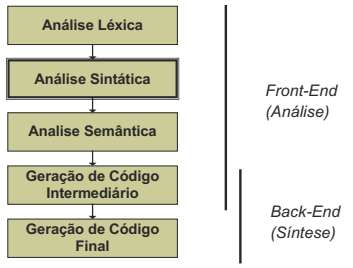
Análise Sintática (parte 1)

Etapas da Compilação



Análise Sintática

- Alguns autores consideram que **análise sintática** se refere a tudo que diz respeito à verificação do formato do código fonte
  - Inclui a análise léxica como subfase
  - Visão mais coerente, porém o livro que usamos tem outra visão...

Análise Sintática

- Entenderemos **análise sintática** como sendo apenas a segunda fase dessa verificação de formato:
  - Fase que analisa a seqüência de tokens para descobrir a "estrutura gramatical" do código fonte
  - Também chamada "Reconhecimento" (***Parsing***)
  - Nome melhor: "Análise Gramatical" ?

Sumário da Aula

- Gramáticas Livres de Contexto
- Introdução à Análise Sintática

Gramáticas Livres de Contexto

Gramáticas

- São usadas para organizar os tokens em "sequências de tokens"
  - Análogas a "frases"
- Definem regras de formação recursivas

```
expressão → CTE_INT
          | ID
          | expressão + expressão
```

Notações

- **Notação comum nos livros de Teoria da Computação**
  - Símbolo não-terminal: letras maiúscula
  - Símbolo terminal: letras minúsculas, sinais, etc.

```
E → T
  | T + E
T → x
  | i
```

Notações

- **Notação BNF** (Backus-Naur Form)
  - Símbolo não-terminal delimitado por "<" e ">"

```
<expressão> ::= <termo>
              | <termo> "+" <expressão>

<termo> ::= "x"
          | "i"
```

Notações

- **Notações EBNF** (extended BNF)
  - Nome genérico – existem várias!
  - Não-terminais sem delimitadores
  - BNF + operadores de expressões regulares

```
expressão = termo {"+" termo}*

termo = "x"
      | "i"
```

Notações

- Notações que usaremos
  - **Tradicional**: para mostrar conceitos mais teóricos
  - **EBNF**: para os exemplos práticos
    - Tokens aparecerão literalmente se forem simples (ex.: sinais, palavras-chave, etc.)
    - Tokens com várias opções de casamento aparecerão em maiúscula (ex.: identificadores, valores literais)

Gramáticas

- No estudo de Teoria da Computação, a gramática podem ser chamada de formalismo **gerador**
- A partir do símbolo inicial, podemos **derivar** (gerar) as *cadeias* (sequências) de terminais que são válidas na linguagem

Derivação

- O processo de derivação consiste em substituir cada ocorrência de um **não-terminal** pelo lado direito (corpo) de alguma de suas produções
  - **Derivação mais à esquerda**: o não-terminal a ser substituído é sempre o mais à esquerda
  - **Derivação mais à direita**: análogo
- A derivação pára quando sobrarem apenas terminais
  - A cadeia resultante fica nas folhas

Derivação

- Seja a gramática BNF anterior
  - Derivar a cadeia "i+i+x"

```
  <expressão>
⇒ <termo> + <expressão>
⇒ i       + <expressão>
⇒ i       + <termo> + <expressão>
⇒ i       + i       + <expressão>
⇒ i       + i       + <termo>
⇒ i       + i       + x
```

  - Esta é uma derivação mais à esquerda

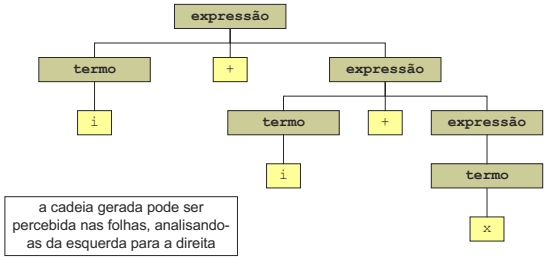
Derivação

- Seja a gramática BNF mostrada antes
  - Derivação mais à direita da cadeia "i+i+x"

```
  <expressão>
⇒ <termo> + <expressão>
⇒ <termo> + <termo> + <expressão>
⇒ <termo> + <termo> + <termo>
⇒ <termo> + <termo> + x
⇒ <termo> + i       + x
⇒ i       + i       + x
```

Árvore de Derivação

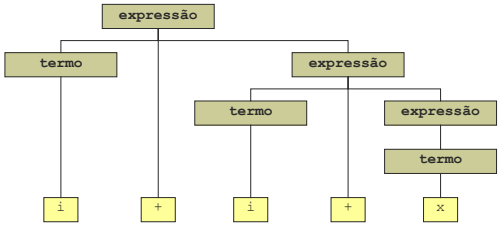
- A árvore dos exemplos anteriores



a cadeia gerada pode ser percebida nas folhas, analisando-as da esquerda para a direita

Árvore de Derivação

- A mesma árvore reorganizada



a seqüência de terminais (cadeia) gerada

Árvore de Derivação

- Mostra de maneira estática as produções aplicadas
  - Não diferencia se foi uma derivação mais à esquerda ou mais à direita que gerou a cadeia
- Forma
  - Tem o símbolo inicial como raiz
  - Não-terminais formam nós intermediários
  - As folhas são os terminais (tokens) da cadeia

Características de Gramáticas

- Gramática fatorada à esquerda:
  - GLC que **não** possui produções do tipo A -> $\alpha\beta_1|\alpha\beta_2$ para alguma forma sentencial de α.
- Gramática recursiva à esquerda:
  - GLC que permite derivação

OBS:
RECONHECEDOR TOP-DOWN não aceita gramáticas recursivas à esquerda
Exemplo:

E -> E+T|T
T -> T*F|F
F -> (E)|Id

Eliminação da recursividade à esquerda

- Exemplo:
  - A -> Aa | b

Com palavra vazia	Sem palavra vazia
A -> bX X -> aX | ε	A -> b | bX X -> a | aX

Fatoração de uma gramática

- Elimina indecisão de qual produção aplicar quando duas ou mais produções iniciam com a mesma forma sentencial:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Se torna:

$$A \rightarrow \alpha X$$
$$X \rightarrow \beta_1 \mid \beta_2$$

Exemplo de fatoração à esquerda

- Exemplo:

Cmd → if Expr then Cmd else Cmd
Cmd → if Expr then Cmd
Cmd → Outro

Fatorando a esquerda:

Cmd → if Expr then Cmd ElseOpc
Cmd → Outro
ElseOpc → else Cmd | ε

Análise Sintática

- Vimos que gramáticas são formalismos que geram cadeias
- Em compiladores, não vamos gerar uma cadeia, mas já temos a cadeia de terminais (ou seja, de tokens) pronta...
- Diante disso, qual seria a função do analisador sintático?

Introdução à Análise Sintática

Análise Sintática

- O objetivo
  - Descobrir como uma seqüência de tokens pode ser gerada pela gramática da linguagem
- Em outras palavras
  - Entrada: seqüência de tokens
  - Saída: árvore

Análise Sintática

- O módulo de software responsável por essa etapa pode ser chamado de
  - Analisador sintático ou parser (reconhecedor)
- Existem duas estratégias algorítmicas que podem ser adotadas
  - **Bottom-up** ou Ascendente
  - **Top-down** ou Descendente

Análise Sintática

- Usaremos a seguinte gramática não-ambígua para ilustrar, a seguir, as duas estratégias de análise sintática

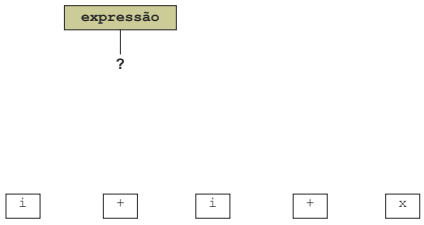
```
<expressão> ::= <termo>
              | <termo> "+" <expressão>

<termo> ::= "x"
          | "i"
```

Análise Descendente

- Parte da raiz (não-terminal inicial) e tenta criar a árvore de cima para baixo
- Para cada não-terminal, tenta adequar a cadeia de tokens (lida na entrada) a uma de suas produções
  - O desafio é escolher a produção adequada...
- Se a produção tiver não-terminais, faz um processo similar para aquele não-terminal
- O processo de construção da árvore lembra uma derivação mais à esquerda da cadeia

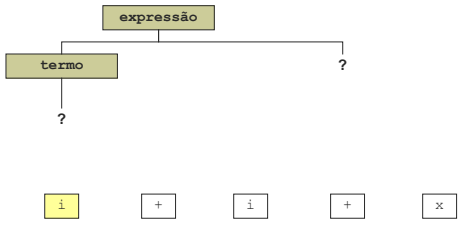
Análise Descendente

- O parser vai ter que escolher uma produção adequada para o símbolo inicial <expressão>



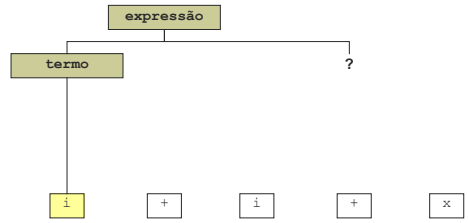
Análise Descendente

- Sabe-se que ambas as produções iniciam com <termo>, variando o restante



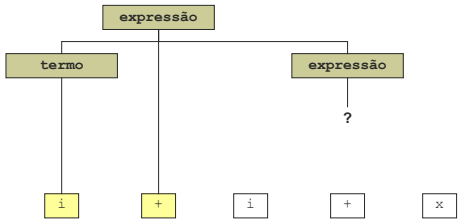
Análise Descendente

- Ao ler o token “i”, o parser identifica que é gerado por <termo>, mas falta decidir o resto...



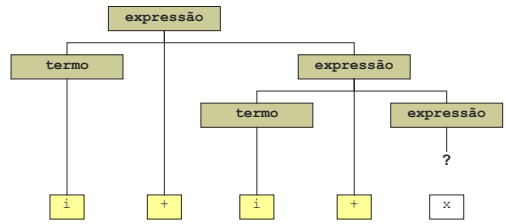
Análise Descendente

- A descoberta do “+” faz o parser decidir a produção adequada



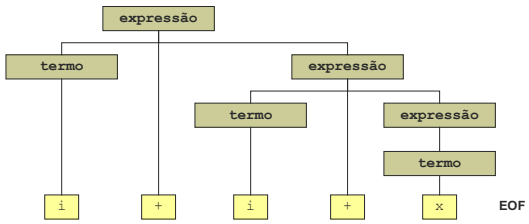
Análise Descendente

- Processo similar acontece após a leitura dos dois tokens seguintes



Análise Descendente

- Porém, no último token, <expressão> vai ter apenas <termo> e o parser encerra no EOF (fim de arquivo)



Análise Descendente

- Veremos na próxima aula como construir um parser desse tipo com relativa facilidade
- Agora, vamos ver como funciona a estratégia ascendente (de baixo para cima)

Análise Ascendente

- Parte das folhas (tokens) e tenta crescer a árvore até a raiz (símbolo inicial)
- Para isso, compara a seqüência de símbolos o lado direito (ou corpo) das produções para tentar criar um ramo da árvore
- Diz-se que a seqüência é “reduzida” ao não-terminal do lado esquerdo da produção

Análise Ascendente

- Tenta crescer a árvore usando o corpo das produções, até chegar ao símbolo inicial

Análise Ascendente

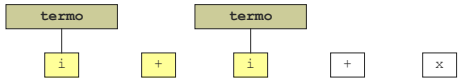
- O token "i" é reduzido ao não-terminal <termo>, devido à produção <termo> ::= "i"

Análise Ascendente

- O parser lê o token "+", mas ainda não pode fazer uma redução

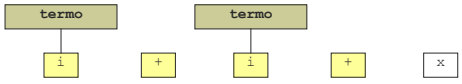
Análise Ascendente

- Mais uma redução ao não-terminal <termo>



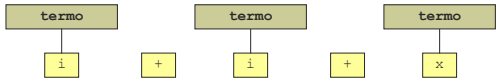
Análise Ascendente

- Apenas continua a leitura



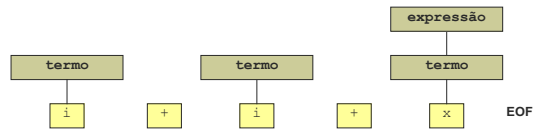
Análise Ascendente

- Nova redução ao não-terminal <termo>



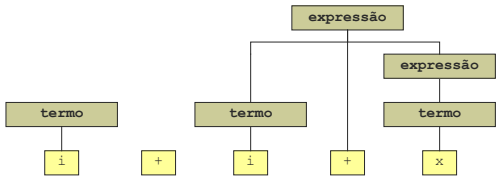
Análise Ascendente

- Como acabaram-se os tokens, o último <termo> só pode ser gerado diretamente por <expressão>, então reduz



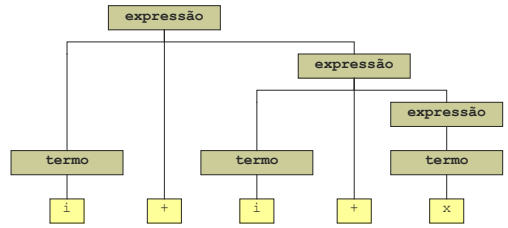
Análise Ascendente

- Agora, a subcadeia "<termo>+<expressão>" pode ser reduzida para <expressão>



Análise Ascendente

- Outra redução, usando a mesma produção



Descendente x Ascendente

- A análise sintática descendente é mais fácil de entender e de implementar, porém a ascendente é mais poderosa (aplicável em mais linguagens)
- Uso principal das duas estratégias
  - **Ascendente**: geralmente, usado apenas em geradores semi-atomáticos de parsers
  - **Descendente**: tem *uma* técnica simples de implementar manualmente (e outras usadas em geradores também)

Descendente x Ascendente

- Chama-se **gramática LL** àquela que permite a construção de um parser descendente para reconhecê-la
- Chama-se **gramática LR** àquela que permite a construção de um parser ascendente para reconhecê-la

Análise Sintática

- Além de construir a árvore, outras atribuições importantes do analisador sintático são:
  - **Reportar erros**, o que deve ser feito de maneira clara para permitir ao usuário corrigir o problema
  - **Recuperar-se de erros** automaticamente (pouco uso em compiladores comerciais)

Análise Sintática

- Na verdade, a análise sintática não precisa construir a árvore durante o reconhecimento
- Só é necessário construir a árvore se ela for realmente separada das etapas seguintes
  - Em “passagens” distintas
  - **Passagem** – percorrer todo o código fonte
- Em todo caso, o parser vai funcionar descobrindo como a árvore poderia ser construída, na gramática dada,para os tokens dados