



Paradigmas de Linguagens de programação

Prof. Salvador



SINTAXE DE LINGUAGENS DE PROGRAMAÇÃO

- **É uma descrição precisa de todos os seus programas gramaticalmente corretos.**
- **A Sintaxe tem sido definida por métodos formais desde a década de 60.**



SINTAXE DE LINGUAGENS DE PROGRAMAÇÃO

- **Quem usa definições de linguagens?**
 - **Projetistas de linguagens**
 - **Implementadores**
 - **Programadores**
- **Sintaxe** – a forma ou estrutura de expressões, comandos ou unidades do programa.
- **Semântica** – o significado das expressões, comandos e unidades do programa.



Descrivendo a sintaxe

- Uma **sentença** é uma cadeia de caracteres sobre um alfabeto.
- Uma **linguagem** é um conjunto de sentenças
- Um **lexema** é a menor unidade sintática de uma linguagem. Ex.: scanf, read, if, else.
- Um **token** é uma categoria de lexemas. Ex.: identificador.



Descrivendo a sintaxe

- **Formalismos para descrever a sintaxe:**

- **Reconhecedor**

- Um dispositivo de reconhecimento lê uma cadeia de entrada e decide se a mesma pertence à linguagem.
- Exemplo: *o analisador sintático* de um compilador

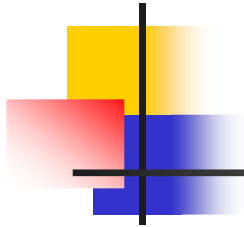
- **Gerador**

- Um dispositivo que gera sentenças de uma linguagem. Ex.: uma gramática.



Hierarquia de Chomsky

- **Linguagens regulares – tipo 3**
 - As mais simples. Reconhecidas por AFD.
- **Linguagens livre de contexto – tipo 2**
 - Permite construções do tipo parênteses balanceados.
 - Reconhecidas por autômato de pilha.
- **Linguagens dependente de contexto – tipo 1**
 - Gramáticas possuem regras do tipo $aAb \rightarrow ayb$.
 - Reconhecidas por máquina de Turing.
- **Linguagens recursivamente enumeráveis – tipo 0**
 - Abrange todas as classes de linguagens.

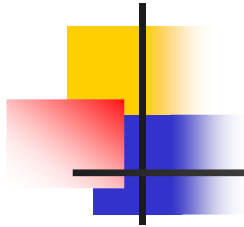


Formalmente as gramáticas, são caracterizadas como quádruplas ordenadas

$$G = (V, T, P, S)$$

onde:

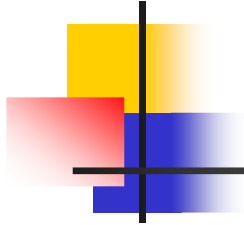
V representa o vocabulário não terminal da gramática. Este vocabulário corresponde ao conjunto de todos os símbolos dos quais a gramática se vale para definir as leis de formação das sentenças da linguagem.



$$G = (V, T, P, S)$$

T é o vocabulário terminal, contendo os símbolos que constituem as sentenças da linguagem. Dá-se o nome de terminais aos elementos de T.

P representa o conjunto de todas as leis de formação utilizadas pela gramática para definir a linguagem. A cada uma dessas regras de formação que compõem o conjunto P dá-se o nome de produção da gramática.

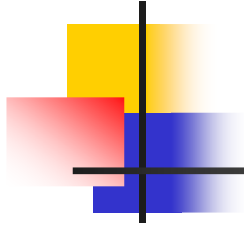


Cada produção P tem a forma:

$$\alpha \rightarrow \beta \quad \alpha \in (V \cup T)^+; \quad \beta \in (V \cup T)^*$$

$$G = (V, T, P, S)$$

$S \in V$ é dito o símbolo inicial ou o axioma da gramática. Indica onde se inicia o processo de geração de sentenças.



Exemplo:

$G = (\{A, B\}, \{0, 1\}, P, A)$

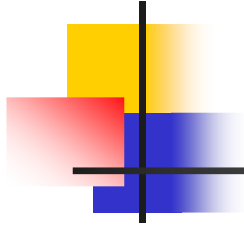
$P: A \Rightarrow 0A$

$A \Rightarrow B$

$B \Rightarrow 1B$

$B \Rightarrow \lambda$

Qual a linguagem gerada?



Exemplo:

$G = (\{A, B\}, \{0, 1\}, P, A)$

$P: A \Rightarrow 0A$

$A \Rightarrow B$

$B \Rightarrow 1B$

$B \Rightarrow \lambda$

Qual a linguagem gerada?

Resp.: $L(G) = \{0^n 1^m; n \geq 0, m \geq 0\}$



Métodos formais para descrever a sintaxe

- **Gramáticas livres de contexto**
 - **Desenvolvidas por Noam Chomsky em meados dos anos 1950.**
 - **Definem uma classe de linguagens chamadas linguagens livres de contexto.**
 - **Também podem ser usadas para descrever linguagens de programação (com poucas exceções).**



Gramática livre de contexto

**Seja uma gramática livre de contexto,
definida como $G = (V, T, P, S)$**

Características:

- **Produções do tipo:**
 $A \rightarrow a$, onde $A \in V$
- **Construções aninhadas**
- **Linguagem reconhecida por autômato de pilha**

Métodos formais para descrever a sintaxe



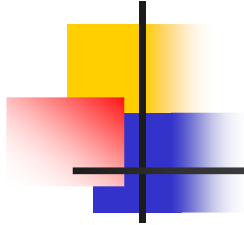
- **Backus-Naur Form e gramáticas livre de contexto**
 - Métodos mais usados para descrever a sintaxe de linguagens de programação
- **BNF estendida**
 - Acrescenta legibilidade à BNF



Métodos formais para descrever a sintaxe

■ Backus-Naur Form (1959)

- Inventada por John Backus para descrever Algol 58; modificada por Peter Naur para descrever Algol 60.
- BNF é equivalente a gramáticas livres de contexto. É uma metalinguagem.
- Uma **metalanguage** é uma linguagem usada para descrever outra.
- Em BNF, abstrações são usadas para representar classes de estruturas sintáticas. Usam variáveis sintáticas: símbolos não terminais.



- **Em BNF uma gramática é definida por um conjunto de regras**
- **Uma regra tem um lado esquerdo (LE) e direito (LD)**

<bloco> ::= <instruções> <instrução>

- **Os símbolos que constituem as regras BNF podem ser terminais ou não terminais**
 - Os símbolos não terminais são delimitados por < e >
 - Os símbolos terminais não são delimitados
- **Em BNF existem dois operadores:**
 - ::= o símbolo não terminal do LE deve ser substituído pelos símbolos terminais ou não terminais do LD
 - | permite especificar símbolos alternativos no LD



BNF estendida

- **O padrão ISO 14977 define uma extensão à BNF designada EBNF, na qual existem quatro novos operadores:**
 - **Símbolos terminais são colocados entre aspas**
 - **(... | ... | ...) escolha múltipla**
 - **[] símbolos opcionais (zero ou uma vez)**
 - **{ } símbolos opcionais com repetição (zero ou mais vezes)**
 - **{ }+ símbolos com repetição (uma ou mais vezes)**
 - **Cada regra tem um carácter final explícito, de modo que nunca há ambiguidade sobre onde a mesma termina**



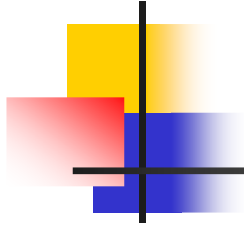
Associatividade e precedência

- **Um operador possui maior precedência que um segundo se, quando aparecerem em uma expressão sem parênteses, o primeiro sempre for analisado antes do segundo.**
- **Associatividade especifica se os operadores de mesma precedência são analisados da esquerda para a direita ou da direita para a esquerda.**

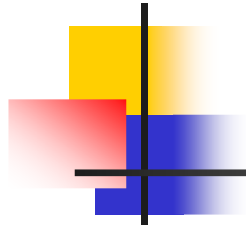


Especificação de uma linguagem

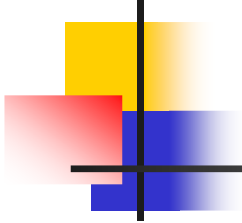
- programa = [declaração-de-variáveis]
 { declaração-de-procedimento }
 sequência-de-comandos ";" .

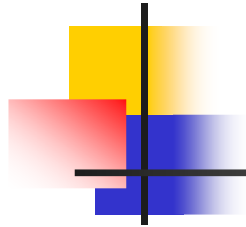


- declaração-de-variáveis = decl-de-variáveis
{ ";" decl-de-variáveis } ";" .
- decl-de-variáveis = { "**INT**" lista-de-nomes |
- "**CHAR**" lista-de-nomes } .

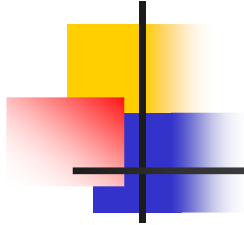


- decl-de-procedimento =
"PROC" nome "(" lista-de-nomes ")"
sequência-de-comandos "END" ";" .
lista-de-nomes = nome { "," nome } .

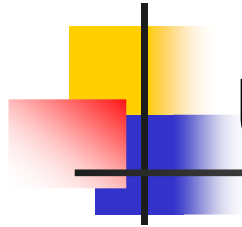
- 
-
- sequência-de-comandos = comando {
“;” comando } .
 - comando = [atribuição | leitura |
impressão | decisão | chamada | desvio
] .



- atribuição = nome “:=” expressão.
- leitura = “**READ**” [lista-de-nomes] .
- impressão = “**PRINT**” lista-de-expressões .



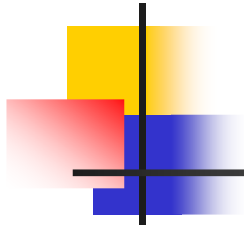
- lista-de-nomes = nome { “,” nome } .
- nome = letra { letra | dígito } .
- letra = “a” | “b” | “c” | “d” | “e” | “f” |
.....



Um pouco de história

Em 1954 a IBM lança o 704, a primeira máquina de sucesso comercial.

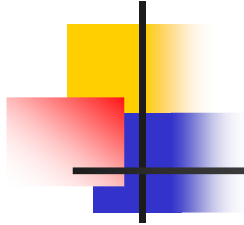
Um aspecto curioso: os usuários constataram que os custos de *software* eram altíssimos em relação aos de hardware, numa época em que o custo do *hardware* era bastante alto.



O IBM 704 trouxe o recurso de ponto flutuante e indexação em hardware. Foi um avanço considerável em relação às demais máquinas da época.

É discutível o crédito do primeiro compilador.

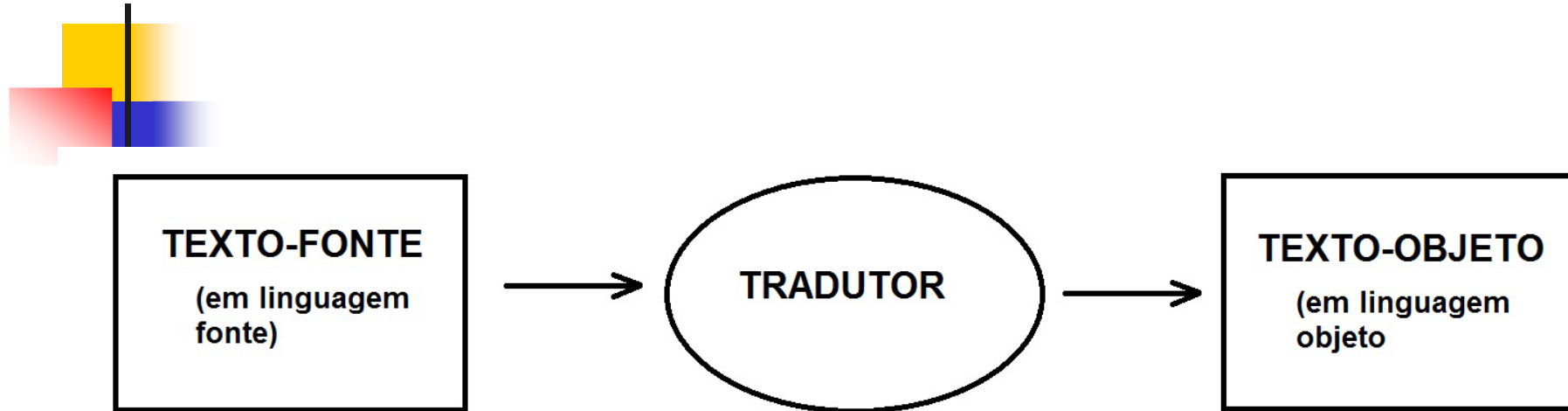
Foi John Bakus, em 1954, que lançou o **FOR**mula **TRAN**slation – ou FORTRAN, considerada a primeira linguagem de alto nível compilada, de ampla aceitação.



O compilador FORTRAN foi lançado em abril de 1957.

Em 1958 cerca de metade dos programas existentes eram em FORTRAN.

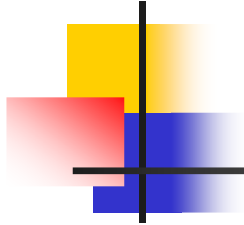
Teve uma influência que perdura até o presente.



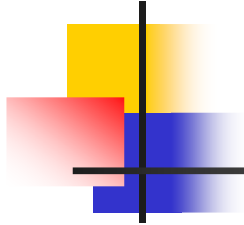
Esquema de conversão efetuado por um tradutor

A linguagem-fonte pode ser, ou não, de alto nível.

Quando a primeira linguagem (linguagem-fonte) é de alto nível, o tradutor recebe o nome de COMPILADOR.



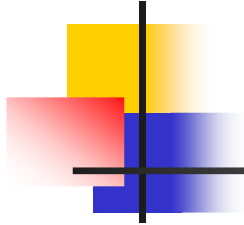
- Quando a linguagem-fonte é de baixo nível, o tradutor é chamado de **montador**, como é o caso das linguagens de montagens (*assembly languages*).



- O que é um compilador?

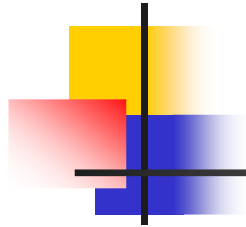
Um dos programas do software básico.

Tarefa principal: fazer a conversão automática de programas-fonte em um equivalente, que pode ser executado em um computador.



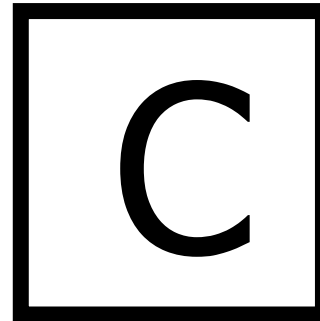
- Interpretadores x Compiladores
- O que é um interpretador





■ Compilador

Programa fonte →



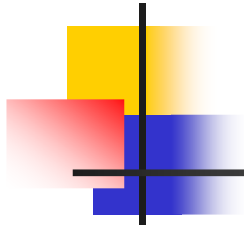
→ Executável

Entrada



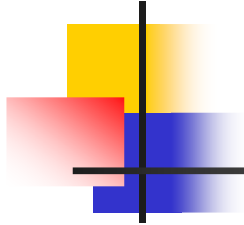
Saída



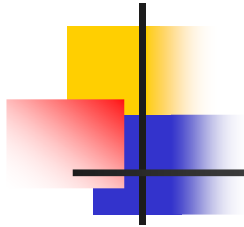


Compilador x interpretador

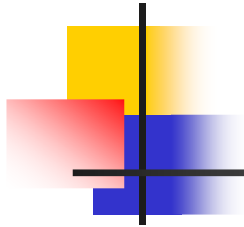
Interpretador	Compilador
Lê o programa fonte	Lê o programa fonte
Processa linha a linha	Varre todo o programa e produz um módulo executável
Dados fornecidos ao mesmo tempo que o progr. fonte	Dados serão lidos somente pelo módulo executável
"on line" – execução depende do interpretador	"off line" – execução independe do compilador



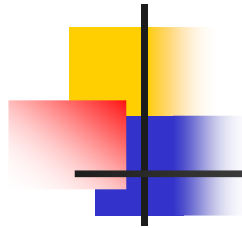
- Uma função importante do compilador é listar quaisquer erros detectados durante o processo de compilação.
- Interpretador: executa diretamente os comandos do programa fonte, processando as entradas do usuário.
- Um interpretador antigo, muito simples, porém eficiente, foi o Basic.



- Os processadores da linguagem Java combinam compilação com interpretação.
- O programa fonte passa por um tradutor, que gera um código intermediário, chamado de *bytecodes*.
- O código intermediário mais a entrada do usuário são processados por uma máquina virtual, gerando a saída.
- Uma vantagem reside em os bytecodes poderem ser gerados em uma máquina e processados em outra, via rede, por exemplo.



- Formas de organização de um compilador:
- Compilador de um único passo: quando lê o programa-fonte apenas uma vez e produz o código objeto.
 - Vantagem: eficiência
 - Desvantagem: dificuldade de introduzir otimização
- Compilador de múltiplos passos: A execução de uma fase termina antes da execução da próxima.
 - Vantagem: possibilidade de otimização
 - Desvantagem: aumenta o tempo de compilação.



Estrutura de um compilador

- O processo de compilação é realizado em duas grandes fases:
 - Análise – (back end) divide o programa fonte em partes e realiza uma análise léxica e sintática. Caso sejam encontrados erros, reporta-os ao usuário.
Caso contrário, cria uma tabela de símbolos e gera o código intermediário, que são passados para a fase de síntese.
 - Síntese – (front end) constrói o programa objeto desejado.

Fases da compilação

Programa fonte



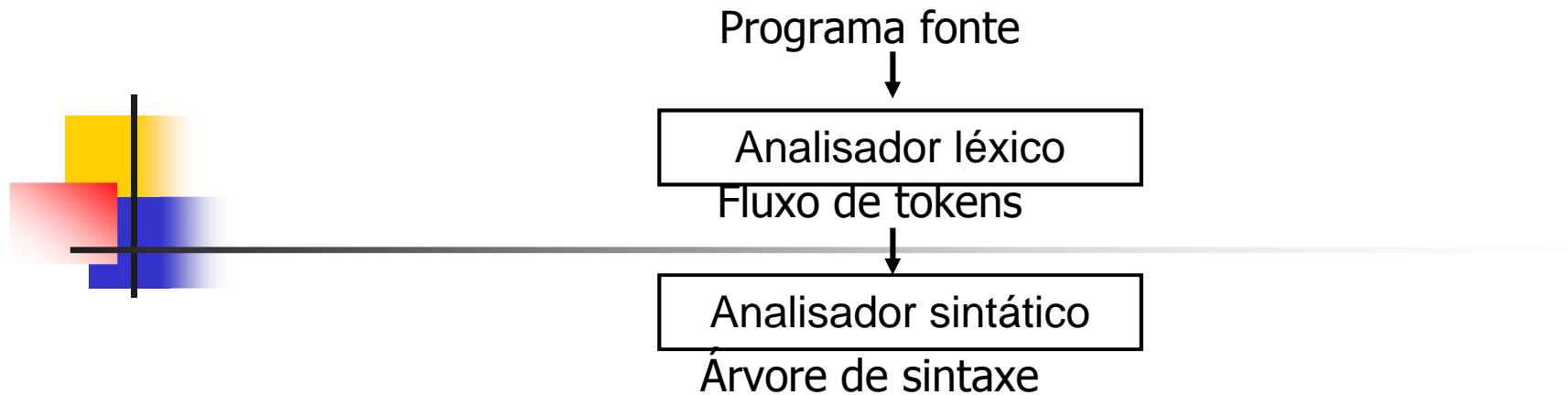
Analizador léxico

Fluxo de tokens

O analisador léxico lê um fluxo de caracteres que compõem o programa fonte e os agrupa em sequências significativas, chamadas lexemas.

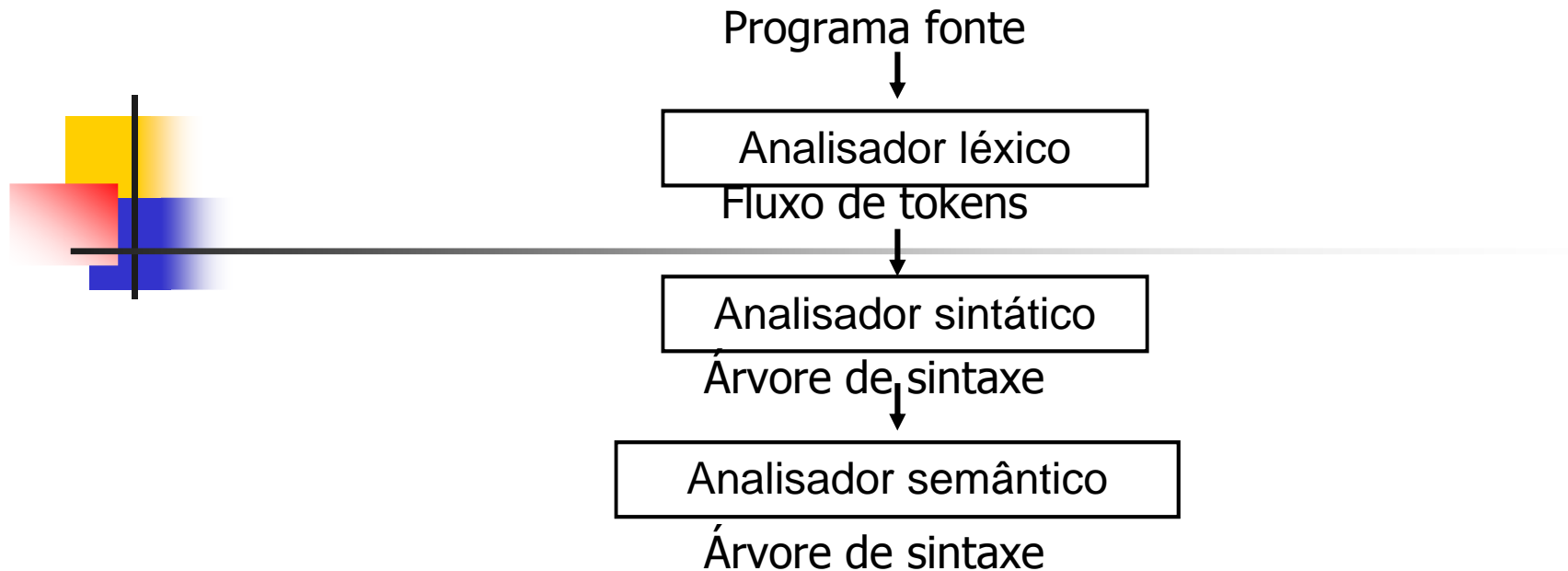
Os lexemas dão origem a uma tabela de tokens e uma tabela de símbolos.

Fases da compilação

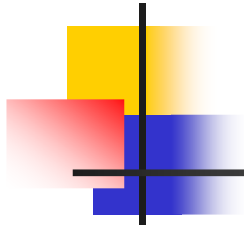


- O analisador sintático utiliza os tokens produzidos pelo analisador léxico para criar uma representação intermediária tipo árvore.
- Na árvore de sintaxe cada nó interior representa uma operação, e os filhos do nó representam os argumentos da operação.

Fases da compilação

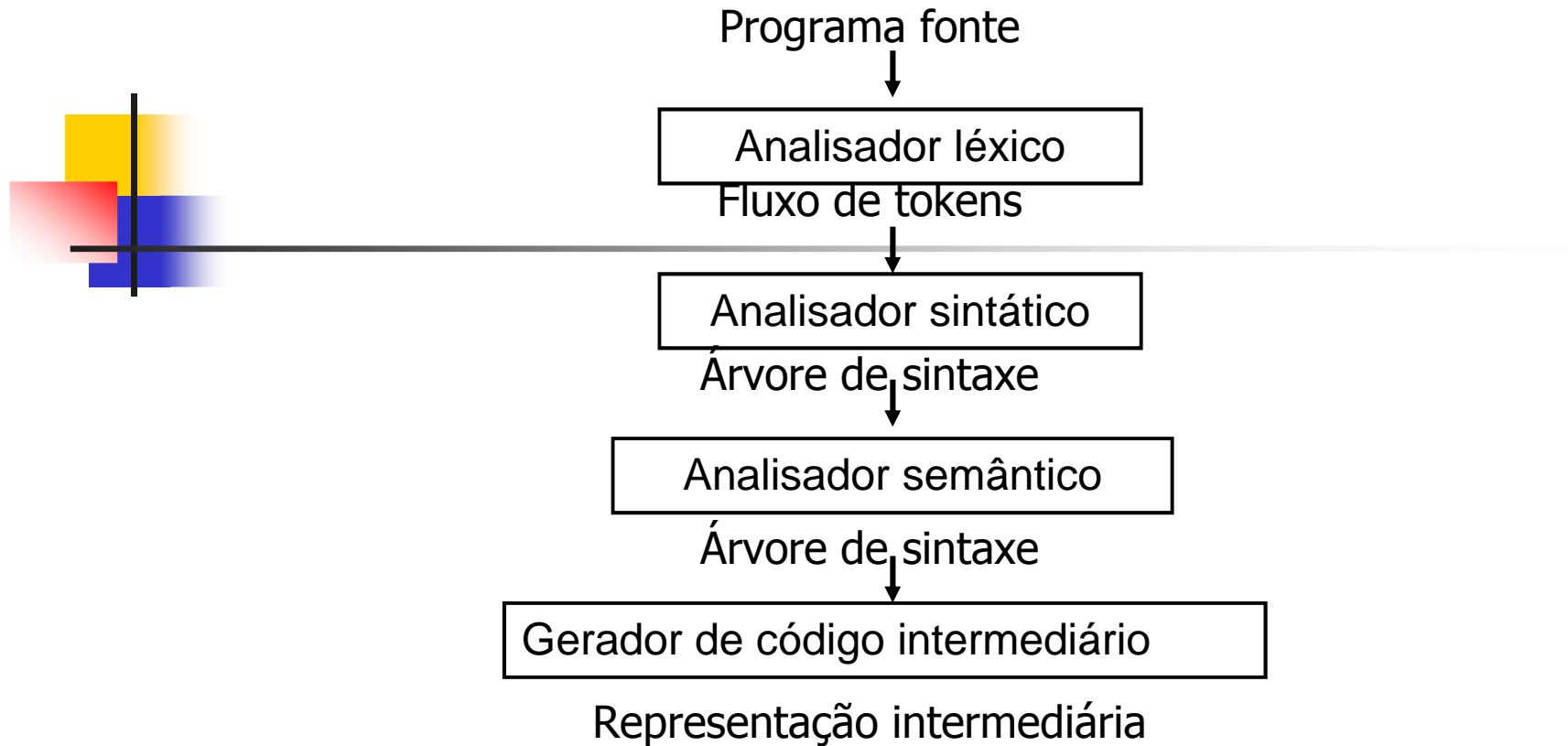


O analisador semântico utiliza a árvore de sintaxe e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem.



- O analisador semântico também reúne informações gerais sobre os tipos e as salva na árvore de sintaxe ou na tabela de símbolos, para uso na geração de código intermediário.
- Exemplo: verificação de tipo – Exigência que um índice de vetor seja um inteiro.
- Coerções: conversões de tipos.

Fases da compilação



Reproduz o código fonte em um código intermediário.

Exemplo:

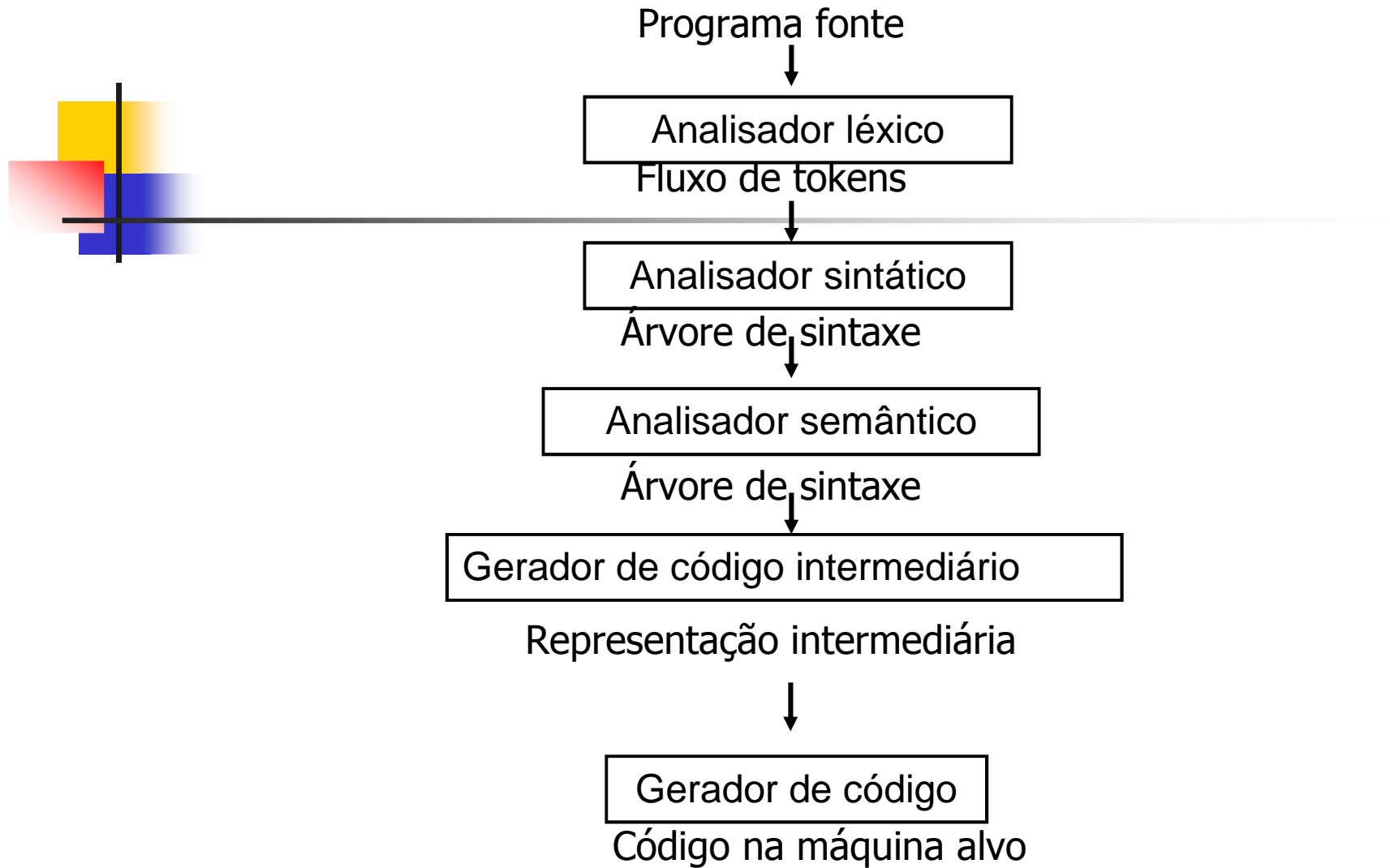
t1 = inttofloat (60)

t2 = id3 * t1

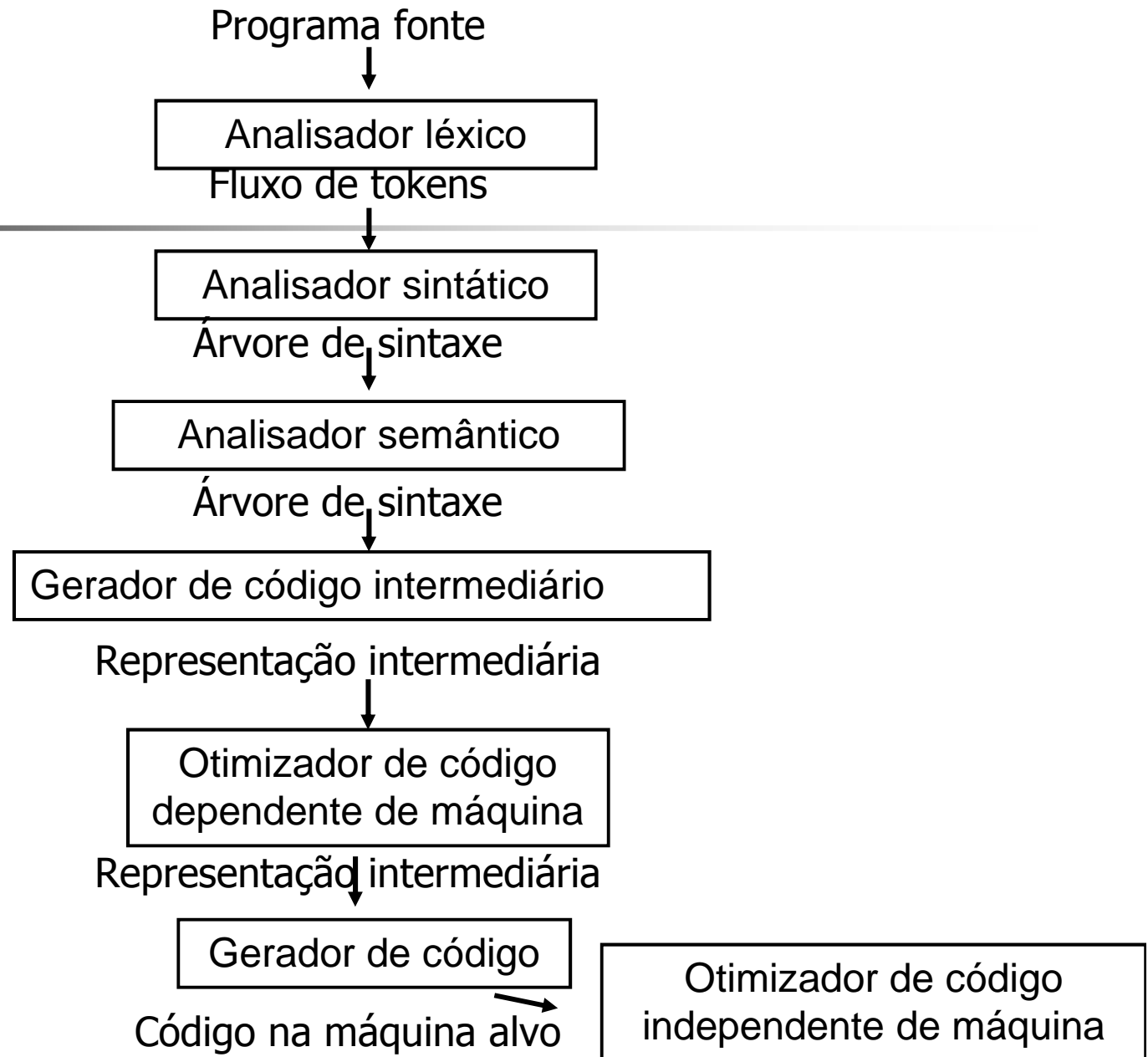
t3 = id2 + t2

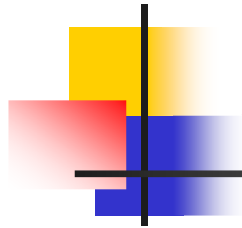
id1 = t3

Fases da compilação



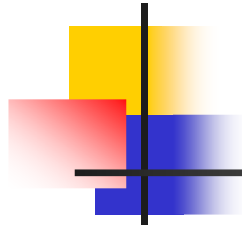
Fases da compilação





Análise Léxica/sintática

- Um analisador sintático, geralmente, é composto de duas partes:
 - Análise léxica (baseada em autômatos finitos e gramáticas regulares)
 - Análise sintática (baseada em autômato de pilha e gramática (E)BNF)



Razões para separar análise léxica e sintática

- Simplicidade – separar o analisador léxico do *parser* é uma abordagem menos complexa.
- Eficiência – a separação permite otimização do analisador léxico.
- Portabilidade – a parte do analisador léxico pode não ser portátil, mas o *parser* sempre é portátil.

Análise léxica



- Tem por objetivo realizar a extração e classificação dos átomos.
 - Classes de átomos mais frequentemente encontradas
 - Identificadores
 - Palavras reservadas
 - Números (inteiros, reais, com ou sem sinal)
 - Cadeias de caracteres ("strings")
 - Sinais de pontuação, operadores
 - Caracteres especiais



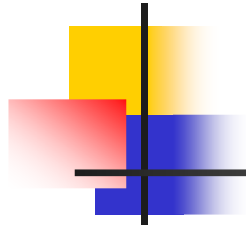
Análise léxica

- Busca ler e agrupar os caracteres significativos de um programa fonte em sequências chamadas *lexemas*.
 - Lexema: A menor unidade depois dos caracteres.

Isto é uma sentença.

Isto é uma sentença.

Ist o éu mase ntença.
- Um analisador léxico é um casador de padrões.




- Os lexemas produzem *tokens*, na forma

<nome-token, valor-atributo>

que são passados para a fase de análise sintática.

nome-token – é um símbolo abstrato

valor-atributo – ponteiro para uma entrada da tabela de símbolos



Exemplo de token: suponha uma linha do programa fonte do tipo

`posicao = inicio + taxa * 60`

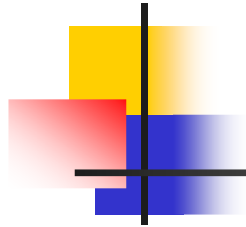
- `posicao` é um lexema, mapeado em token para `<id, 1>`
- `=` é um lexema que não necessita de atributo e é mapeado em token `< = >`.
- O processo segue, de modo que se terá, ao final, a representação do comando em uma sequência de tokens:

`<id,1> < = > <id,2> <+> <id,3> <*> <60>`



Terminologia

- Lexema – sequencia de caracteres que casa com um padrão de token.
- Padrão – é uma descrição da forma que o lexema de um token pode assumir.
Ex.: em uma palavra-chave o padrão é a sequencia de caracteres que formam palavra-chave.
- Token – consiste de um nome e um valor de atributo opcional
O nome do token representa um tipo de unidade léxica.
Exemplos: palavra chave da linguagem, identificador, número.



- Token

IF

Id

Número

Literal

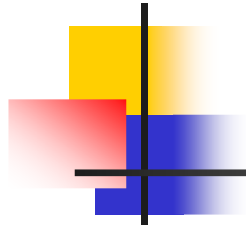
Exemplos de lexemas

IF

pi, xd1, v2a

35, 44.5

“sequencia entre aspas”



Elimina delimitadores (espaço em branco, tabulação) e comentários.

Realiza conversão numérica

Valores numéricos pode vir em notações diversas: binário, hexadecimal, notação científica, etc.

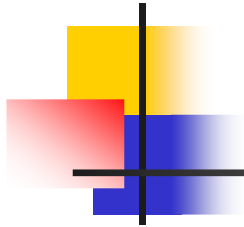
Tratamento de identificadores

Cadeias com comprimento e composição não padronizados.

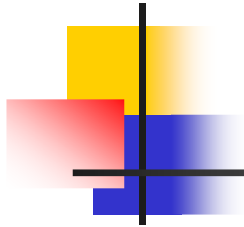
Tratamento realizado pela criação de uma tabela de símbolos.

Identificação de palavras reservadas

Identificadores com significado determinado na linguagem

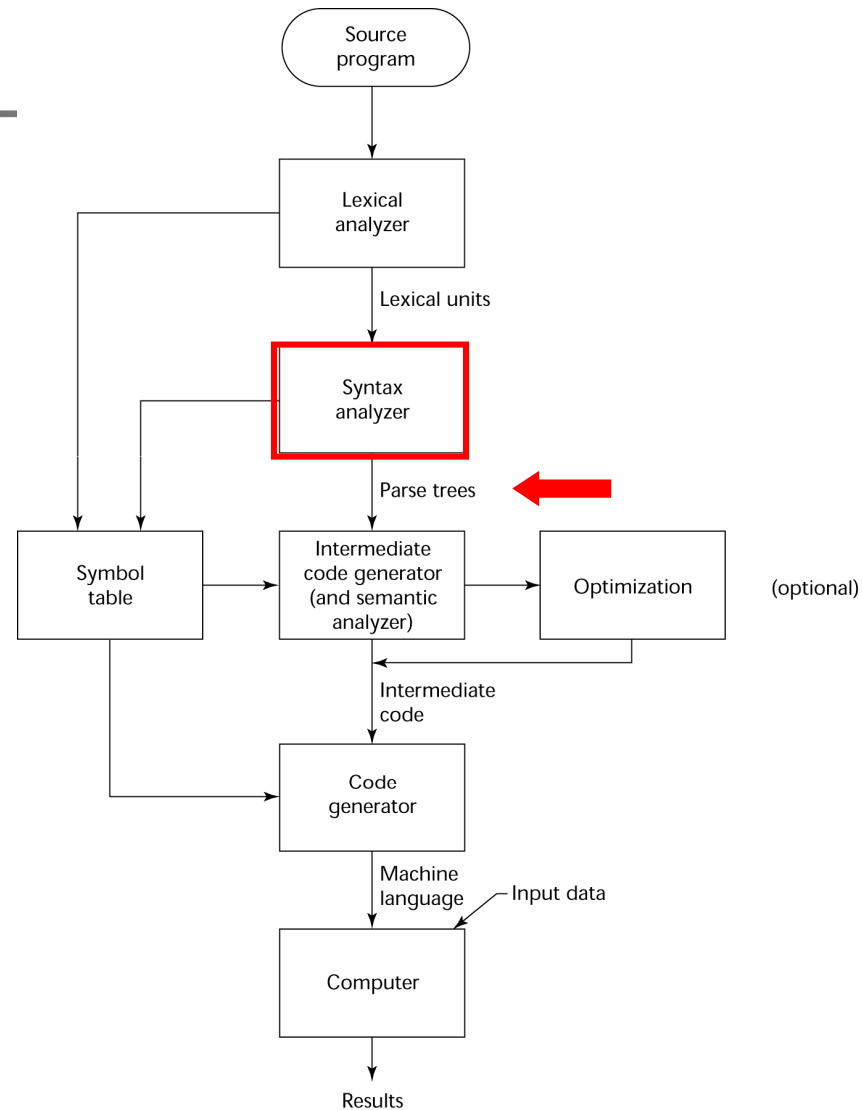
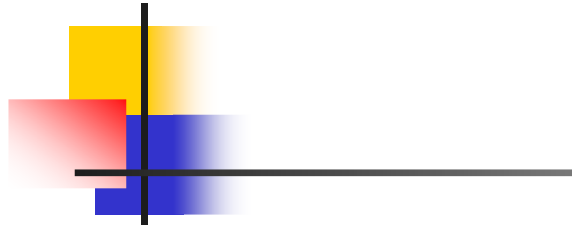


- As classes de tokens a seguir abrangem todos ou quase todos os tokens da maioria das linguagens de programação.
 - 1 – um token para cada palavra chave
 - 2 – tokens para os operadores, individualmente ou em classes.
 - 3 – Um token para todos os identificadores.
 - 4 – um ou mais tokens representando constantes e cadeias de caracteres.
 - 5 – um token para cada símbolo de pontuação: () , ;



-
- Erros léxicos
 - É difícil para o analisador léxico reparar erros. No entanto, existem algumas técnicas para reparar erros léxicos.
 - Algumas dessas técnicas são consideradas muito dispendiosas para compensar o esforço e geralmente não são implementadas, a não ser em compiladores experimentais.

Processo de compilação





Análise Sintática

A análise léxica de uma frase da língua portuguesa reconhece os agrupamentos de caracteres válidos da linguagem (cazza não faz parte do léxico da L.P.) e os classifica: substantivo, adjetivo, artigo, etc.

No caso das linguagens de programação:

A#\$2 não é um agrupamento de caracteres válido em, praticamente, todas as linguagens de programação, enquanto 3241, THEN, ELSE, nome, são válidos e classificados como: número, palavra chave ou identificador.



Análise Sintática

Sintaxe é um conjunto de regras que estabelece como são compostas as estruturas básicas de uma linguagem.

Exemplo:

Na língua portuguesa, o agrupamento de palavras casa quarto é e grande o é a azul é válido, considerando que todas são palavras do vernáculo da língua mas carece de sentido. Quem é grande? Quem é azul?



Análise Sintática

Uma regra sintática diz que a ordem normal das palavras deve ser: sujeito seguido do predicado e que o predicado é composto por verbo e complementos.

Reorganizando: A casa **é** grande e o quarto **é** azul.



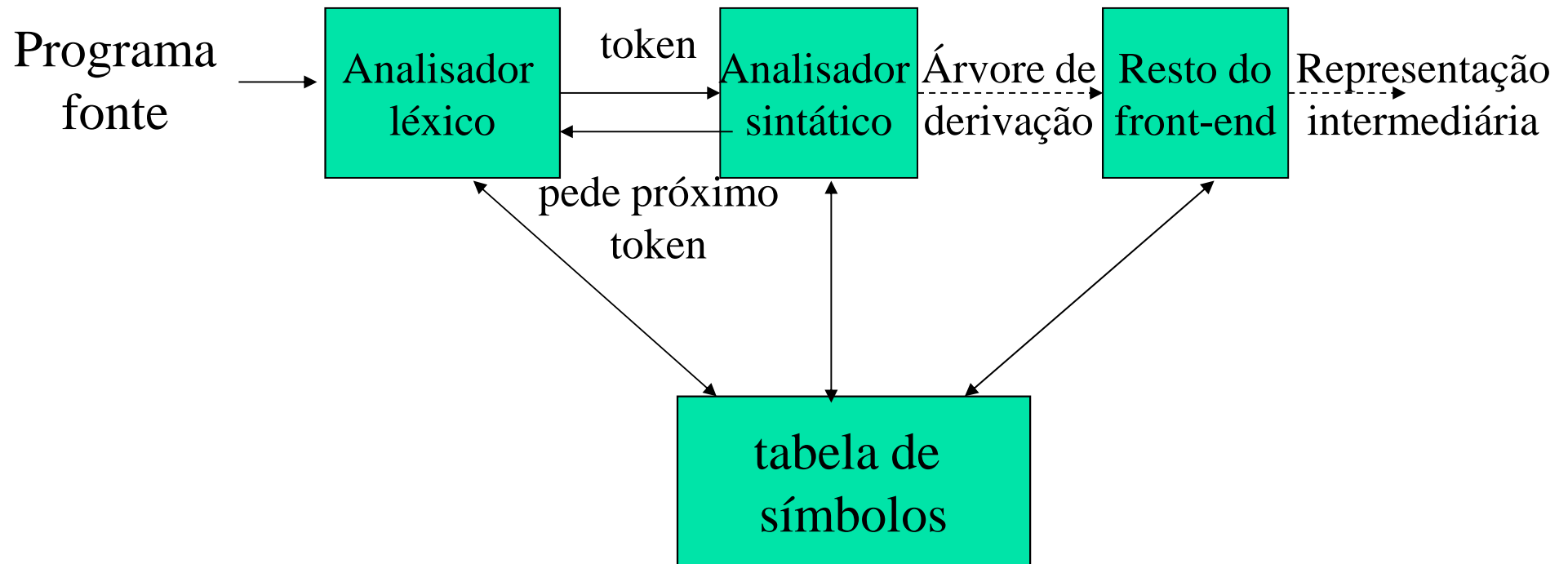
■ **Gramáticas livre de contexto**

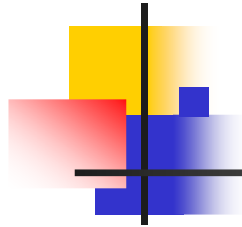
Têm sua maior aplicação na formalização de linguagens de programação devido à sua simplicidade.

Linguagens de programação apresentam características de dependência de contexto, que não podem ser representadas por gramáticas tipo 2 (livres de contexto).

A capacidade de representar construções aninhadas torna essas gramáticas adequadas para representar linguagens de programação.

Analizador Sintático - Parser



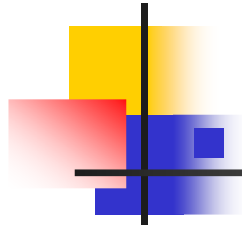


Analizador sintático

É segundo grande bloco componente de um compilador é o analisador sintático.

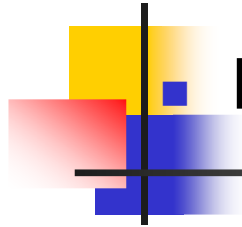
A função principal é promover a análise da sequência com que os átomos componentes do texto-fonte se apresentam.

A partir dessa análise, efetua a síntese da árvore de sintaxe.



Uma gramática:

- Oferece uma descrição da linguagem precisa e fácil de entender
- Facilita a construção de um analisador sintático que determina se um programa-fonte está sintaticamente bem formado.
- Se constitui em uma estrutura da linguagem de programação útil à tradução correta de códigos-fonte em códigos-objeto e também à detecção de erros.

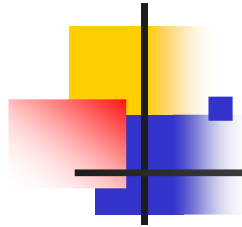


Funções da análise sintática

Identificar a sentença: o analisador pode ser visto como um aceitador de cadeias de átomos.

Detecção de erros de sintaxe: deve ter a capacidade de detectar e reportar como erro uma cadeia que não pertença à linguagem.

Recuperação de erros: uma vez detectado um erro, ressincronizar o analisador e continuar a análise do restante do texto-fonte.

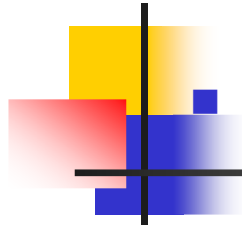


Funções da análise sintática (cont.)

Correção de erros: compiladores sofisticados incorporam mecanismos para alterar o texto-fonte. São mecanismos empíricos, pela própria natureza dos erros.

Montagem da árvore abstrata da sentença: o analisador deveria, ao menos conceitualmente, levantar a árvore de derivação da gramática.

Comando de ativação do analisador léxico: presente em compiladores de um passo.



Funções da análise sintática (cont.)

Ativação de rotinas de análise referente às dependências de contexto da linguagem: em geral são necessárias rotinas de análise sintática para tratar as dependências de contexto, presentes na linguagem.

Exemplos de dependências de contexto:

- ✓ verificação de escopo de variáveis,
- ✓ coerências de tipos de dados em expressões,
- ✓ relacionamento entre as declarações e os comandos executáveis

- 
- Analisadores que executam as funções mencionadas, implementam o núcleo dos compiladores dirigido por sintaxe.
-

- Como se trata, em geral, de linguagens não regulares, a implementação se dá, quase na totalidade dos casos, pela utilização de autômatos de pilha.
 - Um formalismo mais adequado é o autômato de pilha estruturado.
- Embora um resultado fundamental da análise sintática seja a obtenção da árvore de derivação, em muitas implementações ela não é construída fisicamente. Manifesta-se apenas conceitualmente.