com um mínimo de modificações.**

Interatividade

Analise o texto:

Na compilação, a análise consiste em três fases. Em uma das fases, os caracteres ou *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo. Essa fase envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, que são usadas pelo compilador, a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical.

A fase citada no texto é conhecida como análise:

a) **sintática.**
b) **semântica.**
c) **léxica.**
d) **binária.**
e) **linear.**

Análise sintática

- Tarefa: dada uma gramática livre de contexto G e uma sentença s, o analisador sintático deve verificar se s pertence à linguagem gerada por G.
  - O analisador tenta construir a árvore de derivação para s segundo as regras de produção dadas pela gramática G.
  - Se esta tarefa for possível, o programa é considerado sintaticamente correto.
- O analisador não precisa efetivamente construir a árvore, mas sim comprovar que é possível construí-la.
  - Pode ser emulado utilizando-se uma pilha de dados.

Análise sintática descendente: analisadores descendentes recursivos

- São construídos transcrevendo-se cada uma das regras de produção da gramática como uma sub-rotina que será responsável por consumir os *tokens* da sentença.

Assim, temos:

- para cada símbolo não terminal da regra, invocamos a sub-rotina correspondente; e
- para cada símbolo terminal da regra verificamos se ele ocorre na posição corrente da análise.

Análise sintática descendente: analisadores descendentes recursivos

Suponha a gramática abaixo:

(1) $L \rightarrow (S)$

(2) $S \rightarrow I , S \mid I$

(3) $I \rightarrow a \mid L$

(1)
```
void parseL() {
   accept("(");
   parseS();
   accept(")");
}
```

(2)
```
void parseS() {
   parseI();
   while s==","  {
      acceptIt();
      parseI();
   }
}
```

(3)
```
void parseI() {
  switch s {
  case "a": acceptIt();
            break();
  case "(": parseS();
            break;
  default: ERRO();
  }
}
```

UNIP Interativa

Análise sintática descendente: analisadores LL(1)

O nome LL(1) indica que:

- **a cadeia de entrada é examinada da esquerda para a direita (*left-to-right*);**
- **o analisador procura construir uma derivação esquerda (*leftmost*);**
- **considera-se apenas o 1º símbolo do restante da entrada.**

Análise sintática descendente: analisadores LL(1)

- Temos de decidir qual regra $A \rightarrow \alpha$ deve ser aplicada a um nó rotulado por um não terminal A.
  - A expansão de A é feita criando nós filhos rotulados com os símbolos de α.
- Considera duas informações:
  - o não terminal a ser expandido; e
  - o símbolo corrente da entrada.
- Uma tabela M nos fornece a regra a ser utilizada com base nessas duas entradas.
- Essa técnica só pode ser usada para a classe das gramáticas LL(1).

Analisador LL(1)

Seja a gramática:

(1) E → T E‘	First(TE’) = { (, a}
(2) T → F T‘	First(FT’) = { (, a}
(3) F → (E)	First((E)) = { (}
(4) F → a	First(a) = { a }
(5) E' → + T E‘	First(+TE’) = { + }
(6) E' → ε	Follow(E’) = {), *$*}
(7) T' → * F T‘	First(*FT’) = { * }
(8) T' → ε	Follow(T’) = { +,), *$*}

Analisador LL(1)

Seja a gramática:

(1) E → T E‘

(2) T → F T‘

(3) F → (E)

(4) F → a

(5) E' → + T E‘

(6) E' → ε

(7) T' → * F T‘

(8) T' → ε

First(TE’) = { (, a}

M[E, ( ] = 1

M[E, a] = 1

UNIP
Interativa

Analisador LL(1)

Seja a gramática:

(1) $E \rightarrow T\ E'$

(2) $T \rightarrow F\ T'$

(3) $F \rightarrow (\ E\ )$

(4) $F \rightarrow a$

(5) $E' \rightarrow +\ T\ E'$

(6) $E' \rightarrow \varepsilon$ ⟶ Follow(E') = {), $}

(7) $T' \rightarrow *\ F\ T'$ M[E',)] = 6

(8) $T' \rightarrow \varepsilon$ M[E', $] = 6

UNIP
UNIVERSIDADE PAULISTA
Interativa

Analisador LL(1)

Seja a gramática:

(1) E → T E‘

(2) T → F T'

(3) F → (E)

(4) F → a

(5) E' → + T E‘

(6) E' → ε

(7) T' → * F T‘

(8) T' → ε

	(	a	+	*	)	$
E	1	1	-	-	-	-
T	2	2	-	-	-	-
F	3	4	-	-	-	-
E'	-	-	5	-	6	6
T'	-	-	8	7	8	8

UNIP Interativa

Analisando a sentença: a+a

Pilha	Entrada	Regra	
E	a+a	M[E, a] = 1	E → T E‘
T E'	a+a	M[T, a] = 2	T → F T '
F T 'E'	a+a	M[F, a] = 4	F → a
a T 'E'	a+a	*Verifica* a	
T 'E'	+a	M[T', +] = 8	T ' → ε
E'	+a	M[E', +] = 5	E' → + T E‘
+ T E'	+a	*Verifica* +	
T E'	a	M[T, a] = 2	T → F T '
F T ' E'	a	M[F, a] = 4	F → a
a T ' E'	a	*Verifica* a	
T ' E'	ε	M[T', *$*] = 8	T ' → ε
E'	ε	M[E', *$*] = 6	E' → ε
ε	ε	-	

Identificando se uma gramática é LL(1)

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → a

First(E+T) = { (, a }
First(T) = { (, a }

M[E, (] = 1 e M[E, a] = 1
M[E, (] = 2 e M[E, a] = 2

- A gramática dada acima não é LL(1) por causa dos conflitos.
- Podemos descobrir por inspeção. As duas características mais óbvias são:
  - a possibilidade de fatoração; e
  - a recursão à esquerda.

Interatividade

Analise cada uma das afirmações dadas a seguir e indique a que julgar incorreta.

a) **Os *parsers top-down* não têm problemas em relação a gramáticas recursivas à esquerda.**

b) ***Yacc* gera *parsers bottom-up*, que são mais eficientes.**

c) **Os *parsers bottom-up* são normalmente gerados por ferramentas.**

d) ***Parsers* descendentes recursivos são um exemplo de *parser top-down*.**

e) **É relativamente fácil escrever um *parser top-down* manualmente, usando funções recursivas.**

UNIP
Interativa

ATÉ A PRÓXIMA!

UNIP
UNIVERSIDADE PAULISTA
Interativa