Linguagens de Programação e Compiladores

Compiladores

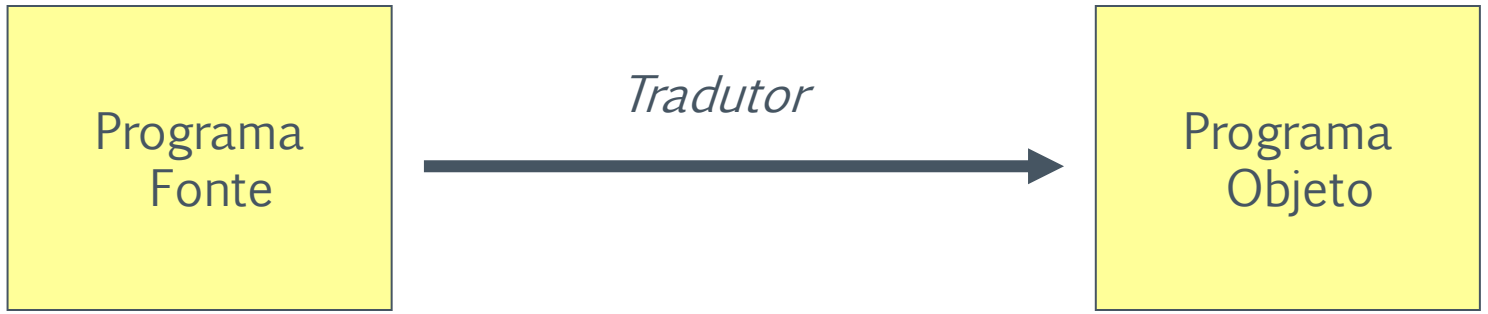
José Osvano da Silva, PMP, PSM I

Sumário

- Introdução;
- Interpretadores;
- Aspectos Básicos;
- Fases de um Compilador;
- Definições Indutivas;
- Gramáticas
- Análises

Introdução

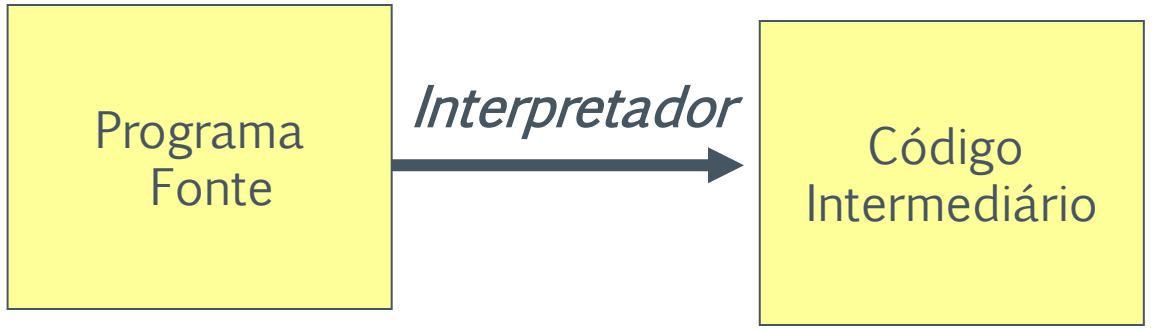
- *Linguagens:*
  - Homem: natural + notações (como a matemática)
  - Máquina: nível muito atômico (dígitos, binários, registradores, memória etc)
- Solução proposta: ling. Alto Nível



Tradutor: Compilador *ou* Interpretador

Interpretadores

Interpretador: o tradutor transforma uma L.P. numa linguagem simplificada, chamada **Código intermediário,** que pode ser diretamente executado usando um programa chamado **interpretador**.

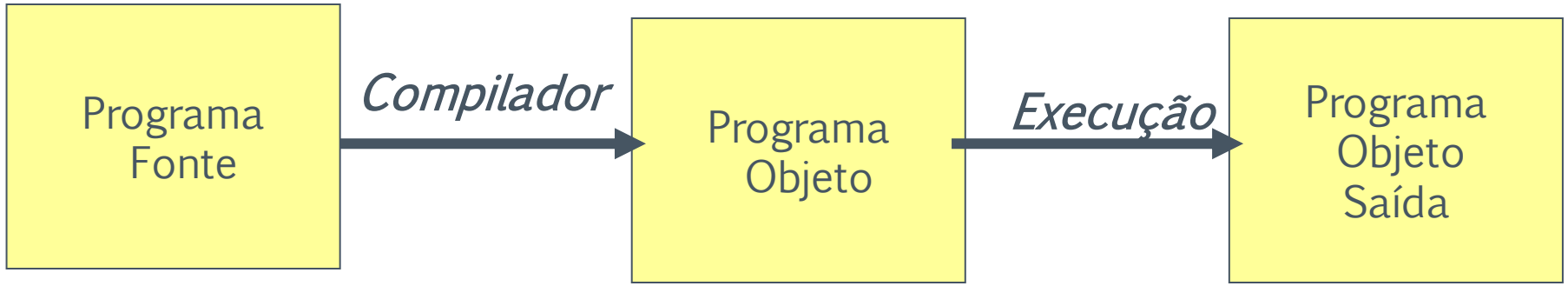


OBS: Podemos pensar na linguagem intermediária como sendo a linguagem de máquina de um computador abstrato designado a executar o código fonte.

Interpretadores

- Em alguns casos, a própria linguagem fonte pode ser a linguagem intermediária. Por ex, a maioria das linguagens de comandos, na qual nos comunicamos diretamente com o sistema Operacional, são interpretadas sem nenhuma tradução prévia.
- Os Interpretadores são em geral, menores que os Compiladores e facilitam as implementações mais completas de L.P.
- A principal desvantagem é que o tempo de execução de um programa interpretado é em geral, maior que o de um correspondente programa objeto compilado.

Aspectos Básicos



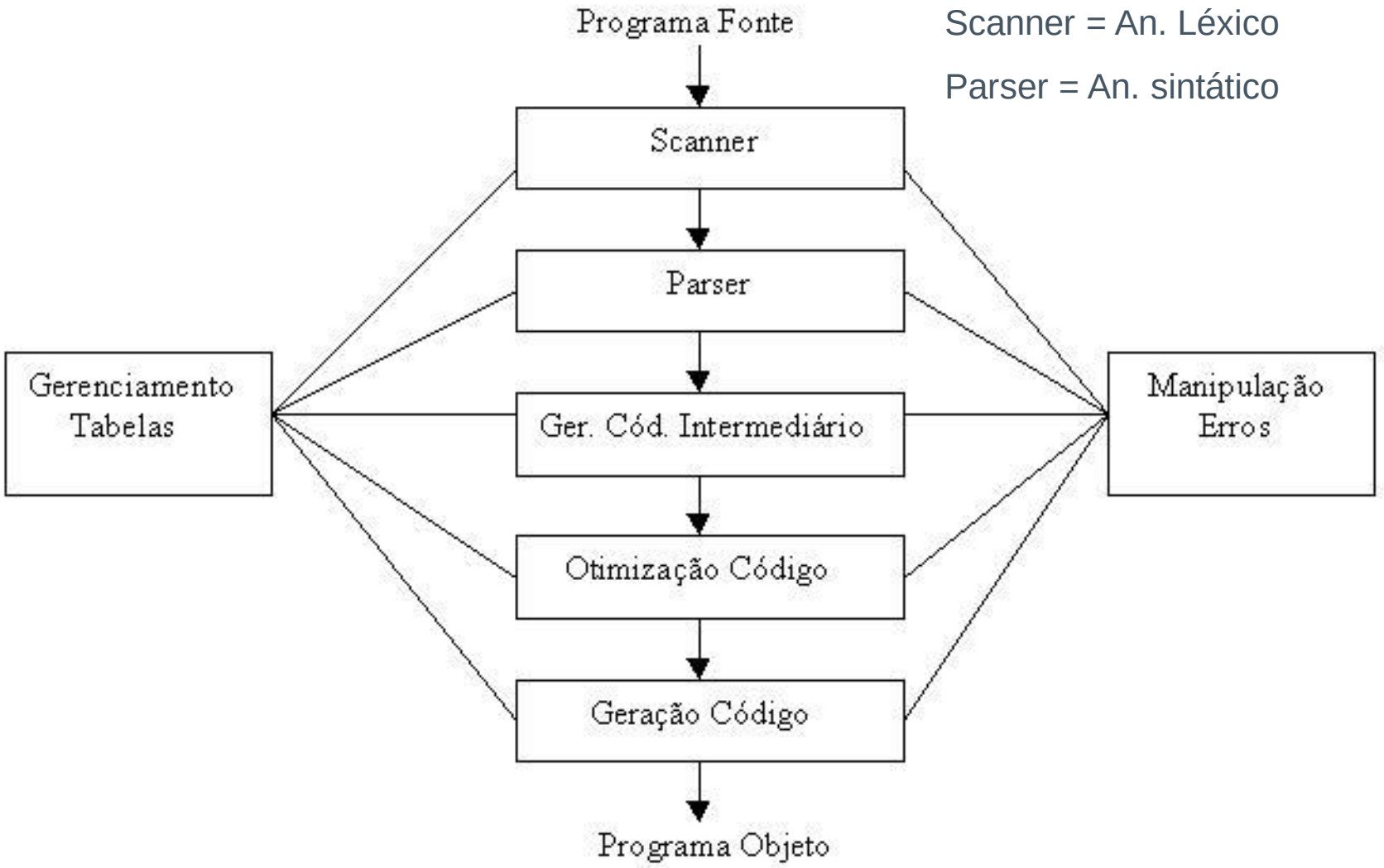
- Programa Fonte: sequência de caracteres que corresponde a uma frase, elaborada de acordo com as regras da linguagem fonte.

- Aspectos da Compilação
  - Aspecto Sintático
  - Aspecto Semântico
  - Aspecto Pragmático

Aspectos Básicos

- Considerações:
  - Aspecto Sintático: há uma formalização conveniente através de Gramáticas Livres de Contexto, que permitem a descrição da linguagem
  - Aspecto Semântico: pouca generalização - inexistência de modelos adequados
  - Aspecto Pragmático: mais variável, apresentando soluções diferenciadas para cada Sistema Operacional adotado.

Diagrama de um Compilador



Scanner = An. Léxico

Parser = An. sintático

Fases de um Compilador

A *Análise Léxica* ou *Scanner* agrupa caracteres da linguagem fonte em grupos chamados **itens léxicos (tokens).** Geralmente, as classes à que pertencem esses itens são:

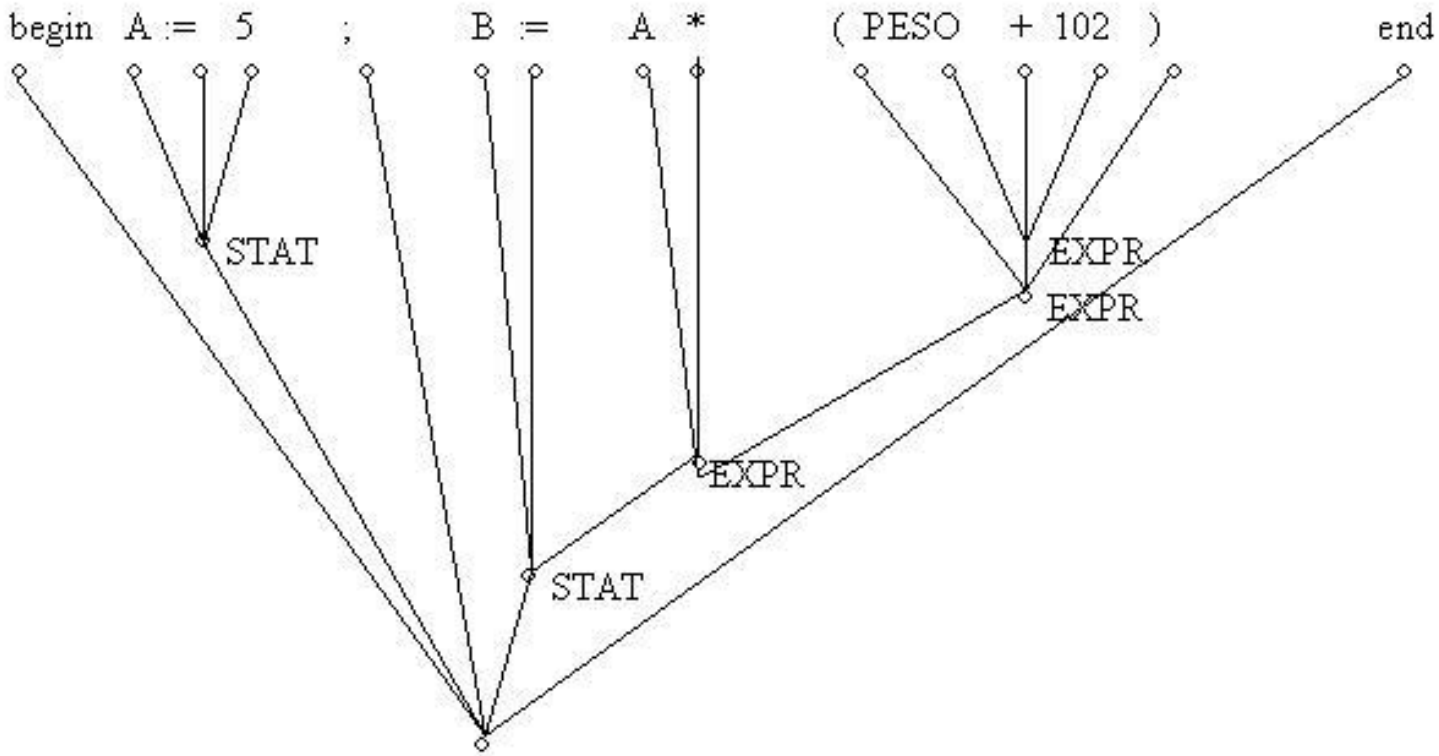
- PALAVRAS RESERVADAS : DO, IF, etc
- IDENTIFICADORES : x, num, etc
- SÍMBOLOS DE OPERADORES : <=, +, etc
- SÍMBOLOS DE PONTUAÇÃO : (,), ; , etc
- NÚMEROS : 1024, 105, etc

Por exemplo, em Pascal:

```
begin
        A := 5 ;
        B := A * ( PESO + 102 )
end
```

Fases de um Compilador

A **Análise Sintática** agrupa os itens léxicos (tokens) em diversas unidades sintáticas, construindo uma **árvore sintática:**



Obs.: A árvore sintática mostra a estrutura gramatical de um programa. Cada um de seus nós representa uma unidade sintática.

Fases de um Compilador

Tipos de Erros da Fase de Análise:

- *Erros léxicos*: O "scanner" deve detectar **erros léxicos** que podem ser, por exemplo, o uso de caracteres não usados pela linguagem, ou nos inteiros com grandeza maior do que a máxima representada no computador;
- *Erros Sintáticos*: O "parser" tem como tarefa o reconhecimento de **erros sintáticos:** construções do programa fonte em desacordo com as regras de formação de unidades sintáticas, como especificado pela gramática.

*Ex.: Na sequência A + * B, deve ser detectado um operador aritmético a mais.*

Obs: Após reconhecer um erro de sintaxe, o analisador deve emitir mensagem de erro adequada, e tratar ("recover") esse erro, isto é, continuar a análise do resto do programa, de forma que o erro comprometa o mínimo possível o processo de análise.

Fases de um Compilador

- **Gerador de código intermediário** usa a estrutura produzida pelo "parser" para criar uma cadeia de instruções simples. Muitos estilos de código intermediário são possíveis. Um estilo comum usa instruções com um operador e um nº pequeno de operandos;

- **Otimização de Código (fase opcional)**: melhorar o código intermediário tal que o programa objeto seja mais rápido e/ou ocupe menos espaço. Sua saída é outro programa em código intermediário que faz a mesma tarefa do original;
- **Gerador de código:** gera o programa objeto. O código é gerado sempre para determinadas unidades sintáticas, sendo utilizadas informações fornecidas pelo analista de contexto.

Fases de um Compilador

- O **gerenciamento de tabelas** ou **"bookkeeping"** é a porção do compilador que manipula os nomes usados pelo programa e registra informações essenciais sobre cada um deles, tal como seu tipo (inteiro, real, etc). A estrutura de dados usada para registrar essa informação é chamada **Tabelas(s) de Símbolos.**

- O **manipulador de erros** é ativado quando uma falha é detectada no programa fonte. Ele avisa o programador, fornecendo um diagnóstico claro e preciso, e torna possível a continuação do processo de análise. É desejável que sejam detectados todos os erros numa única compilação.

Definições Indutivas

Proposição: elaborar uma gramática (conjunto de regras) que permita especificar todas as expressões possíveis com a utilização de *a* e *b* como operandos e '+', '*', '(' e ')' como operadores.

Solução:

1. *a e b são expressões;*
2. *Se a e b são expressões, então as sequências* □ + □ *e* □ * □ *são expressões;*
3. *Se* □ *é uma sequência de símbolos que é uma expressão, então* (□) *é uma expressão*
4. *Expressões são todas e somente as sequências de símbolos obtidas pelas aplicações das regras 1 a 3 um número finito de vezes.*

Definições Indutivas

Uma definição mais concisa seria dada pela segunda solução:

Segunda Solução:

E = conjunto de expressões válidas;
1. *a e b estão em E;*
2. *Se a e b estão em E, então a + b e a * b estão em E;*
3. *Se a está em E, então (a) está em E.*

Obs: aqui aparece a necessidade de formalizar as soluções: necessidade de uma meta-linguagem que permita elaborar as regras da gramática que gera a linguagem desejada.

Forma de Backus Naur

Teríamos:

E ::= a

E ::= b

E ::= E + E

E ::= E * E

E ::= (E)

Ou:

E ::= a
| b
| E + E
| E * E
| (E)

Gramáticas

Uma gramática livre de contexto (GLC) é composta de quatro componentes:

- um conjunto de tokens, conhecidos como símbolos terminais;
- um conjunto de não-terminais;
- um conjunto de produções, usadas para definir as regras particulares à gramática em questão;
- um símbolo de partida para a utilização das regras.

Gramáticas

Formalmente as gramáticas, dispositivos de geração de sentenças das linguagens que definem, podem ser caracterizadas como quádruplas ordenadas:

$$G = (Vn, Vt, P, S)$$

- **Vn** = vocabulário não terminal da gramática G (corresponde ao conjunto de todos os elementos simbólicos dos quais a gramática se vale para definir as leis de formação das sentenças da linguagem)
- **Vt** = vocabulário terminal contendo os símbolos ou átomos dos quais as sentenças da linguagem são constituídas. Dá-se o nome de terminais aos elementos de Vt.

Gramáticas

- **P** = conjunto de todas as leis de formação utilizadas pela gramática para definir a linguagem. Para tanto, cada construção parcial, representada por um não-terminal, é definida como um conjunto de regras de formação relativas à definicão do não-terminal a ela referente. A cada uma destas regras de formação que compõem o conjunto P dá-se o nome de produção da gramática. Cada produção P tem a forma: $\alpha \Rightarrow \beta$ $\alpha \in (Vn \cup Vt)^+$; $\beta \in (Vn \cup Vt)^*$ *onde* α *é, no caso geral, uma cadeia contendo no mínimo um não-terminal*
- **S** é um elemento de Vn, cuja propriedade ser o não-terminal que dá início ao processo de geração de sentenças. S é dito o símbolo inicial ou o axioma da gramática.

Gramáticas

No exemplo dado, teríamos:

Vn = {E}

Vt = {a, b, +, *, (,)}

P = regras já apresentadas para a definição

S = E

Obs: ao conjunto de cadeias geradas pela gramática dá-se o nome de linguagem.

Forma Normal de Chomsky

Toda gramática livre de contexto é equivalente a uma gramática na **Forma Normal de Chomsky***, onde todas as produções podem ser colocadas nas formas:

$$A \rightarrow BC$$
$$A \rightarrow \alpha$$

* *Chomsky introduziu as Gramáticas Livres de Contexto em 1956 como parte de um estudo de linguagens naturais, enquanto a notação para a BNF aparecem em 1964*

Gramáticas Regulares

Gramáticas regulares são gramáticas onde as produções são apenas da forma:

$A \rightarrow \alpha$:Gramática linear à direita

$A \rightarrow \alpha\ A$

ou

$A \rightarrow \alpha$

$A \rightarrow A\ \alpha$:Gramática linear à esquerda

Gramáticas Regulares e GLC

- Toda *gramática regular* é uma *gramática livre de contexto;*
- As produções das *gramáticas regulares* são mais simples do que as produções das *gramáticas livres de contexto;*
- As *gramáticas livres de contexto* permitem a construção de *árvores sintáticas* mais complexas.

$E \rightarrow T \mid E + T \mid E - T$

$T \rightarrow F \mid T * F \mid T / F$

$F \rightarrow i \mid F ** i$

A Gramática Regular equivalente seria:

$E \rightarrow iM \mid i$

$M \rightarrow + E \mid - E \mid * E \mid / E \mid ** E$

Análise do programa fonte

- Análise léxica
  - Organiza caracteres de entrada em grupos, chamados tokens
- Análise sintática
  - Organiza tokens em uma estrutura hierárquica
- Análise semântica
  - Checa se o programa respeita regras básicas de consistência

Análise léxica (scanning)

- Lê os caracteres de entrada e os agrupa em sequências chamadas tokens
- Os tokens são consumidos na fase seguinte (parsing)

Exemplo

```
position = initial + rate * 60
```

<identificador, 1>, <=>,
<identificador, 2>, <+>,
<identificador, 3>, <*>,
<number, 60>

Tabela de
Símbolos

	nome	tipo	
1	position	-	...
2	initial	-	
3	rate	-	
...			

Exemplo

Analisador Léxico

O projetista do compilador caracteriza o analisador léxico através de **expressões regulares (ERs)**.

Qualquer semelhança com Teoria da Computação não é coincidência!!!

Exemplo

Analisador Léxico

A geração do analisador léxico é automática a partir da definição das ERs.
Ver: FLEX, JLex, etc.

Tabela de símbolos

- Estrutura de dados usada para guardar identificadores e informações sobre eles.
- Por exemplo:
  - tipo do identificador
  - escopo: onde o identificador é válido no programa
  - se for um procedimento ou função: número e tipo dos argumentos, forma de passagem dos parâmetros e tipo do resultado.

Tabela de símbolos

	nome	tipo	
1	position	-	...
2	initial	-	
3	rate	-	
...			

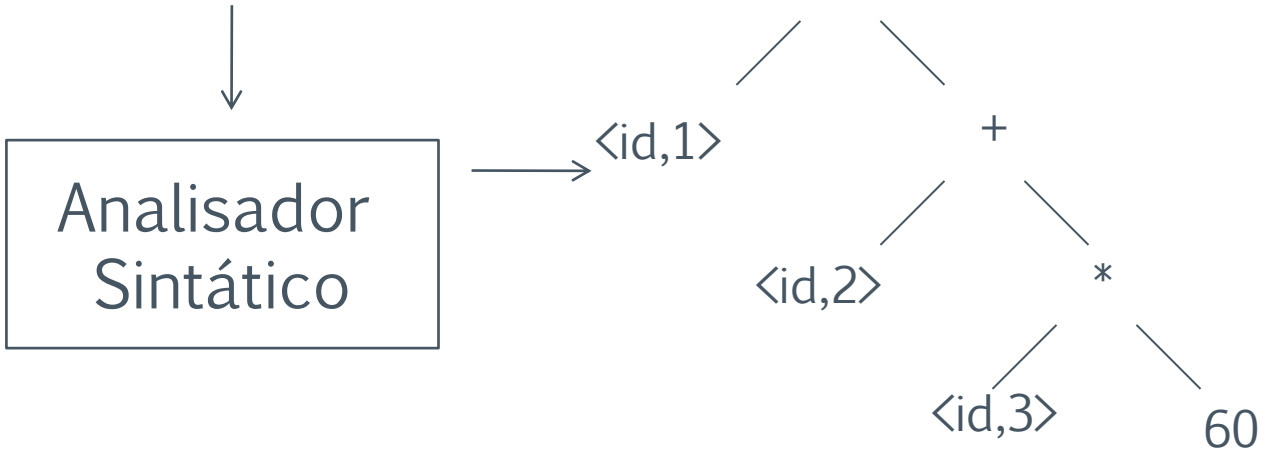
Usada e atualizada em várias etapas da compilação.

Análise sintática (parsing)

- A partir dos *tokens* cria uma estrutura em árvore (*árvore sintática*) que representa a estrutura gramatical do programa

Exemplo

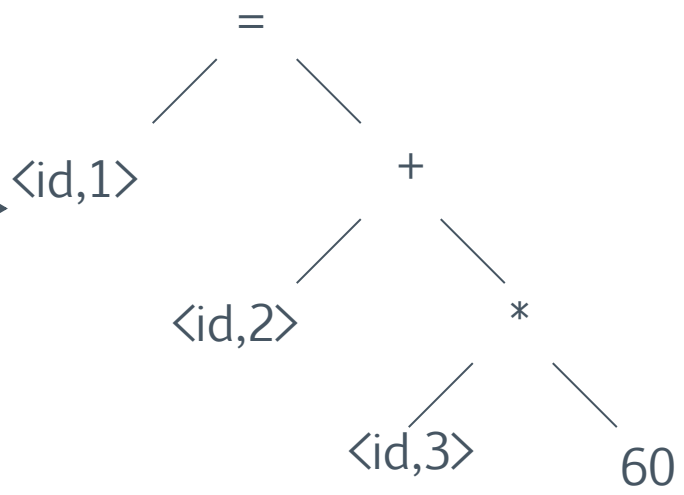
<id,1>, <=>,
<id,2>, <+>,
<id,3>, <*>,
<number, 60>



Exemplo

Gramática livre de contexto (BNF - *Backus-Naur Form*) caracteriza a linguagem.

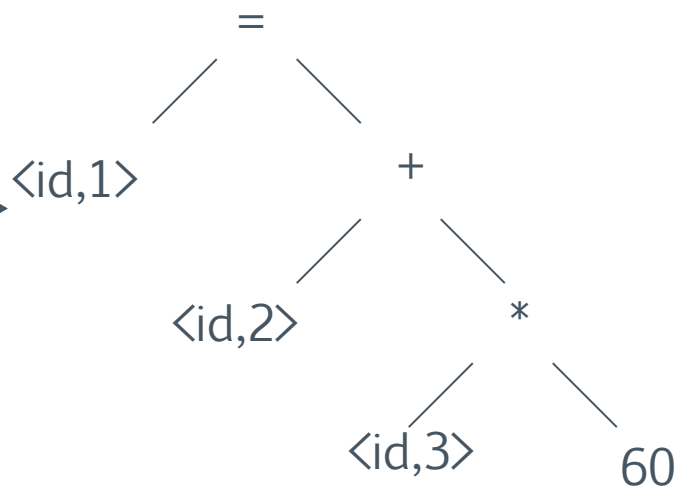
<identificador, 1>, <=>,
<identificador, 2>, <+>,
<identificador, 3>, <*>,
<number, 60>



Exemplo

A geração do parser a partir de uma BNF é automática.

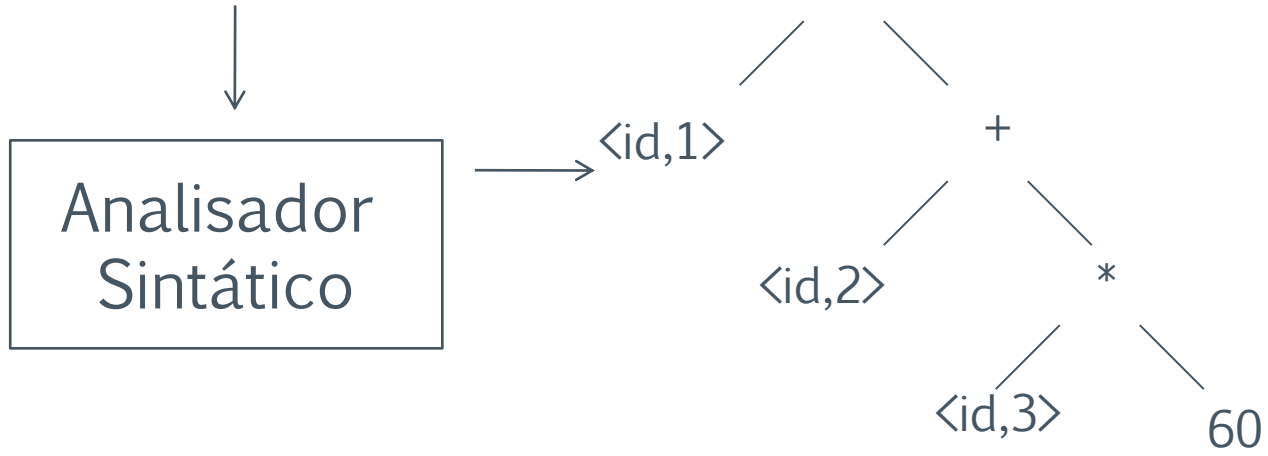
<identificador, 1>, <=>,
<identificador, 2>, <+>,
<identificador, 3>, <*>,
<number, 60>



Exemplo

Para cada classe gramatical da BNF haverá uma estrutura de dados correspondente.

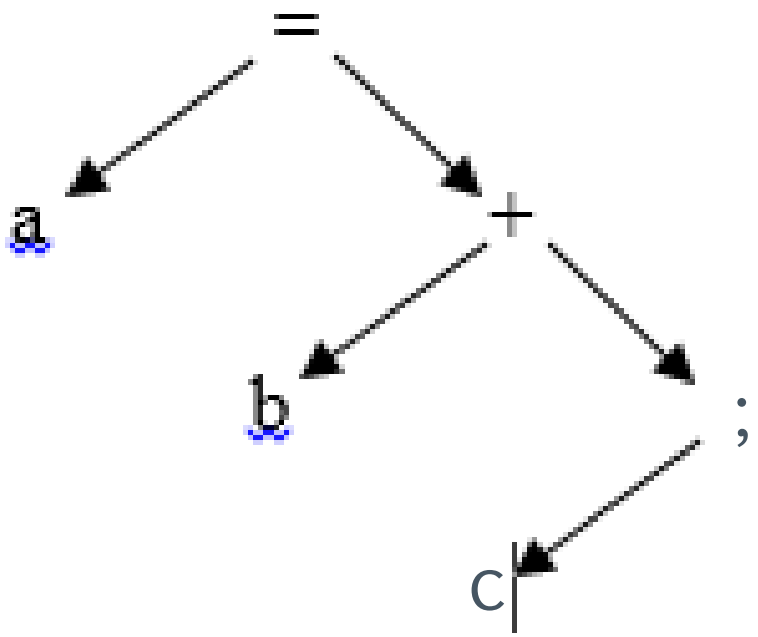
<identificador, 1>, <=>,
<identificador, 2>, <+>,
<identificador, 3>, <*>,
<number, 60>



Exemplo 2

› Ex: a = b + c;

Lexema	Token
a	Identificador
	Caracter em branco
=	Operador de atribuição
	Caracter em branco
b	Identificador
	Caracter em branco
+	Operador Aritmético
	Caracter em branco
c	Identificador
;	Separador de comandos



Árvores de Derivação

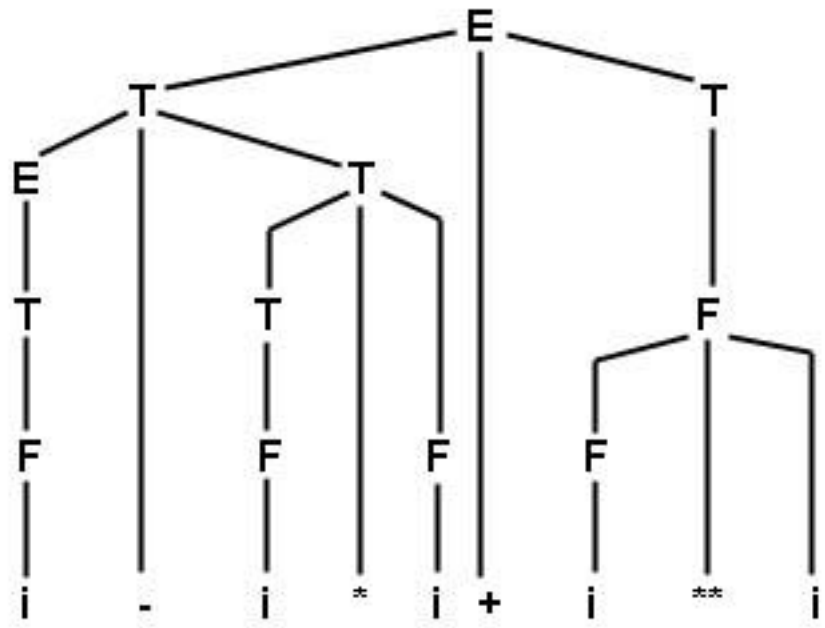
- Árvores: representam, passo a passo a geração de uma cadeia por uma gramática
- Exemplo: Seja a gramática de geração de Expressões:

E → T | E + T | E - T

T → F | T * F | T / F

F → i | F ** i

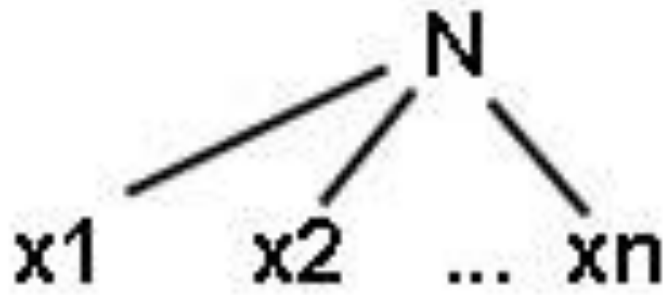
  - árvore de derivação:



Árvores de Derivação

Dada uma *gramática livre de contexto* :

- 1. A árvore constituída inicialmente pelo nó rotulado S é uma árvore sintática.
- 2. A árvore obtida pela substituição de uma folha (nó final), rotulada N, de uma árvore sintática, pela árvore



onde N → x1, x2, ..., xn pertencentes a P é uma árvore sintática.

Árvores de Derivação

Nas árvores de derivação:

- a raiz da árvore é o símbolo de partida;
- as folhas da árvore são símbolos terminais (tokens), na seqüência em que ocorrem na sentença analisada; e
- os nós intermediários da árvore correspondem a símbolos não terminais, onde um nó cujo rótulo é *P* com filhos *s1, s2, s3, ..., sn* pode ocorrer apenas se houver uma regra:

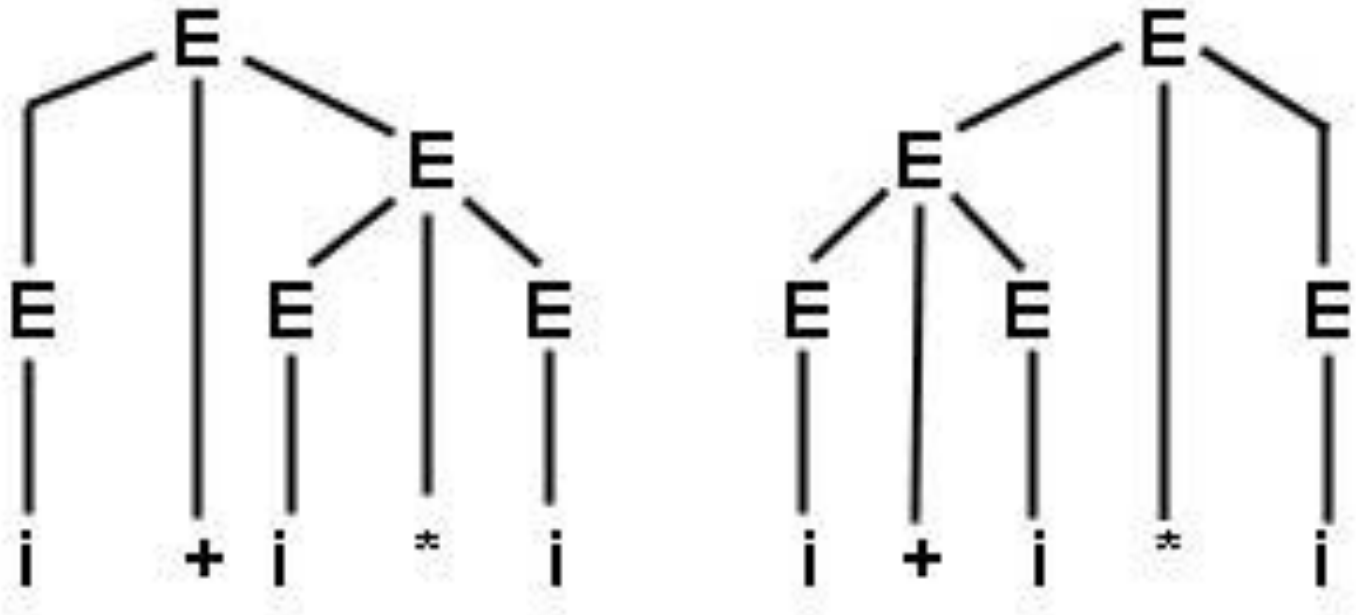
$$P \rightarrow s_1\ s_2 \ldots s_n$$

na gramática.

Ambigüidade

Uma gramática é ambígua se, por duas árvores sintáticas diferentes, pode-se obter a mesma sentença.

Exemplo: supondo as produções: $E \rightarrow E + E \mid E * E \mid i$



Associatividade de operadores

Na maioria das linguagens de programação, os operadores aritméticos +, -, * e / são associativos à esquerda.

Assim: "3 + 4 * 5" equivale a "(3 + 4) * 5".

Quando um operando, como 4, estiver entre dois operadores de mesma precedência (neste caso, + e -), ele será associado ao operador mais à esquerda, neste caso, +. Por isto dizemos que os operadores aritméticos são associados à esquerda.

A gramática

Expr ::= Expr + Term | Expr - Term | Term

Term ::= Term * Numero | Termo / Numero | Numero

com terminais +, -, * , / e Numero, associa todos os operadores à esquerda.

Derivações Canônicas

Considerando expressões simples como a do exemplo:

- há diversas opções para a sequência de aplicação das regras que poderiam levar ou não ao reconhecimento da sentença;
- tal fato dificultaria a automação desse processo.
- é importante ter formas sistemáticas de aplicações dessas regras que levem a uma conclusão sobre a validade ou não da sentença.
- *Essa forma sistemática de aplicação das regras de uma gramática é estabelecida através das derivações canônicas.*
- Duas formas de derivação canônica são estabelecidas:
  - mais à esquerda
  - mais à direita.

Derivações Canônicas

1. **Derivação mais à esquerda** (*leftmost derivation*): a opção é aplicar uma regra da gramática ao símbolo não-terminal mais à esquerda da forma sentencial sendo analisada. A correspondente seqüência de regras aplicadas é denominada a **seqüência de reconhecimento mais à esquerda**, ou *leftmost parse*.

2. **Derivação mais à direita** (*rightmost derivation*) o símbolo não-terminal mais à direita é sempre selecionado para ser substituído usando alguma regra da gramática. No caso desse tipo de derivação, a **seqüência de reconhecimento mais à direita**, ou *rightmost parse*, é o reverso da seqüência de regras associada à derivação.

Exemplo

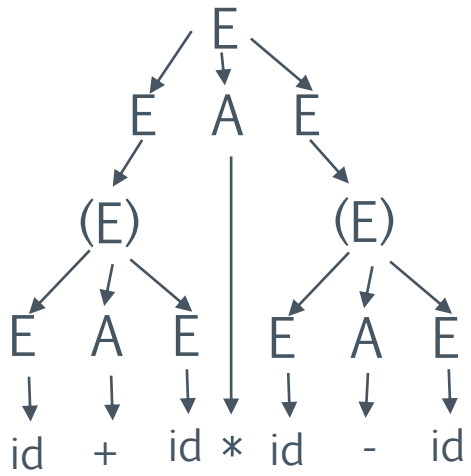
Usando a gramática:

E → E A E|(E)|-E| id

A → +|-|*|/

Construa uma árvore de derivação de:

(id + id) * (id – id)



Dúvidas

José Osvano da Silva
joseosvano@unipac.br