Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Análise Sintática**
  * Introdução
  * Gramáticas Livres de Contexto
  * Análise descendente e ascendente
  * Analisadores LL e LR

Introdução

* Iremos nos dedicar agora aos métodos de reconhecimento sintático tipicamente usados nos compiladores.

* Como os programas podem conter erros sintáticos, discutiremos também as extensões propostas aos métodos de análise para a recuperação dos erros mais comuns.

* A discussão sobre tratamento de erro é importante pois é desejável que o analisador sintático responda corretamente ao descobrir que a sua entrada não pode ser sintaticamente reconhecida de acordo com a gramática dada.

Introdução

* Por definição, as linguagens de programação possuem regras precisas para descrever a estrutura sintática de programas bem formados.

* Em C, por exemplo, um programa é composto de funções, uma função é composta de declarações e comandos, um comando é formado a partir de expressões, e assim por diante.

Introdução

* Examinaremos como o analisador sintático se integra às demais fases de um compilador.

* Depois, examinaremos as gramáticas típicas para representar expressões aritméticas.

* Essas gramáticas são suficientes para ilustrar a essência da análise sintática, pois as técnicas de análise para expressões se aplicam à maioria das construções de linguagens de programação.

Introdução

* A análise sintática, ou parsing, é o núcleo principal do front-end de um compilador.

* O analisador sintático (ou *parser*) é o responsável pelo controle de fluxo do compilador através do código fonte em compilação.

* É ele quem ativa o léxico para que este lhe retorne tokens que façam sentido dentro da gramática especificada pela linguagem e também ativa a análise semântica e a geração de código.

Introdução

* A análise sintática é um processo que verifica se uma determinada entrada (sentença) pertence à uma linguagem gerada por uma gramática.

  * Seja $G1$ uma gramática;
  * Seja $L(G1)$ a linguagem definida por $G1$;
  * Seja α uma sentença de entrada.

  * Então, formalmente, um analisador sintático é uma ferramenta capaz de dizer se: $\alpha \in L(G1)$

Introdução

* A análise sintática é o processo para determinar como uma cadeia de terminais pode ser gerada por uma gramática.

* Na discussão desse problema, é conveniente pensarmos em uma árvore de derivação sendo construída, embora, na prática, um compilador não precise construí-la.

* No entanto, um analisador sintático precisa, em princípio, ser capaz de construir a árvore, do contrário a tradução não terá garantias de ser correta.

Introdução

* Para qualquer gramática livre de contexto, existe um analisador que gasta no máximo $O(n^3)$ para analisar uma cadeia de n terminais. Mas o tempo cúbico geralmente é muito dispendioso.

* Felizmente, para as linguagens de programação reais, em geral podemos projetar uma gramática que pode ser analisada rapidamente.

* Os algoritmos de tempo linear são suficientes para analisar basicamente todas as linguagens usadas na prática.

Introdução

* Os reconhecedores sintáticos de linguagem de programação quase sempre fazem uma única leitura da esquerda para a direita sobre a entrada, examinando à frente um símbolo terminal de cada vez, e construindo partes da árvore de derivação enquanto prosseguem.

* O analisador sintático agrupa os tokens em frases gramaticais usadas pelo compilador com o objetivo de criar uma saída que representa a estrutura hierarquia do programa fonte.

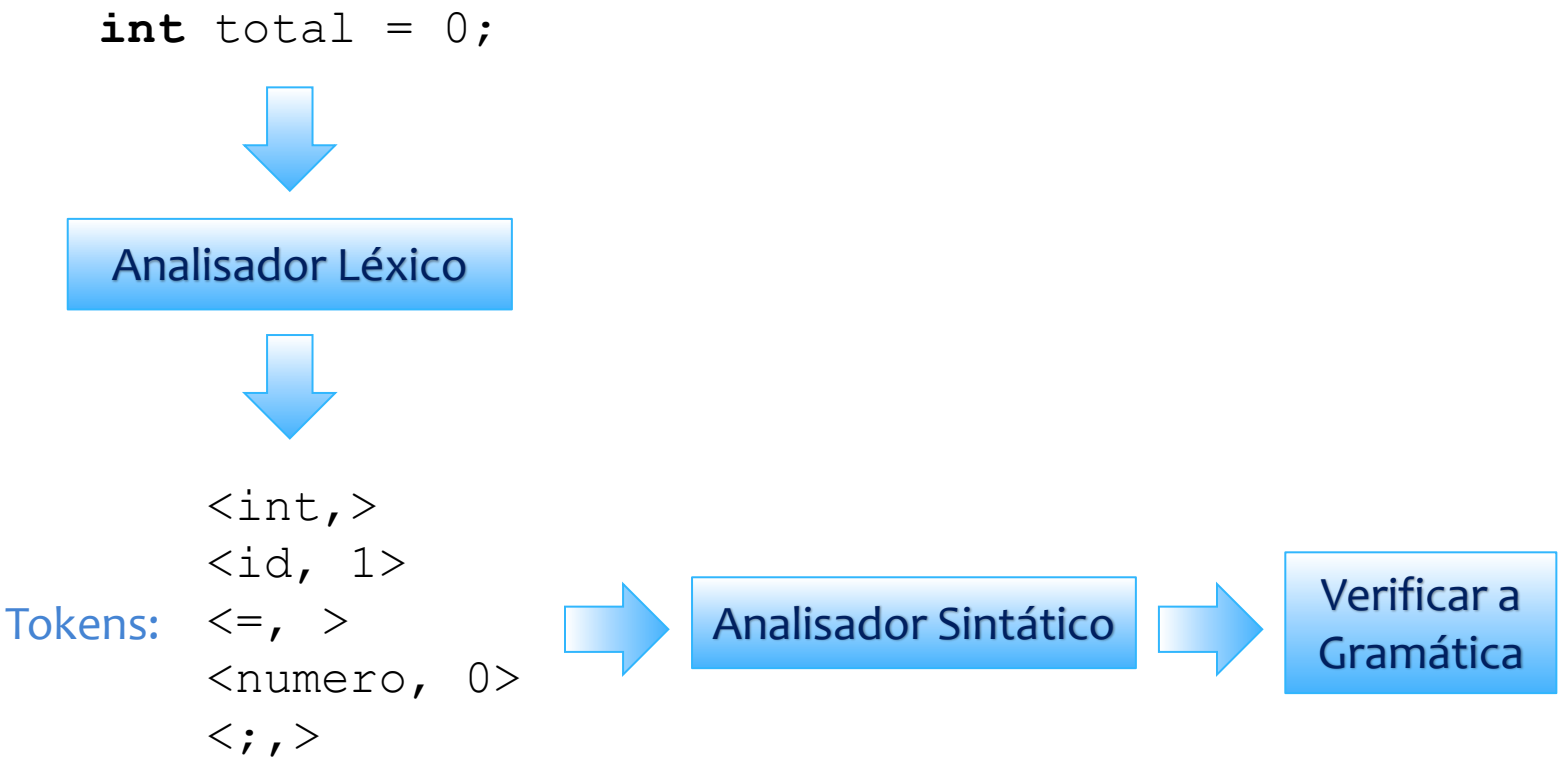
Introdução

* Veja no quadro abaixo as especificações de entrada e saída das etapas vistas até o momento:

Fase	Entrada	Saída
Lexer	Conjunto de caracteres	Conjunto de tokens
Parser	Conjunto de tokens	Árvore sintática

Introdução

* O diagrama abaixo demonstra o processo de compilação visto das etapas estudadas até o momento:



Introdução

* O objetivo do analisador sintático é encontrar os erros *sintáticos*, também chamados de erros estruturais, do código fonte.

* Veja o exemplo abaixo:

```
01  private static Integer maior(Integer numero01 Integer numero02) {
02          if (numero01 > numero02) {
03                  return numero01
04          } else {
05                  return numero02;
06
07  }
```

* Quantos erros sintáticos estão presentes no programa acima?

Introdução

* O objetivo do analisador sintático é encontrar os erros *sintáticos*, também chamados de erros estruturais, do código fonte.

* Veja o exemplo abaixo:

```
01  private static Integer maior(Integer numero01 Integer numero02) {
02          if (numero01 > numero02) {
03                  return numero01
04          } else {
05                  return numero02;
06
07  }
```

* Na linha 06 falta a chaves } para fechar o bloco do else.
* Na linha 03 falta o ponto e vírgula marcando o final do comando.
* Na linha 01 falta a vírgula separando os parâmetros.

Introdução

* Pode parecer simples reconhecer erros estruturais, porém os métodos de análise sintática são complexos, trabalhosos e difíceis de serem implementados à mão.

* Muitas vezes geradores de analisadores sintáticos são utilizados para evitar que se codifique a análise sintática manualmente.

Introdução

* As maiorias dos métodos de análise sintática estão em uma de duas classes, chamadas métodos ***descendentes*** e ***ascendentes***.

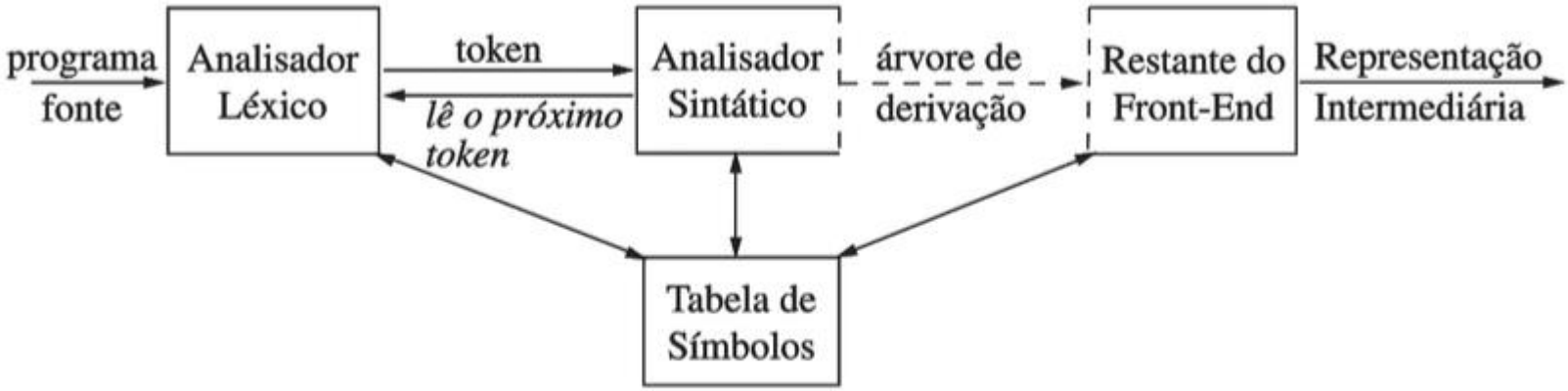
* Esses termos se referem à ordem em que os nós na árvore de derivação são construídos. Nos analisadores ***descendentes***, a árvore é construída de cima para baixo, ou seja, da raiz para as folhas.

* A popularidade dos analisadores descendentes se deve ao fato de que analisadores eficientes podem ser construídos mais facilmente à mão usando esses métodos.

Introdução

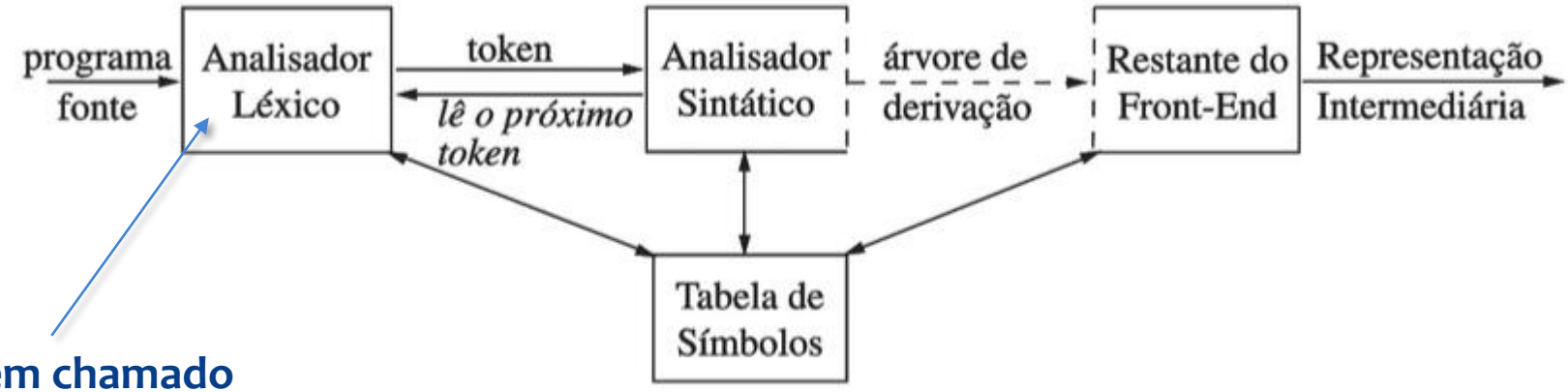
* Contudo, a análise ascendente pode tratar de uma classe maior de gramáticas e esquemas de tradução, de modo que as ferramentas de software para gerar analisadores diretamente a partir de gramáticas normalmente utilizam métodos ascendentes.

* **O Papel do Analisador Sintático**
  * Em nosso modelo de compilador, o analisador sintático recebe do analisador léxico uma cadeia de tokens representando o programa fonte, como mostra a Fig. 1 abaixo, e verifica se essa cadeia de tokens pertence à linguagem gerada pela gramática.



Posição do analisador sintático no modelo de compilador.

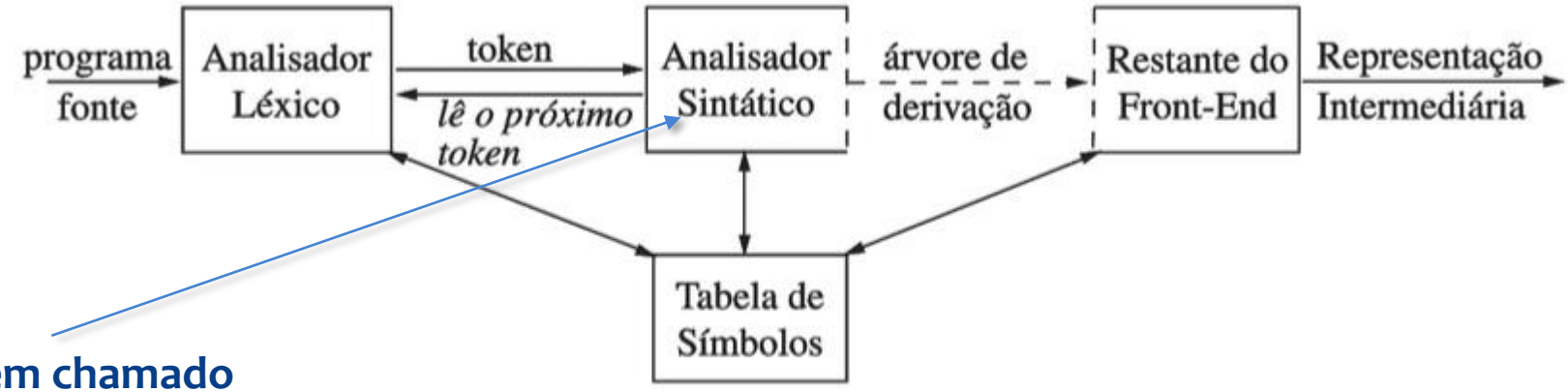
* **O Papel do Analisador Sintático**
  * Em nosso modelo de compilador, o analisador sintático recebe do analisador léxico uma cadeia de tokens representando o programa fonte, como mostra a Fig. 1 abaixo, e verifica se essa cadeia de tokens pertence à linguagem gerada pela gramática.



Posição do analisador sintático no modelo de compilador.

Também chamado de Scanner

* **O Papel do Analisador Sintático**
  * Em nosso modelo de compilador, o analisador sintático recebe do analisador léxico uma cadeia de tokens representando o programa fonte, como mostra a Fig. 1 abaixo, e verifica se essa cadeia de tokens pertence à linguagem gerada pela gramática.



Posição do analisador sintático no modelo de compilador.

Também chamado de Parser

Introdução

* **O Papel do Analisador Sintático**
  * O analisador deve ser projetado para emitir mensagens para quaisquer erros de sintaxe encontrados no programa fonte em uma forma inteligível e também para se recuperar desses erros, a fim de continuar processando o restante do programa.
  * Conceitualmente, para programas bem formados, o analisador sintático constrói uma árvore de derivação e a passa ao restante do *front-end* do compilador para mais processamento.

Introdução

* **O Papel do Analisador Sintático**
  * Na prática, não é necessário construir a árvore de derivação explicitamente, pois as ações de verificação e tradução podem ser entremeadas com a análise.
  * Assim, o analisador sintático e o restante do *front-end* podem ser implementados em um único módulo.

Introdução

* **O Papel do Analisador Sintático**
  * Existem três estratégias gerais de análise sintática para o processamento de gramáticas: universal, descendente e ascendente.

  * Os métodos de análise baseados na estratégia universal como o algoritmo *Cocke-Younger-Kasami* e o algoritmo de *Earley* podem analisar qualquer gramática (Sugestão de pesquisa). No entanto, esses métodos são muitos ineficientes para serem usados em compiladores de produção.

Introdução

* **O Papel do Analisador Sintático**
  * Os métodos geralmente usados em compiladores são baseados nas estratégias descendente ou ascendente.
  * Os métodos de análise descendentes constroem as árvores de derivação de cima (raiz) para baixo (folhas), enquanto os ascendentes fazem a análise no sentido inverso, começam nas folhas e avançam até a raiz construindo a árvore.
  * Em ambas as estratégias, a entrada do analisador sintático é consumida da esquerda para a direita, um símbolo de cada vez.

Introdução

* **O Papel do Analisador Sintático**
  * Os métodos ascendentes e descendentes mais eficientes funcionam apenas para subclasses de gramáticas, mas várias dessas subclasses, particularmente as gramáticas LL e LR, são expressivas o suficiente para descrever a maioria das construções sintáticas das linguagens de programação modernas.

  * Os analisadores implementados à mão geralmente utilizam gramáticas LL. Os analisadores sintáticos para a maior classe de gramáticas LR geralmente são construídos usando ferramentas automatizadas.

Introdução

* **O Papel do Analisador Sintático**
  * Iremos considerar que a saída do analisador sintático é alguma representação da árvore de derivação para a cadeia de tokens reconhecidos pelo analisador léxico.

  * Na prática, existem diversas tarefas que poderiam ser conduzidas durante a análise sintática, como a coleta de informações sobre os tokens na tabela de símbolos, a realização da verificação de tipo e outros tipos de análise semântica, e a geração de código intermediário.

  * Todas essas atividades foram reunidas na caixa denominada "*Restante do Front-End*", da Fig. 1 anterior. Veremos essas atividades em detalhes nas AULAS 05 e 06.

Introdução

* **Análise de Descida Recursiva**
  * *Análise de descida recursiva* é um método de análise sintática descendente em que um conjunto de procedimentos recursivos é usado para processar a entrada. Um procedimento é associado a cada não-terminal de uma gramática.
  * A sequência de chamadas de procedimento durante a análise de uma cadeia de entrada define implicitamente a árvore de derivação para a entrada e pode ser usada para construir uma árvore de derivação explícita, se desejado.
  * Veremos este método em detalhes em "Análise Descendente".

Introdução

* **Recursão à Esquerda**
  * É possível que um analisador de descida recursiva fique em um *loop* para sempre.
  * Este problema surge devido às produções "recursivas à esquerda" do tipo:

$$expr \rightarrow expr + term$$

  * Onde o símbolo mais à esquerda do lado direito da produção é igual ao não-terminal do lado esquerdo da produção.

Introdução

* **Recursão à Esquerda**
  * Suponha que o procedimento para *expr* decida aplicar essa produção.
  * O corpo começa com *expr*, de modo que o procedimento para *expr* é chamado recursivamente.
  * Como o símbolo *lookahead* (terminal corrente sendo lido na entrada) muda apenas quando um terminal no corpo é casado, nenhuma mudança na entrada ocorre entre as chamadas recursivas de *expr*.

Introdução

* **Recursão à Esquerda**
  * Como resultado, a segunda chamada a *expr* faz exatamente o que a primeira chamada faz, o que significa uma terceira chamada a *expr*, e assim por diante, indefinidamente.

  * Uma produção recursiva à esquerda **pode ser eliminada pela reescrita da produção problemática**.

Introdução

- **Recursão à Esquerda**
  - Considere um não-terminal A com duas produções:

$$A \rightarrow A\alpha \mid \beta$$

  - Onde α, β são sequências de terminais e não-terminais que não começam com A. Por exemplo em:

$$\text{expr} \rightarrow \textit{expr} \ + \ \textit{term} \mid \textit{term}$$

  - O não-terminal A = *expr*, α = *+ term*, *e* β = *term*.

  - O não-terminal A e sua produção são considerados *recursivos à esquerda*, pois a produção $A \rightarrow A\alpha$ tem o próprio A como símbolo mais à esquerda no lado direito.

Introdução

* **Recursão à Esquerda**
  * Em uma gramática recursiva à esquerda mais **geral**, em vez de uma produção $A \rightarrow A\alpha$, o não-terminal A pode derivar $A\alpha$ por meio de produções intermediárias.
  * A aplicação repetida dessa produção acumula uma sequência de αs à direita de A, como na Fig. 2 (a) a seguir.
  * Quando A é finalmente substituído por β, temos um β seguido por uma sequência de zero ou mais αs.

Introdução

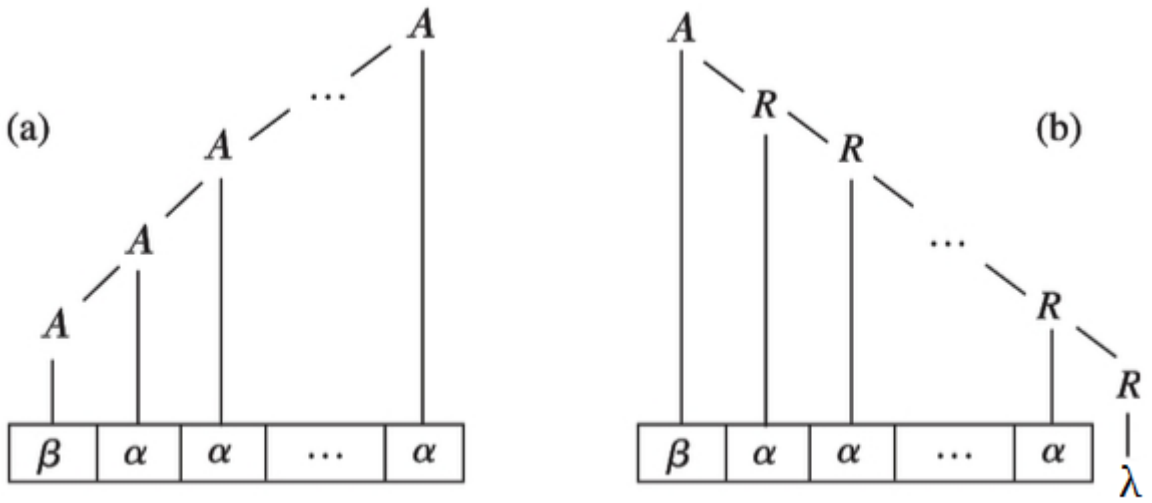
* **Recursão à Esquerda**
  * O mesmo efeito pode ser conseguido, como na Fig. 2 (b), reescrevendo-se as produções para A, usando um novo não terminal R da seguinte maneira:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \lambda \end{aligned}$$

  * O não-terminal R e sua produção $R \rightarrow \alpha R$ são recursivos à direita, pois essa produção para R tem o próprio R como último símbolo no lado direita da produção.

Introdução

- **Recursão à Esquerda**
  - As produções recursivas à direita produzem árvores que crescem para baixo e para a direita, como na Fig. 2 (b).
  - As árvores que crescem dessa forma dificultam a tradução de expressões contendo operadores associativos à esquerda, como o operador de subtração.
  - Abaixo, Fig. 2



Formas recursivas à esquerda e à direita para gerar uma seqüência de símbolos.

Introdução

* **Recursão à Esquerda**
  * Em "Gramáticas Livres de Contexto", vamos considerar formas mais **gerais** de recursão à esquerda e mostrar como toda a recursão à esquerda pode ser eliminada de uma gramática.

Introdução

* **Gramáticas Representativas**
  * As construções sintáticas que começam com palavras-chave como, por exemplo **`while`** ou **`int`** são reativamente fáceis de analisar, pois a palavra-chave guia na escolha da produção da gramática que deve ser aplicada para casá-la com a entrada.
  * Portanto, vamos nos concentrar em expressões, que apresentam um desafio maior, devido à associatividade e precedência dos operadores.

Introdução

* **Gramáticas Representativas**
  * Associatividade e precedência são capturadas na gramática a seguir para descrever expressões, termos e fatores.

  * O não-terminal E representa expressões consistindo em termos separados pelo operador +, T representa termos consistindo em fatores separados pelo operador * e F representa fatores que podem ser expressões entre parênteses ou identificadores.

Introdução

* **Gramáticas Representativas**

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

  * A gramática da expressão acima pertence à classe de gramáticas LR que são adequadas para a análise sintática **ascendente.**

Introdução

* **Gramáticas Representativas**
  * Essa gramática pode ser adaptada para incorporar novos operadores e novos níveis de precedência.
  * No entanto, ela não pode ser usada com o método de análise **descendente,** pois é recursiva à esquerda.

Introdução

* **Gramáticas Representativas**
  * Uma variação da gramática da expressão anterior sem recursão à esquerda será usada para a análise sintática descendente:

$$
\begin{aligned}
E &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \lambda \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \lambda \\
F &\rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
$$

Introdução

* **Gramáticas Representativas**
  * A gramática a seguir trata os operadores + e * da mesma forma, de modo que é útil para ilustrar as técnicas para o tratamento de ambiguidades durante a análise sintática:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

  * Neste exemplo, E representa expressões de todos os tipos. A gramática acima permite mais de uma árvore de derivação para expressões como $a + b * c$.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Consideraremos agora a natureza dos erros sintáticos e as estratégias gerais utilizadas para a recuperação de erro.
  * Duas dessas estratégias, chamadas *modo pânico* e *recuperação em nível de frase*, são discutidas com mais detalhes em conexão com os métodos de análise específicos.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Se um compilador tivesse de processar apenas programas corretos, o seu projeto e sua implementação seriam muito simplificados.
  * Porém, espera-se que um compilador auxilie o programador na localização e rastreamento de erros que inevitavelmente surgem nos programas, apesar de esforços do programador.

Introdução

* **Tratamento de Erro de Sintaxe**
  * É impressionante que poucas linguagens tenham sido projetadas com o tratamento de erro em mente, embora os erros sejam tão comuns.
  * Nossa civilização seria radicalmente diferente se as linguagens faladas tivessem os mesmos requisitos de precisão sintática das linguagens de computador.

Introdução

* **Tratamento de Erro de Sintaxe**
  * A maioria das especificações de linguagem de programação não descreve como um compilador deve responder a erros.
  * O tratamento de erros é deixado para o projetista de compilador.
  * Planejar o tratamento de erros logo no início pode simplificar a estrutura de um compilador e melhorar seu tratamento dos erros.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Os erros de programação mais comuns podem ocorrer em diferentes níveis:
    * Erros ***léxicos*** incluem ortografias erradas de identificadores, palavras-chave ou operadores – por exemplo, o uso de um identificador *elipseSize* no lugar de *ellipseSize* – e a ausência de aspas ao redor do texto definido como uma cadeia de caracteres.

    * Erros ***sintáticos*** incluem ponto-e-vírgulas mal colocados ou chaves extras ou faltando; ou seja, "{" ou "}". Outro exemplo: em C ou Java, a definição de um comando **`case`** sem um **`switch`** delimitando-o é um erro sintático, porém essa situação normalmente é permitida pelo analisador sintático e somente é identificada mais adiante no processamento, no momento da geração de código pelo compilador.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Os erros de programação mais comuns podem ocorrer em diferentes níveis:
    * Erros ***semânticos*** incluem divergências de tipo entre operadores e operandos. Por exemplo, um comando **`return`** em um método Java com tipo de resultado **`void`**.
    * Erros ***lógicos*** incluem desde o raciocínio incorreto por parte do programador ao uso, em um programa C, do operador de atribuição = em vez do operador de comparação ==. O programa contendo = pode estar bem formado; porém, pode não refletir a intenção do programador.

Introdução

* **Tratamento de Erro de Sintaxe**
  * A precisão dos métodos de análise permite que erros sintáticos sejam detectados eficientemente.
  * Vários métodos de análise, como os métodos LL e LR, detectam um erro o mais cedo possível; ou seja, quando a cadeia de tokens retornados pelo analisador léxico não puder mais ser analisada de acordo com a gramática para a linguagem.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Mais precisamente, esses métodos possuem a *propriedade de prefixo viável*, significando que detectam a ocorrência de um erro assim que encontram um prefixo da entrada que não pode ser completado para formar uma cadeia na linguagem.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Outro motivo para enfatizar a recuperação de erros durante a análise é que muitos erros parecem ser sintáticos, qualquer que seja sua causa, e são expostos quando a análise não pode continuar.

  * Alguns erros semânticos, como a incompatibilidade de tipo, também podem ser detectados eficientemente; porém, a detecção precisa dos erros semânticos e lógicos é em geral uma tarefa difícil de ser resolvida em tempo de compilação.

Introdução

* **Tratamento de Erro de Sintaxe**
  * O recuperador de erros em um analisador sintático possui objetivos simples, mas desafiadores em sua implementação:
    * Informar a presença de erros de forma clara e precisa.
    * Recuperar-se de cada erro com rapidez suficiente para detectar erros subsequentes.
    * Acrescentar um custo mínimo no processamento de programas corretos.

Introdução

* **Tratamento de Erro de Sintaxe**
  * Felizmente, **os erros comuns não são complicados**, portanto um mecanismo de tratamento de erros relativamente simples muitas vezes é suficiente.
  * Como um recuperador de erro deve informar a presença de um erro? No mínimo, ele precisa informar o local no programa fonte onde o erro foi detectado, pois existe uma boa chance de que o local exato do erro seja em um dos tokens anteriores.
  * Uma estratégia comum é a impressão da linha problemática com um apontador para a posição em que o erro foi detectado.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Quando um erro é detectado, como o analisador sintático deve recuperar-se?** Embora não exista uma única estratégia aceita universalmente, alguns métodos possuem ampla aplicabilidade.

  * A técnica mais simples é interromper a análise sintática e emitir uma mensagem de erro informativa assim que o primeiro erro for detectado.

Introdução

* **Estratégias de Recuperação de Erro**
  * Erros adicionais muitas vezes não são tratados se o analisador sintático puder restaurar-se para um estado no qual o processamento da entrada pode prosseguir na esperança de que o processamento posterior fornecerá informações de diagnóstico significativas.

  * Se os erros se acumularem, é melhor o compilador desistir depois de ultrapassar algum limite de erros do que produzir uma incômoda avalanche de “falsos” erros.

Introdução

* **Estratégias de Recuperação de Erro**
  * A seguir serão apresentadas as seguintes estratégias de recuperação de erro:
    * modo pânico;
    * nível de frase;
    * produções de erro; e
    * correção global.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Recuperação no Modo Pânico**
    * Com esse método, ao detectar um erro, o analisador sintático descarta um símbolo da entrada de cada vez até que um dentre um conjunto de *tokens de sincronismo* seja encontrado.

    * Os tokens de sincronismo normalmente são delimitadores, como o ponto-e-vírgula ou }, cujo papel no programa fonte é claro e não ambíguo.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Recuperação no Modo Pânico**
    * O projetista do compilador deve selecionar o tokens de sincronismos de acordo com a linguagem fonte.
    * Embora a correção no modo pânico normalmente ignore uma quantidade considerável de símbolos terminais do programa fonte sem se preocupar com a busca de erros adicionais, ele tem a vantagem da simplicidade e, diferentemente de alguns métodos que serão considerados mais tarde, tem-se a garantia de não entrar em um *loop* infinito.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Recuperação em Nível de Frase**
    * Ao detectar um erro, um analisador sintático pode realizar a correção local sobre o restante da entrada; ou seja, pode substituir o prefixo da entrada restante por alguma cadeia que permita a continuação da análise.
    * Um correção local típica compreende a substituição de uma vírgula por um ponto-e-vírgula, a exclusão de um ponto-e-vírgula desnecessário, ou a inserção de um ponto-e-vírgula.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Recuperação em Nível de Frase**
    * A escolha da correção local fica a critério do projetista do compilador.
    * Naturalmente, precisamos ter cuidado para que as substituições não provoquem *loops* infinitos, como aconteceria, por exemplo, se sempre inseríssemos algo na frente do símbolo da entrada corrente.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Recuperação em Nível de Frase**
    * A substituição em nível de frase tem sido usada em diversos compiladores com recuperação de erro, pois pode corrigir qualquer cadeia da entrada corrente.
    * Sua principal desvantagem é a dificuldade de lidar com situações em que o erro real ocorreu antes do ponto de detecção.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Produções de Erro**
    * Nesta estratégia de recuperação de erro podemos estender a gramática da linguagem em mãos com produções que geram construções erradas, antecipando assim os erros mais comuns que poderiam ser encontrados em um programa.

    * Um analisador sintático construído a partir de uma gramática estendida por essas produções de erro detecta os erros antecipadamente quando uma produção de erro é usada durante a análise.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Produções de Erro**
    * O analisador sintático pode, então, gerar um diagnóstico de erro apropriado sobre a construção errônea que foi reconhecida na entrada.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Correção Global**
    * Idealmente, gostaríamos que um compilador fizesse o mínimo de mudanças possível no processamento de uma cadeia de entrada incorreta.
    * Existem algoritmos que auxiliam na escolha de uma sequência mínima de mudanças a fim de obter uma correção com um custo global menor.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Correção Global**
    * Dada uma cadeia incorreta x na entrada e uma gramática G, esses algoritmos encontram uma árvore de derivação para uma cadeia relacionada y, tal que o número de inserções, exclusões e substituições de tokens necessários para transformar x em y seja o menor possível.

    * Infelizmente, esses métodos em geral são muito caros em termos de tempo e espaço de implementação, de modo que essas técnicas têm atualmente interesse meramente teórico.

Introdução

* **Estratégias de Recuperação de Erro**
  * **Correção Global**
    * Observe que o programa mais próximo do correto pode não ser aquele que o programador tinha em mente.
    * Apesar disso, a noção de correção menos dispendiosa oferece uma medida para avaliar as técnicas de recuperação de erro, e tem sido usada para encontrar cadeias de substituição ideais para a recuperação em nível de frase.

Gramáticas Livre de Contexto

* A estrutura sintática das construções de uma linguagem de programação é especificada pelas regras gramaticais de uma **gramática livre de contexto**.

* Dica: Revise o material de FTC para relembrar o conteúdo de GLC's.

Gramáticas Livre de Contexto

* A gramática livre de contexto é utilizada para especificar a sintaxe de uma linguagem.
* Trata-se de um conjunto de produções ou regras gramaticais.
* As gramáticas são usadas para organizar os *front-ends* de um compilador.

Gramáticas Livre de Contexto

* As gramáticas oferecem benefícios significativos para projetistas de linguagens e de compiladores:
  * Uma gramática provê uma especificação sintática precisa e fácil de entender para as linguagens de programação.
  * A partir de determinadas classes de gramáticas, podemos construir automaticamente uma analisador sintático eficiente, que descreve a estrutura sintática de um programa fonte. Como benefício adicional, durante o processo de construção do analisador, podem ser detectadas ambiguidades sintáticas, bem como outros problemas não identificados na fase inicial do projeto de uma linguagem.

Gramáticas Livre de Contexto

* As gramáticas oferecem benefícios significativos para projetistas de linguagens e de compiladores:
  * A estrutura imposta a uma linguagem por uma gramática devidamente projetada facilita a tradução de programas fonte para código correto e a detecção de erros.
  * Uma gramática permite o desenvolvimento de uma linguagem iterativamente, possibilitando lhe acrescentar novas construções para realizar novas tarefas. Essas novas construções podem ser integradas mais facilmente em implementações que seguem a estrutura gramatical da linguagem.

Gramáticas Livre de Contexto

* Uma gramática descreve naturalmente a estrutura hierárquica da maioria das construções de linguagens de programação.

* Por exemplo, um comando *if-else* em Java pode ter a forma:

if (expression) statement **else** statement

Gramáticas Livre de Contexto

* Ou seja, um comando *if-else* é a concatenação da palavra-chave **if**, um parêntese de abertura, uma expressão, um parêntese de fechamento, um comando, a palavra-chave **else** e outro comando.

* Usando a variável *expr* para representar uma expressão e a variável *stmt* para denotar um comando, essa regra de estruturação pode ser expressa como no exemplo:

$$stmt \rightarrow \textbf{if} \; (\; expr \;) \;\; stmt \;\; \textbf{else} \;\; stmt$$

* Onde a seta pode ser lida como “pode ter a forma”.

Gramáticas Livre de Contexto

* Essa regra é chamada de *produção*.
* Em um produção, os elementos léxicos como a palavra-chave **if** e os parênteses são chamados de *terminais*.
* Variáveis como *expr* e *stmt* representam sequências de terminais e são chamadas de *não-terminais*.

Gramáticas Livre de Contexto

* Uma *gramática livre de contexto* possui quatro componentes:

* 1) Um conjunto de símbolos *terminais*, às vezes chamados de "tokens". Os terminais são os símbolos elementares da linguagem, definidos pela gramática.

* 2) Um conjunto de *não-terminais*, às vezes chamados de "variáveis sintáticas". Cada não-terminal representa um conjunto de cadeias de terminais, de uma maneira que descreveremos.

Gramáticas Livre de Contexto

* Uma *gramática livre de contexto* possui quatro componentes:

* 3) Um conjunto de *produções*, em que cada consiste em não-terminal, chamado de *cabeça* ou *lado esquerdo* da produção, uma seta e uma sequência de terminais e/ou não-terminais, chamados de *corpo* ou *lado direito* da produção. Intuitivamente, o objetivo de uma produção é especificar uma das formas escritas de uma construção; se o não-terminal da cabeça representar uma construção, então o corpo representa uma forma escrita desta construção.

* 4) Uma designação de um dos não-terminais como o símbolo *inicial* da gramática.

Gramáticas Livre de Contexto

* **A Definição Formal de uma Gramática Livre de Contexto**

* 1) *Terminais* são os símbolos básicos a partir dos quais as cadeias são formadas. O termo "nome de token" é um sinônimo para "terminal" e frequentemente usaremos a palavra "token" em vez de terminal quando estiver claro que estamos falando apenas o nome do token. Assumimos que os terminais são os primeiros componentes dos tokens gerados pelo analisador léxico. No exemplo visto anteriormente, como exemplos de terminais, pode-se citar as palavras-chave **`if`** e **`else`**, e os símbolos "(" e ")".

Gramáticas Livre de Contexto

* **A Definição Formal de uma Gramática Livre de Contexto**

* 2) *Não-Terminais* são variáveis sintáticas que representam cadeias. No exemplo visto anteriormente, *stmt* e *expr* são não-terminais. Os conjuntos de cadeias denotando não-terminais auxiliam na definição da linguagem gerada pela gramática. Os não-terminais impõem uma estrutura hierárquica sobre a linguagem que é a chave para a análise sintática e tradução.

* 3) Em uma gramática, um não-terminal é distinguido como o símbolo inicial, e o conjunto de cadeias que ele representa é a linguagem gerada pela gramática. Por convenção, as produções para o símbolo inicial são listadas primeiro.

Gramáticas Livre de Contexto

* **A Definição Formal de uma Gramática Livre de Contexto**

* 4) As produções de uma gramática especificam a forma como os terminais e os não-terminais podem ser combinados para formar cadeias. Cada produção consiste em:
  * **(a)** Um não-terminal chamado de *cabeça* ou *lado esquerdo* da produção; essa produção define algumas das cadeias representadas pela cabeça.
  * **(b)** O símbolo $\rightarrow$. Às vezes, **: :=** é usado no lugar da seta.
  * **(c)** Um *corpo* ou *lado direito* da produção consiste em zero ou mais terminais e não-terminais. Os componentes do corpo descrevem uma forma como as cadeias do não-terminal do lado esquerdo da produção podem ser construídas.

Gramáticas Livre de Contexto

* **A Definição Formal de uma Gramática Livre de Contexto**
  * A gramática abaixo define expressões aritméticas simples. Nessa gramática, os símbolos terminais são: **id + - * / ( )**

  * Os símbolos não-terminais são *expression*, *term* e *factor*, e *expression* é o símbolo inicial da gramática (variável de partida).

expression	→	*expression* + *term*
expression	→	*expression* – *term*
expression	→	*term*
term	→	*term* * factor
term	→	*term* / factor
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

Gramática para expressões aritméticas simples.

Gramáticas Livre de Contexto

* Especificamos as gramáticas listando suas produções, com as produções para o símbolo inicial listadas primeiro.

* Consideramos que os dígitos, os sinais como < e <= e cadeias de caracteres em negrito como **while** são terminais.

* Um nome em itálico é um não-terminal, e qualquer nome ou símbolo não em itálico pode ser considerado um terminal.

Gramáticas Livre de Contexto

* Por conveniência de notação, as produções com o mesmo não-terminal como cabeça podem ter seus corpos agrupados, com os corpos alternativos separados pelo símbolo |, que lemos como “ou”.

* Vejamos um exemplo.

Gramáticas Livre de Contexto

* Exemplo 1:
  * Expressões consistindo em dígitos e sinais de adição e subtração.
  * Cadeias como 9-5+2, 3-1 ou 7.
  * Como um sinal de adição ou subtração precisa aparecer entre dois dígitos, nos referimos a essas expressões como "listas de dígitos separadas por sinais de adição ou subtração".
  * A gramática a seguir descreve a sintaxe dessas expressões. As produções são:

$$
\begin{aligned}
list &\rightarrow list + digit \\
list &\rightarrow list - digit \\
list &\rightarrow digit \\
digit &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Gramáticas Livre de Contexto

* Exemplo 1:
  * Os corpos das três produções com o não-terminal *list* como cabeça podem ser agrupados de forma equivalente como:

$$list \rightarrow list + digit \mid list - digit \mid digit$$

  * De acordo com nossas convenções, os terminais dessa gramática são os símbolos:

$$+ \quad - \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$$

  * Os não-terminais são os nomes em itálico *list* e *digit*, com *list* sendo o símbolo inicial, pois suas produções são dadas primeiro.

Gramáticas Livre de Contexto

* Dizemos que uma produção está *para* um não-terminal se o não-terminal for a cabeça da produção.

* Uma cadeia de terminais é uma sequência de zero ou mais terminais. A cadeia de zero terminal, escrita como λ, é chamada de cadeia vazia.

Gramáticas Livre de Contexto

* **Convenções de Notação**

* Para evitar ter de dizer sempre “estes são os terminais”, “estes são os não-terminais” etc., adotaremos as seguintes convenções de notação para as gramáticas que faremos:

* 1) Estes símbolos são terminais:
  * (a) Letras minúsculas do início do alfabeto, como a, b, c.
  * (b) Símbolos operadores como +, * e assim por diante.
  * (c) Símbolos de pontuação como parênteses, vírgula e assim por diante.
  * (d) Os dígitos 0,1,...,9.
  * (e) Cadeias em negrito como **id** ou **if**, cada um representando um único símbolo terminal.

Gramáticas Livre de Contexto

* **Convenções de Notação**

* Para evitar ter de dizer sempre "estes são os terminais", "estes são os não-terminais" etc., adotaremos as seguintes convenções de notação para as gramáticas que faremos:

* 2) Estes símbolos são não-terminais:
  * (a) Letras maiúsculas do início do alfabeto, como A, B, C.
  * (b) A letra S, que, quando aparece, normalmente é o símbolo inicial da gramática.
  * (c) Nomes em minúsculas e itálico, como *expr* ou *stmt*.
  * (d) Nas construções das linguagens de programação, letras maiúsculas podem ser usadas para representar não-terminais. Por exemplo, os não-terminais para expressões, termos e fatores normalmente são representados por E, T e F, respectivamente.

Gramáticas Livre de Contexto

* **Convenções de Notação**

* Para evitar ter de dizer sempre “estes são os terminais”, “estes são os não-terminais” etc., adotaremos as seguintes convenções de notação para as gramáticas que faremos:

* 3) Letras maiúsculas do fim do alfabeto, como X, Y, Z, representam símbolos da gramática, ou seja, não-terminais ou terminais.

* 4) Letras minúsculas do fim do alfabeto, principalmente u, v, ..., z, representam cadeias de terminais, possivelmente vazias.

* 5) Letras gregas minúsculas, por exemplo α, β, γ, representam cadeias de símbolos da gramática, possivelmente vazias. Assim, uma produção genérica pode ser escrita como $A \rightarrow \alpha$, onde A é o lado esquerdo da produção ou a cabeça, e α representa o corpo ou lado direito da produção.

Gramáticas Livre de Contexto

* **Convenções de Notação**

* Para evitar ter de dizer sempre "estes são os terminais", "estes são os não-terminais" etc., adotaremos as seguintes convenções de notação para as gramáticas que faremos:

* 6) Um conjunto de produções $A \to \alpha_1, A \to \alpha_2, ..., A \to \alpha_k$, com um lado esquerdo comum A, denominadas *produções-A*, pode ser escrito como $A \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$. Chame $\alpha_1, \alpha_2, ..., \alpha_k$ de *alternativas* para A.

* 7) A menos que indicado de outra forma, o lado esquerdo da primeira produção é o símbolo inicial.

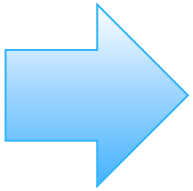
Gramáticas Livre de Contexto

* **Convenções de Notação**

* Usando essas convenções, podemos reescrever a gramática vista anteriormente de forma concisa como:

expression	→	*expression* + *term*
expression	→	*expression* – *term*
expression	→	*term*
term	→	*term* * factor
term	→	*term* / factor
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

Gramática para expressões aritméticas simples.



$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

* As convenções de notação nos dizem que E, T e F são não-terminais, e E é o símbolo inicial. Os símbolos restantes são terminais.

Gramáticas Livre de Contexto

* Uma gramática **deriva** cadeias começando com o símbolo inicial e substituindo repetidamente um não-terminal pelo corpo de uma produção para esse não-terminal.

* As cadeias de terminais que podem ser derivadas do símbolo inicial formam a ***linguagem*** definida pela gramática.

* Dica: Revise o material de FTC para relembrar o conteúdo de derivação.

Gramáticas Livre de Contexto

* A análise sintática consiste em, a partir de uma cadeia de terminais, tentar descobrir como **derivá-la** a partir do símbolo inicial da gramática, e, se não for possível, informar os erros de sintaxe dentro dessa cadeia.

* **A análise sintática é um dos problemas mais importantes em toda a compilação.**

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * As estruturas de dados de árvore têm destaque na compilação.

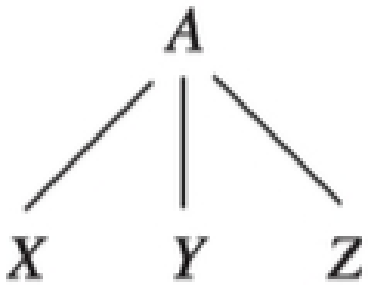
  * Uma árvore consiste em um ou mais *nós*. Os nós podem ter *rótulos*, que trataremos como símbolos da gramática. Quando desenhamos árvores, colocamos o pai de um nó acima desse nó e desenhamos uma aresta entre eles. A raiz, então é o nó mais alto (do topo).

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Se o nó *N* é o pai do nó *M*, então *M* é um *filho* de *N*. Os filhos de um nó são chamados de *irmãos*. Eles têm uma ordem, *a partir da esquerda*, e, quando desenhamos árvores, ordenamos os filhos de determinado nó dessa maneira.
  * Um nó sem filhos é chamado de *folha*. Outros nós – aqueles com um ou mais filhos – são *nós interiores*.
  * Um *descendente* de um nó *N* é o próprio *N*, um filho de *N*, um filho de um filho de *N*, e assim por diante, para qualquer número de nível. Dizemos que o nó *N* é um *ancestral* do nó *M* se *M* for um descendente de *N*.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Uma árvore de derivação mostra de forma representativa como o símbolo inicial de uma gramática deriva uma cadeia na linguagem.
  * Se o não-terminal A possui uma produção $A \rightarrow XYZ$, uma árvore de derivação pode ter um nó interior rotulado com A, com três filhos chamados X, Y e Z, da esquerda para a direita.
  * Como no exemplo da Fig. 3:

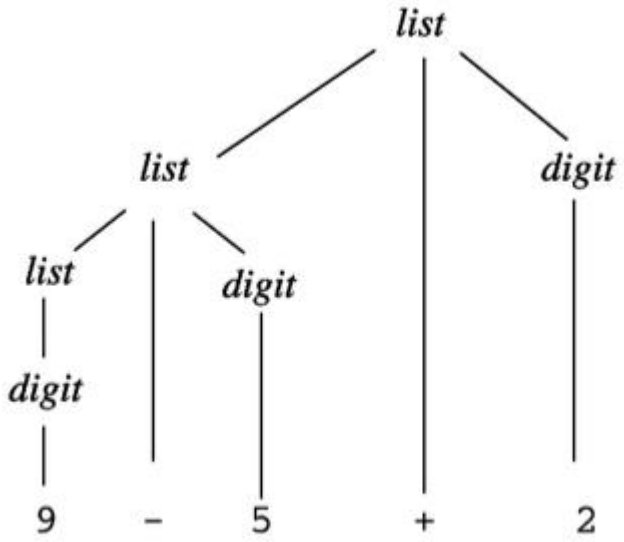


Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Formalmente, dada uma gramática livre de contexto, uma *árvore de derivação* de acordo com a gramática é uma árvore com as seguintes propriedades:
    * 1) A raiz é rotulada pelo símbolo inicial.
    * 2) Cada folha é rotulada por um terminal ou por λ.
    * 3) Cada nó interior é rotulado por um não-terminal.
    * 4) Se *A* é o não-terminal rotulando algum nó interior e $X_1, X_2, ..., X_n$ são os rótulos dos filhos desse nó da esquerda para a direita, deve haver uma produção $A \rightarrow X_1, X_2, ... , X_n$. Cada $X_1, X_2, ..., X_n$ representa um símbolo que é um terminal ou um não-terminal. Como um caso especial, se *A* deriva em λ é uma produção, um nó rotulado com *A* pode ter um único filho rotulado com λ.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * A derivação de 9-5+2 é ilustrada na Fig. 4 a seguir.
  * Cada nó da árvore é rotulado por um símbolo da gramática.
  * Um nó interior e seus filhos correspondem a uma produção; o nó interior corresponde à cabeça da produção, e os filhos ao seu corpo.



A árvore de derivação para 9−5+2 de acordo com a gramática

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Na figura, a raiz é rotulada com *list*, o símbolo inicial da gramática no exemplo 1 visto anteriormente. Os filhos da raiz são rotulados, da esquerda para a direita, com *list*, + e *digit*.

  * Observe que $list \rightarrow list + digit$ é uma produção na gramática do exemplo 1. O filho esquerdo da raiz é semelhante à raiz, com um filho rotulado com – ao invés de +. Os três nós rotulados com *digit* possuem um filho que é rotulado por um dígito.

  * Da esquerda para a direita, as folhas de uma árvore de derivação formam o resultado da árvore, que é a cadeia *gerada* ou *derivada* do não-terminal na raiz da árvore de derivação. Na Fig. 4, o resultado é 9-5+2; por conveniência, todas as folhas aparecem no nível inferior.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * **Caminhamento em Árvore**
    * Os caminhamentos ou travessias em árvores são usados para descrever a avaliação de atributos e para especificar a execução dos fragmentos de código em um esquema de tradução.

    * A *travessia* de uma árvore começa na raiz e visita cada nó da árvore em alguma ordem.

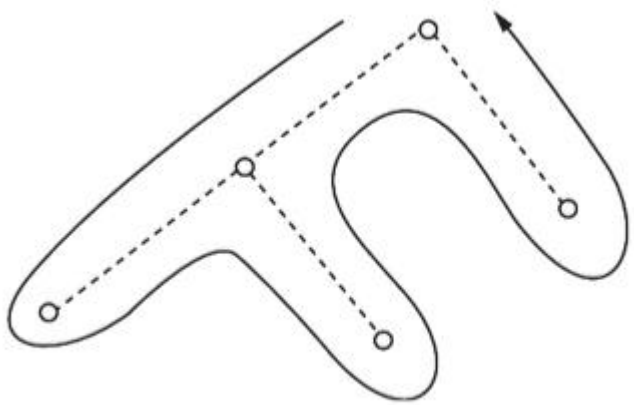
Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * **Caminhamento em Árvore**
    * Uma busca em profundidade (*depth-first*) começa na raiz e visita recursivamente os filhos de cada nó em qualquer ordem, não necessariamente da esquerda para a direita.

    * Ela é chamada "busca em profundidade" porque visita um filho não visitado de um nó sempre que puder, de modo que visita os nós mais distantes da raiz (mais "profundos") o mais rapidamente que puder.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * **Caminhamento em Árvore**
    * O procedimento de visita através de busca em profundidade, que visita os nós da esquerda para a direita, é mostrado na Fig. 5 abaixo:



Exemplo de busca em profundidade em uma árvore.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Outra definição da linguagem gerada por uma gramática é o conjunto de cadeias de terminais que podem ser geradas por alguma árvore de derivação.

  * O processo de encontrar uma árvore de derivação para determinada cadeia de terminais é chamada de *análise* ou *reconhecimento* dessa cadeia.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Precisamos ter cuidado ao falar sobre a estrutura de uma cadeia estar de acordo com uma gramática.
  * Uma gramática pode ter mais de uma árvore de derivação gerando determinada cadeia de terminais.
  * Essa gramática é considerada *ambígua*.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Para mostrar que uma gramática é ambígua, tudo o que precisamos fazer é encontrar uma cadeia de terminais que seja o resultado de mais de uma árvore de derivação.

  * Dica: Revise o material de FTC para relembrar o conteúdo de árvores de derivação.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Como uma cadeia com mais de uma árvore de derivação normalmente possui mais de um significado, **temos de projetar gramáticas não ambíguas para compilar aplicações**, ou usar gramáticas ambíguas com regras adicionais para solucionar as ambiguidades.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Suponha que usemos um único não-terminal *string* e que não façamos distinção entre dígitos e listas, como no Exemplo 1.
  * Poderíamos então escrever a gramática:

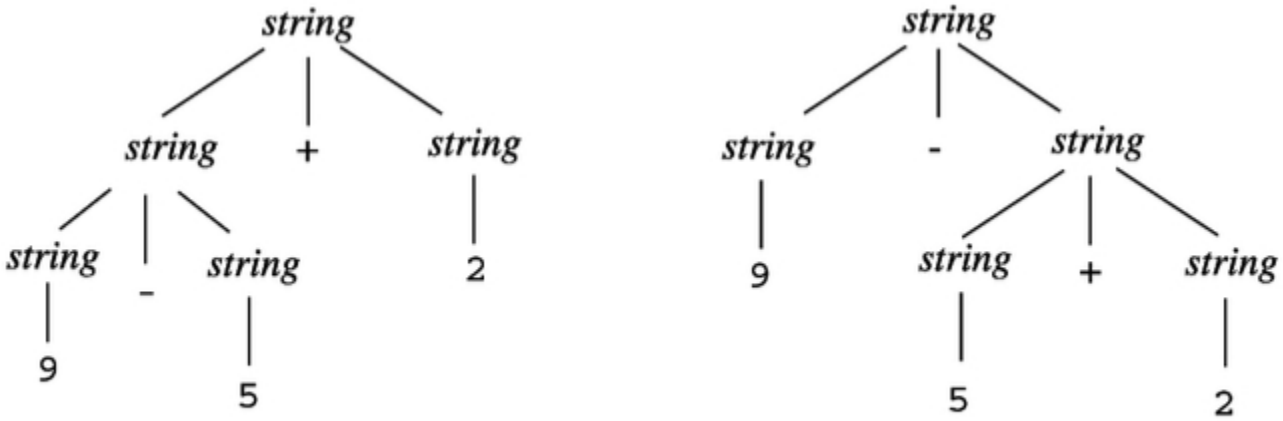
$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

  * Combinar a noção de *dígito* e lista em um não-terminal string faz sentido superficialmente, pois um único dígito é um caso especial de uma lista.

  * A Fig. 6 a seguir, contudo, mostra que uma expressão como 9-5+2 tem mais de uma árvore de derivação para essa gramática.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Abaixo, Fig. 6.



Duas árvores de derivação para 9−5+2.

  * As duas árvores para 9-5+2 correspondem às duas maneiras de colocar parênteses na expressão: (9-5)+2 e 9-(5+2).
  * No primeiro caso, o valor da expressão é 6, enquanto o segundo uso de parênteses dá à expressão o valor inesperado de 2 em vez de 6.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * A gramática do Exemplo 1 não permite essa interpretação.
  * Essa é a vantagem de uma **gramática não-ambígua**.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Definindo formalmente, uma árvore de derivação é uma representação gráfica de uma derivação que filtra a ordem na qual as produções são aplicadas para substituir não-terminais.

  * Cada nó interior de uma árvore de derivação representa a aplicação de uma produção. O nó interior é rotulado com o não-terminal A do lado esquerdo da produção; os filhos do nó são rotulados, da esquerda para a direita, pelos símbolos do corpo da produção pelo qual este A foi substituído durante a derivação.

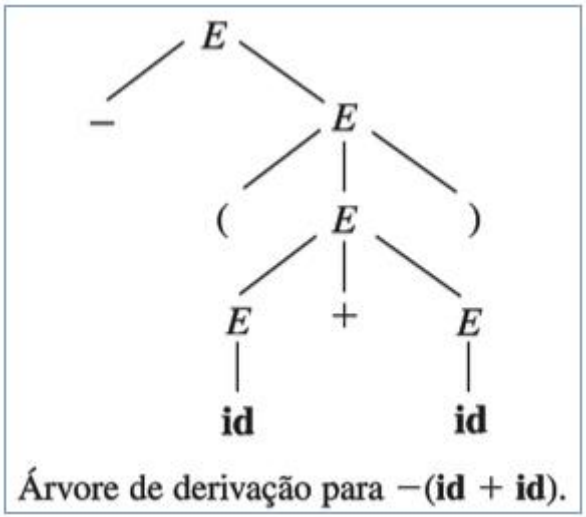
Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Por exemplo, a árvore de derivação para −(**id** + **id**), na Fig. 7 abaixo, resulta da derivação 1 ou 2 da gramática a seguir:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$ Gramática

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$ Derivação 1

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id})$$ Derivação 2



Árvore de derivação para −(**id** + **id**).

Gramáticas Livre de Contexto

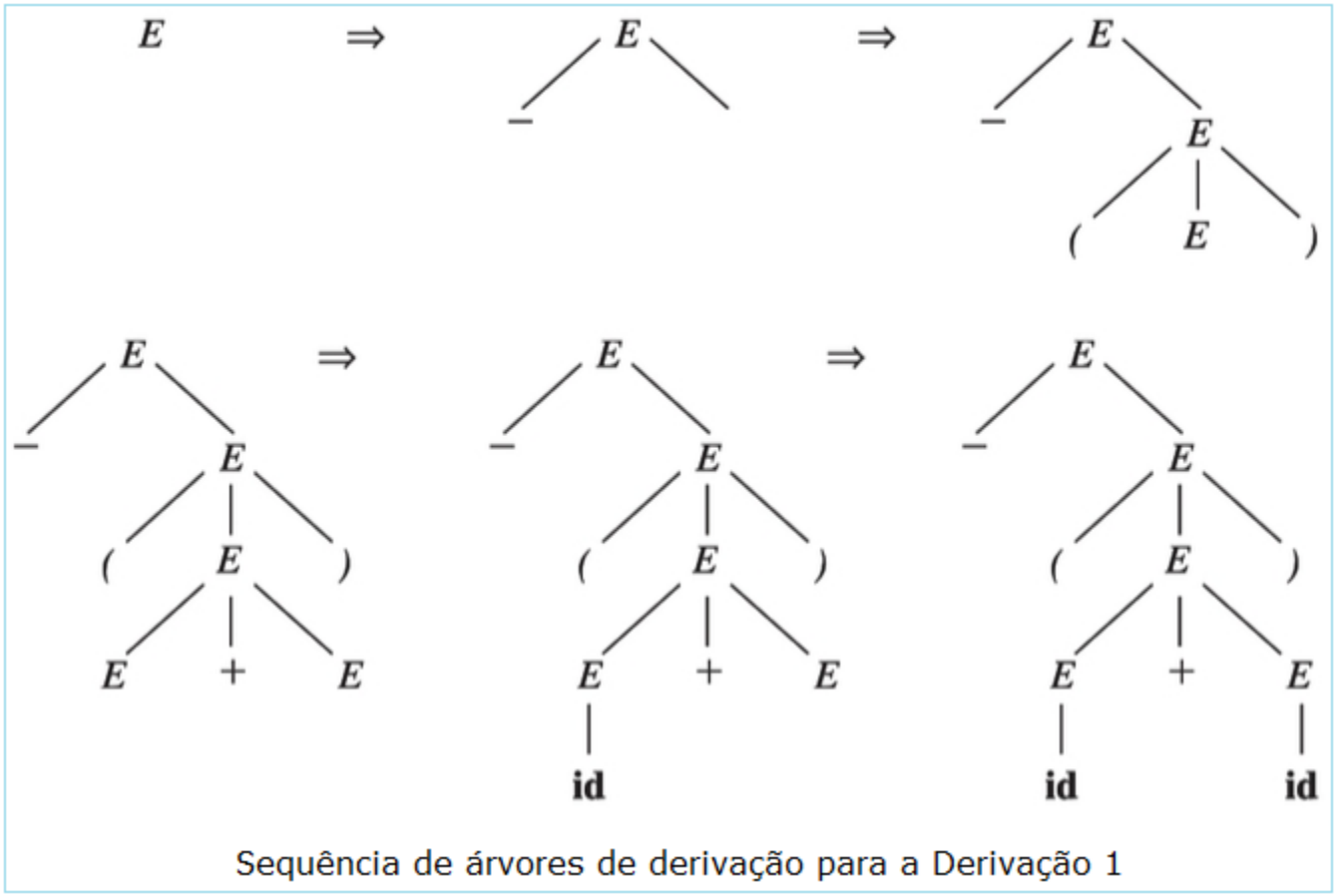
* **Árvores de Derivação**
  * A sequência de árvores de derivação construída a partir da Derivação 1 é mostrada na Fig. 8 a seguir.

  * No **primeiro passo** da derivação, E é expandido em $-E$. Para modelar esse passo, acrescente dois filhos com – e E, à raiz E da árvore inicial. O resultado é a segunda árvore.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Fig. 8

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$ Derivação 1



Sequência de árvores de derivação para a Derivação 1

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * No **segundo passo** da derivação, -E é expandido em $-(E)$. Consequentemente, inclua três filhos, rotulados com (, E e), à folha rotulada com E da segunda árvore, para obter a terceira árvore com fronteira $-(E)$. Continuando dessa forma, obtemos a árvore de derivação completa como a sexta árvore.

    * As folhas de uma árvore de derivação são rotuladas pelos não-terminais ou terminais e, lidas da esquerda para a direita, constituem uma forma sentencial, chamada de fronteira da árvore.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Como uma árvore de derivação ignora variações na ordem em que os símbolos nas formas sentenciais são substituídos, existe um relacionamento *muitos-para-um* entre as derivações e as árvores de derivações.

  * Por exemplo, as derivações 1 e 2 anteriores são associadas à mesma árvore de derivação final ilustrada.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Para fazermos a análise sintática, utilizaremos uma derivação mais à esquerda ou mais à direita, uma vez que há um relacionamento *um-para-um* entre as árvores de derivação e as derivações mais à esquerda ou mais à direita.
  * Ambas as derivações escolhem uma ordem particular para substituir os símbolos nas formas sentenciais, de modo que também filtram variações na ordem.
  * Não é difícil mostrar que toda árvore de derivação possui associada a ela uma única derivação mais à esquerda e uma única derivação mais à direita.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Como vimos, uma gramática que produz mais de uma árvore de derivação para alguma sentença é considerada *ambígua*.
  * Colocando de outra forma, uma gramática livre de contexto é ambígua se permitir a construção de mais de uma derivação mais à esquerda ou mais de uma derivação mais à direita para a mesma sentença.

Gramáticas Livre de Contexto

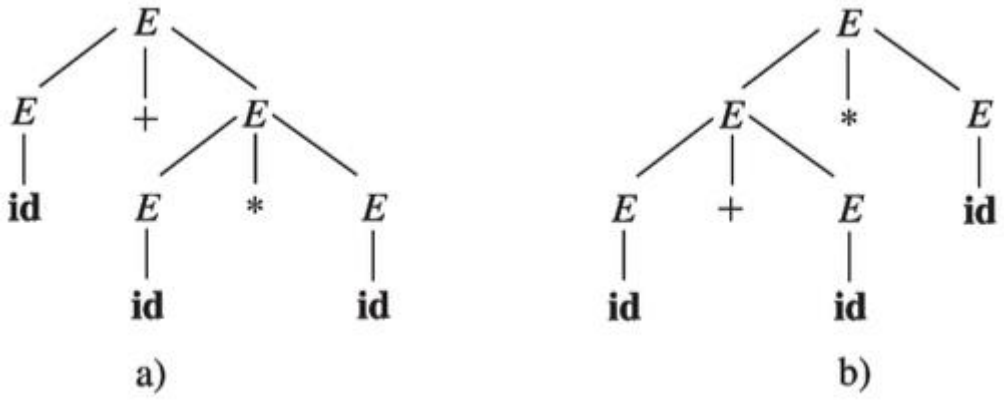
* **Árvores de Derivação**
  * A gramática de expressão aritmética abaixo, permite duas derivações mais à esquerda distintas para a sentença **id + id * id**.

Gramática: $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * A Fig. 9 abaixo mostra as duas árvores de derivação correspondentes.
  * Observe que a árvore de derivação (a) reflete a precedência normalmente assumida dos operadores + e *, o que não acontece na árvore (b). Ou seja, é comum tratar o operador * como tendo maior precedência do que +, o que corresponde à avaliação de uma expressão do tipo $a + b * c$ como $a + (b * c)$, em vez de $(a + b) * c$.



Duas árvores de derivação para **id + id * id**.

Gramáticas Livre de Contexto

* **Árvores de Derivação**
  * Para a maioria dos analisadores sintáticos, é desejável que a **gramática seja não ambígua**, pois, do contrário, não podemos determinar univocamente qual árvore de derivação selecionar para uma dada sentença.

  * Por outro lado, em algumas situações, é conveniente usar gramáticas ambíguas cuidadosamente escolhidas, juntamente com regras para torná-las não-ambíguas, de forma que a árvore de derivação construída seja a única árvore possível para a sentença dada, "descartando" assim árvores de derivação indesejáveis.

Gramáticas Livre de Contexto

* Exemplo 2:
  * As palavras-chave nos permitem reconhecer comandos, pois a maioria deles começa com palavra-chave ou um caractere especial. As exceções a essa regra incluem as atribuições e as chamadas de procedimento. Os comandos definidos pela gramática ambígua abaixo são válidos em **Java.**

stmt	→	**id** = *expression* ;
	|	**if** (*expression*) *stmt*
	|	**if** (*expression*) *stmt* **else** *stmt*
	|	**while** (*expression*) *stmt*
	|	**do** *stmt* **while** (*expression*) ;
	|	{ *stmts* }
stmts	→	*stmts stmt*
	|	λ

Uma gramática para um subconjunto dos comandos Java.

Gramáticas Livre de Contexto

* Exemplo 2:
  * Na primeira produção para *stmt*, o terminal **id** representa qualquer identificador.
  * As produções para *expression* não aparecem.
  * Os comandos de atribuição especificados pela primeira produção são válidos em Java, muito embora a linguagem trate = como um operador de atribuição que pode aparecer dentro de uma expressão.
  * Por exemplo, Java permite a = b = c, mas esta gramática não o permite.

Gramáticas Livre de Contexto

* Exemplo 2:

  * O não-terminal *stmts* gera uma lista de comandos possivelmente vazia. A segunda produção para *stmts* gera a lista vazia λ. A primeira produção gera uma lista possivelmente vazia de comandos seguida por um comando.

  * O posicionamento dos ponto-e-vírgulas é sutil; eles aparecem no fim de cada corpo que não termina em *stmt*. Esta técnica impede o acúmulo de ponto-e-vírgulas após comandos como *if* e *while*, que terminam com subcomandos aninhados. Quando o subcomando aninhado é uma atribuição ou um do-while, um ponto-e-vígula é gerado como parte do subcomando.

Gramáticas Livre de Contexto

* **Gramáticas Livres de Contexto Versus Expressões Regulares**
  * Antes de terminar o assunto sobre gramáticas e suas propriedades, veremos que as gramáticas livres de contexto são consideradas uma notação mais poderosa que as expressões regulares.

  * Toda construção que pode ser descrita por uma expressão regular também pode ser descrita por uma gramática, mas não vice-versa. Alternativamente, toda linguagem regular é uma linguagem livre de contexto, mas não vice-versa.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * As gramáticas são capazes de descrever a maior parte, mas não toda a sintaxe das linguagens de programação.
  * Por exemplo, o fato de os identificadores serem declarados antes de seu uso não pode ser descrito por uma gramática livre de contexto.
  * Portanto, as sequências de tokens aceitos por um analisador sintático formam um superconjunto da linguagem de programação.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * As fases subsequentes do compilador devem analisar a saída do analisador sintático para garantir compatibilidade com as regras que não são verificadas durante a análise.
  * Veremos como dividir o trabalho entre um analisador léxico e um analisador sintático.
  * Depois, iremos considerar várias técnicas de transformação que podem ser aplicadas às gramáticas a fim de adequá-las aos diversos métodos de reconhecimento sintático.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * Uma técnica pode eliminar a ambiguidade da gramática, e outras técnicas, por exemplo – a eliminação da recursão à esquerda e fatoração à esquerda – são importantes para reescrever as gramáticas, tornando-as adequadas para a análise descendente.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Análise Léxica Versus Análise Sintática**
    * Conforme vimos, tudo o que pode ser descrito por uma expressão regular também pode ser descrito por uma gramática livre de contexto.
    * Podemos portanto, perguntar: "Por que usar expressões regulares para definir a sintaxe léxica de uma linguagem?"

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Análise Léxica Versus Análise Sintática**
    * Existem diversos motivos:
    * 1) Separar a estrutura sintática de uma linguagem em partes léxica e não-léxica provê uma forma conveniente de particionar o *front-end* de um compilador em dois módulos de tamanho administrável.
    * 2) As regras léxicas de uma linguagem são frequentemente muito simples, e, para descrevê-las, não precisamos de uma notação tão poderosa quanto as gramáticas livres de contexto.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Análise Léxica Versus Análise Sintática**
    * Existem diversos motivos:

    * 3) As expressões regulares geralmente oferecem uma notação mais fácil de entender e mais concisa para tokens do que as gramáticas.

    * 4) Analisadores léxicos mais eficientes podem ser construídos automaticamente a partir de expressões regulares do que por meio de gramáticas arbitrárias.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Análise Léxica Versus Análise Sintática**
    * Não existem diretrizes consolidadas quanto ao que colocar nas regras léxicas, ao contrário do que ocorre com as regras sintáticas.
    * As expressões regulares são mais adequadas para descrever a estrutura de construções como identificadores, constantes, palavras-chave e espaço em branco.
    * As gramáticas, por outro lado, são mais úteis para descrever estruturas aninhadas como parênteses balanceados, a combinação das construções *begin-end*, os *if-then-else* correspondentes, e assim por diante. **Essas estruturas aninhadas não podem ser descritas por expressões regulares.**

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Às vezes, uma gramática ambígua pode ser reescrita para eliminar a ambiguidade.
    * Por exemplo, vamos eliminar a ambiguidade da seguinte gramática com “else vazio”.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Gramática:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \textbf{other}
\end{array}
$$

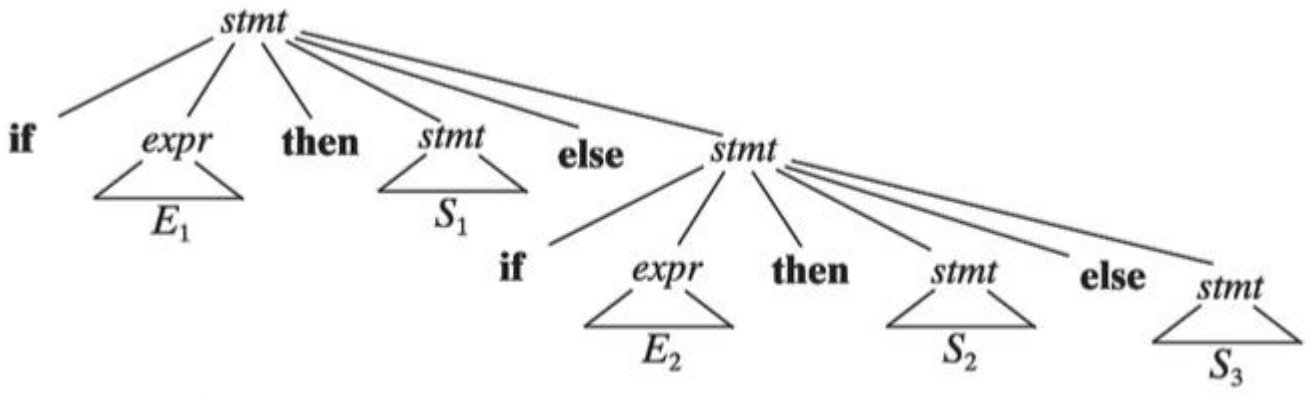
    * Neste exemplo, "**other**" representa qualquer outro comando.

Gramáticas Livre de Contexto

- **Escrevendo uma Gramática**
  - **Eliminando a Ambiguidade**
    - De acordo com essa gramática, o comando condicional composto:

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

    - Possui a árvore de derivação mostrada na Fig. 10 abaixo:



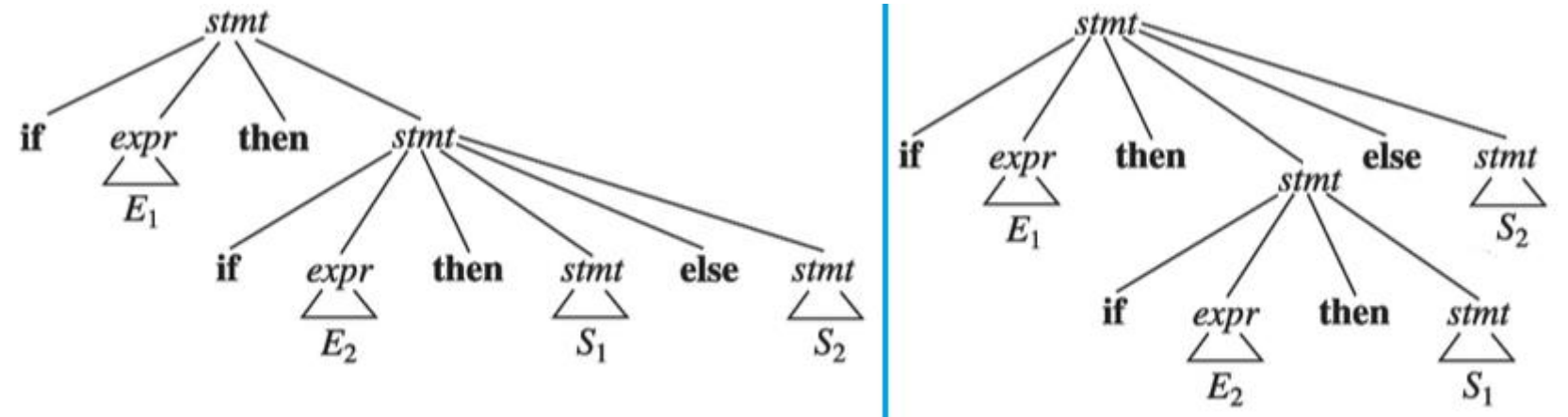
Árvore de derivação para um comando condicional composto.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * A gramática em questão é ambígua, pois a cadeia:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

    * Possui as duas árvores de derivação mostradas na Fig. 11 abaixo:



Duas árvores de derivação para uma sentença ambígua.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Em todas as linguagens de programação com comandos condicionais dessa forma, **a primeira árvore de derivação é a preferida**.

    * A regra geral é "casar cada **else** com o **then** não casado mais próximo". Observe que a linguagem C e suas derivações estão incluídas nessa classe. Embora a família de linguagens C não use a palavra-chave **then**, seu papel é desempenhado pelo par de parênteses que aparece após a condição **if**.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Essa regra de remoção de ambiguidade teoricamente pode ser incorporada diretamente em uma gramática, mas na prática raramente aparece nas produções.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Como exemplo de eliminação de ambiguidade, podemos reescrever a gramática do else vazio definindo a gramática não-ambígua apresentada a seguir.

    * A ideia é que um comando, *stmt*, entre um **then** e um **else**, precisa ser "casado"; ou seja, o comando interno não pode terminar com um **then** ainda não-casado ou aberto seguido por qualquer outro comando, pois o **else** seria forçado a casar com esse **then** não-casado. Um comando casado é um comando **if-then-else** contendo somente comandos casados ou é qualquer outro tipo de comando incondicional.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Ambiguidade**
    * Portanto, podemos usar seguinte gramática não-ambígua abaixo. Essa gramática reconhece as mesmas cadeias geradas pela gramática do else vazio, mas só permite uma análise para a cadeia em questão; a saber, aquela que associa cada **else** ao **then** anterior mais próximo não-casado.

Cadeia: **if** E_1 **then if** E_2 **then** S_1 **else** S_2

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**other**

Gramática ambígua

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

Gramática não-ambígua para comandos if-then-else

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Uma gramática possui *recursão à esquerda* se ela tiver um não-terminal A tal que exista uma derivação $A \rightarrow A\alpha$ para alguma cadeia α.

    * Os métodos de análise descendentes **não podem tratar gramáticas com recursão à esquerda**, de modo que uma transformação é necessária para eliminar essa recursão.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Vimos anteriormente, na introdução, a *recursão à esquerda imediata*, onde existe uma produção na forma $A \rightarrow A\alpha$.
    * Agora, estudaremos o caso **geral**.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Veremos como o par de produções com recursão à esquerda $A \rightarrow A\alpha \mid \beta$ poderia ser substituído por produções sem recursão à esquerda:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \lambda$$

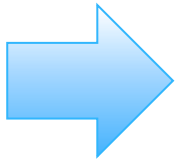
    * sem alterar as cadeias deriváveis de A. Essa regra, por si só, é suficiente para muitas gramáticas.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * A gramática de expressão não-recursiva à esquerda (2), vista anteriormente, repetida aqui, é obtida eliminando-se a recursão à esquerda imediata da gramática (1).

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Gramática 1



$$
\begin{array}{lcl}
E & \rightarrow & T\,E' \\
E' & \rightarrow & +\,T\,E' \mid \lambda \\
T & \rightarrow & F\,T' \\
T' & \rightarrow & *\,F\,T' \mid \lambda \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Gramática 2

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * O par de produções com recursão à esquerda $E \rightarrow E + T \mid T$ é substituído por $E \rightarrow T\,E'$ e $E' \rightarrow +\,T\,E' \mid \lambda$.
    * As novas produções para T e T' são obtidas de forma semelhante eliminando-se a recursão à esquerda imediata.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * A recursão à esquerda imediata pode ser eliminada pela técnica a seguir, que funciona para qualquer quantidade de produções A.
    * Primeiro, agrupe as produções:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

    * onde nenhum βi começa com um A.

Gramáticas Livre de Contexto

- **Escrevendo uma Gramática**
  - **Eliminando a Recursão à Esquerda**
    - Depois, substitua as produções-*A* por:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \lambda$$

    - O não-terminal A gera as mesmas cadeias de antes, contudo sem recursão à esquerda.
    - Esse procedimento elimina toda recursão à esquerda das produções A e A' (desde que nenhum α_i seja λ), mas não elimina a recursão à esquerda envolvendo derivações em dois ou mais passos.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Por exemplo, considere a gramática:

$$S \rightarrow A\,a \mid b$$
$$A \rightarrow A\,c \mid S\,d \mid \lambda$$

    * O não-terminal S é recursivo à esquerda porque $S \Rightarrow Aa \Rightarrow Sda$, mas não possui recursão **imediata à esquerda**.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Por exemplo, considere a gramática:

$$S \rightarrow A\,a \mid b$$
$$A \rightarrow A\,c \mid S\,d \mid \lambda$$

    * Para eliminar a recursão à esquerda, primeiro substituímos o S em $A \rightarrow S\,d$ para obter as produções-A a seguir:

$$A \rightarrow A\,c \mid A\,a\,d \mid b\,d \mid \lambda$$

* **Escrevendo uma Gramática**
  * **Eliminando a Recursão à Esquerda**
    * Nova escrita da gramática:

$$S \rightarrow A\,a \mid b$$
$$A \rightarrow A\,c \mid A\,a\,d \mid b\,d \mid \lambda$$

    * A eliminação da recursão à esquerda imediata entre essas produções-*A* gera a gramática a seguir:

$$S \rightarrow A\,a \mid b$$
$$A \rightarrow b\,d\,A' \mid A'$$
$$A' \rightarrow c\,A' \mid a\,d\,A' \mid \lambda$$

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * A fatoração à esquerda é uma transformação da gramática útil na produção de gramáticas adequadas para um reconhecedor sintático descendente.

    * Quando a escolha entre duas ou mais alternativas das produções-*A* não é clara, ou seja, elas começam com a mesma forma sentencial, podemos reescrever essas produções para adiar a decisão até que tenhamos lido uma cadeia da entrada longa o suficiente para tomarmos a decisão correta.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * Por exemplo, se tivermos as duas produções:

$$\begin{array}{lll} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \end{array}$$

    * Ao ler **if** na entrada, não sabemos imediatamente qual produção escolher para expandir *stmt*.
    * Em geral, se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ forem duas produções-*A*, e a entrada começar com uma cadeia não-vazia derivada de α, não sabemos se *A* deve ser expandido para $\alpha\beta 1$ ou $\alpha\beta 2$.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * Contudo, podemos adiar a decisão expandindo *A* para $\alpha A'$ e então, após ler a entrada derivada de α, expandimos A' para $\beta 1$ ou para $\beta 2$.

    * Ou seja, depois de fatoradas à esquerda, as produções originais se tornam:

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

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * Exemplo:
    * A gramática a seguir abstrai o problema do "else vazio":

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**other**

Gramática do "else vazio"

$$S \rightarrow i\,E\,t\,S \mid i\,E\,t\,S\,e\,S \mid a$$
$$E \rightarrow b$$

Abstração da gramática do "else vazio"

    * Neste exemplo, i, t e e representam **if**, **then** e **else**;
    * E e S representam respectivamente "expressão condicional" e "comando";
    * a representa **other** e b é uma expansão de exemplo de E.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * Após fatorar à esquerda, essa gramática se torna:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Abstração da gramática do "else vazio"

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \lambda$$
$$E \rightarrow b$$

Gramática fatorada à esquerda

    * Assim, podemos expandir S para $iEtSS'$ com a entrada i, e esperar até que $iEtS$ tenha sido lido para decidir se expandimos S' para eS ou para λ.
    * Ambas as gramáticas são ambíguas.

Gramáticas Livre de Contexto

* **Escrevendo uma Gramática**
  * **Fatoração à Esquerda**
    * Após fatorar à esquerda, essa gramática se torna:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Abstração da gramática do "else vazio"

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \lambda$$
$$E \rightarrow b$$

Gramática fatorada à esquerda

    * A ambiguidade se manifesta pela escolha sobre que produção usar quando um e (**else**) é visto. Podemos resolver essa ambiguidade escolhendo $S' \rightarrow eS$. Essa escolha corresponde à associação de um **else** com o **then** anterior mais próximo. Observe que a escolha $S' \rightarrow \lambda$ impede que e seja até mesmo colocado na pilha de símbolos ou removido da entrada, e com certeza está errada.

Gramáticas Sensíveis ao Contexto

* **Escrevendo uma Gramática**
  * **Construções de Linguagens Não-Livres de Contexto**
    * Algumas construções sintáticas encontradas em linguagens de programação típicas não podem ser especificadas usando apenas gramáticas livres de contexto.
    * Veremos duas dessas construções através de linguagens abstratas simples para ilustrar as dificuldades.

Gramáticas Sensíveis ao Contexto

- **Escrevendo uma Gramática**
  - **Construções de Linguagens Não-Livres de Contexto**
    - Exemplo 1:
    - A linguagem neste exemplo é uma abstração do problema de verificar **se os identificadores foram declarados antes de serem usados** em um programa.
    - A linguagem consiste em cadeias de forma wcw, onde o primeiro w representa a declaração de um identificador w, c representa um trecho de programa, e o segundo w representa o uso do identificador.
    - A linguagem abstrata é $L_1 = \{wcw \mid w \text{ está em } (a \mid b)^*\}$.
    - L_1 consiste em todas as palavras compostas de uma cadeia de as e bs separados por c, como em $aabcaab$.

Gramáticas Sensíveis ao Contexto

* **Escrevendo uma Gramática**
  * **Construções de Linguagens Não-Livres de Contexto**
    * Exemplo 1:
    * A dependência do contexto de L_1 implica diretamente a dependência de contexto de linguagens de programação como C e Java, que exigem a declaração dos identificadores antes de seus usos e permitem identificadores de qualquer tamanho.
    * Por esse motivo, uma gramática para C ou Java não distingue entre identificadores que possuem cadeias de caracteres diferentes. Em vez disso, todos os identificadores são representados na gramática por um token como **id**.
    * Em um compilador para tais linguagens, a fase de **análise semântica** verifica se os identificadores são declarados antes de serem usados.

Gramáticas Sensíveis ao Contexto

* **Escrevendo uma Gramática**
  * **Construções de Linguagens Não-Livres de Contexto**
    * Exemplo 2:
    * Neste exemplo, a linguagem não-livre de contexto (também conhecida como gramática sensível ao contexto ou dependente do contexto) é uma abstração do problema de verificar **se o número de parâmetros formais na declaração de uma função casa com o número de parâmetros reais em uma ativação da função**.
    * A linguagem consiste em cadeias da forma $a^n b^m c^n d^m$. (Lembre-se de que a^n significa a escrita de a n vezes.)
    * Neste exemplo, a^n e b^m poderiam representar as listas de parâmetros formais de duas funções declaradas respectivamente com n e m argumentos, enquanto c^n e d^m representam as listas de parâmetros reais nas chamadas para essas duas funções.

Gramáticas Sensíveis ao Contexto

* **Escrevendo uma Gramática**
  * **Construções de Linguagens Não-Livres de Contexto**
    * Exemplo 2:
    * A linguagem abstrata é $L_2 = \{a^n b^m c^n d^m \mid n >= 1 \; e \; m >= 1\}$.
    * Ou seja, L_2 consiste em cadeias da linguagem gerada pela expressão regular **a*b*c*d*** tais que o número de as e cs são iguais e o número de bs e ds são iguais.
    * Essa linguagem não é livre de contexto.
    * Novamente, a sintaxe típica das declarações e dos usos de função não se preocupa em contar o número de parâmetros.

Gramáticas Sensíveis ao Contexto

* **Escrevendo uma Gramática**
  * **Construções de Linguagens Não-Livres de Contexto**
    * Exemplo 2:
    * Por exemplo, uma chamada de função em uma linguagem tipo C poderia ser especificada por:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{id} \; (\; \textit{expr\_list} \;) \\
\textit{expr\_list} & \rightarrow & \textit{expr\_list} \; , \; \textit{expr} \\
 & | & \textit{expr}
\end{array}
$$

    * com produções adequadas para *expr*.

    * Verificar se o número de parâmetros em uma chamada está correto normalmente é feito durante a fase da **análise semântica**.

Análise Descendente

* Introduzimos a análise descendente (top-down) apresentando um gramática que é bem simples, além de adequada, para essa classe de métodos.

* A gramática da Fig. 12 a seguir gera um subconjunto dos comandos C ou Java.

* Usamos os terminais em negrito **if** e **for** para as palavras-chave "`if`" e "`for`", respectivamente, a fim de enfatizar que essas sequências de caracteres são tratadas como unidades, ou seja, como símbolos terminais. Além do mais, o terminal **expr** representa expressões.

Análise Descendente

* Abaixo, Fig. 12
  * Uma gramática mais elaborada usaria um não-terminal *expr* e teria produções para esse não-terminal.
  * Da mesma forma, **other** é um terminal representando outros comandos.

stmt	→	**expr** ;
	|	**if** (**expr**) *stmt*
	|	**for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*
	|	**other**
optexpr	→	λ
	|	**expr**

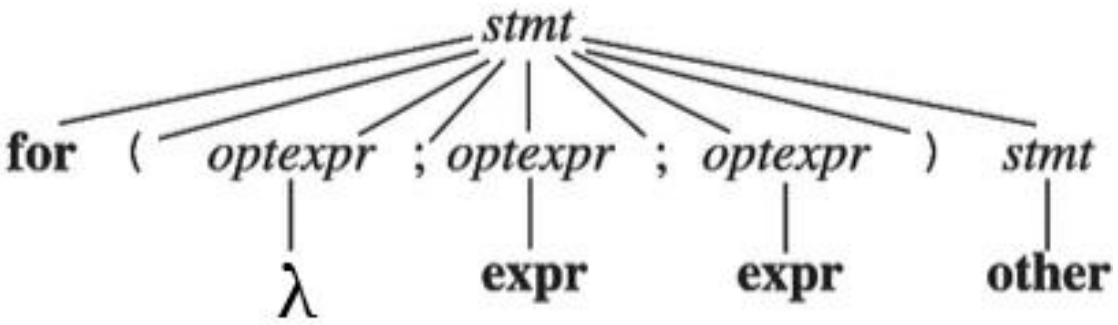
Uma gramática para alguns comandos em C e Java.

Análise Descendente

* A construção descendente de uma árvore de derivação como a da Fig. 13 a seguir é feita iniciando-se na raiz, rotulada com o não-terminal inicial *stmt*, e realizando repetidamente os **dois passos** a seguir.
  * 1) No nó N, rotulado com o não-terminal A, selecione uma das produções para A e construa filhos em N para os símbolos no corpo da produção.
  * 2) Encontre o próximo nó em que uma subárvore deve ser construída, normalmente o não-terminal não expandido mais à esquerda da árvore.

Análise Descendente

* Abaixo, Fig. 13, árvore de derivação para a gramática da Fig. 12.



Uma árvore de derivação de acordo com a gramática

Análise Descendente

* Para algumas gramáticas, os passos anteriores podem ser implementados durante uma única leitura da esquerda para a direita da cadeia da entrada.

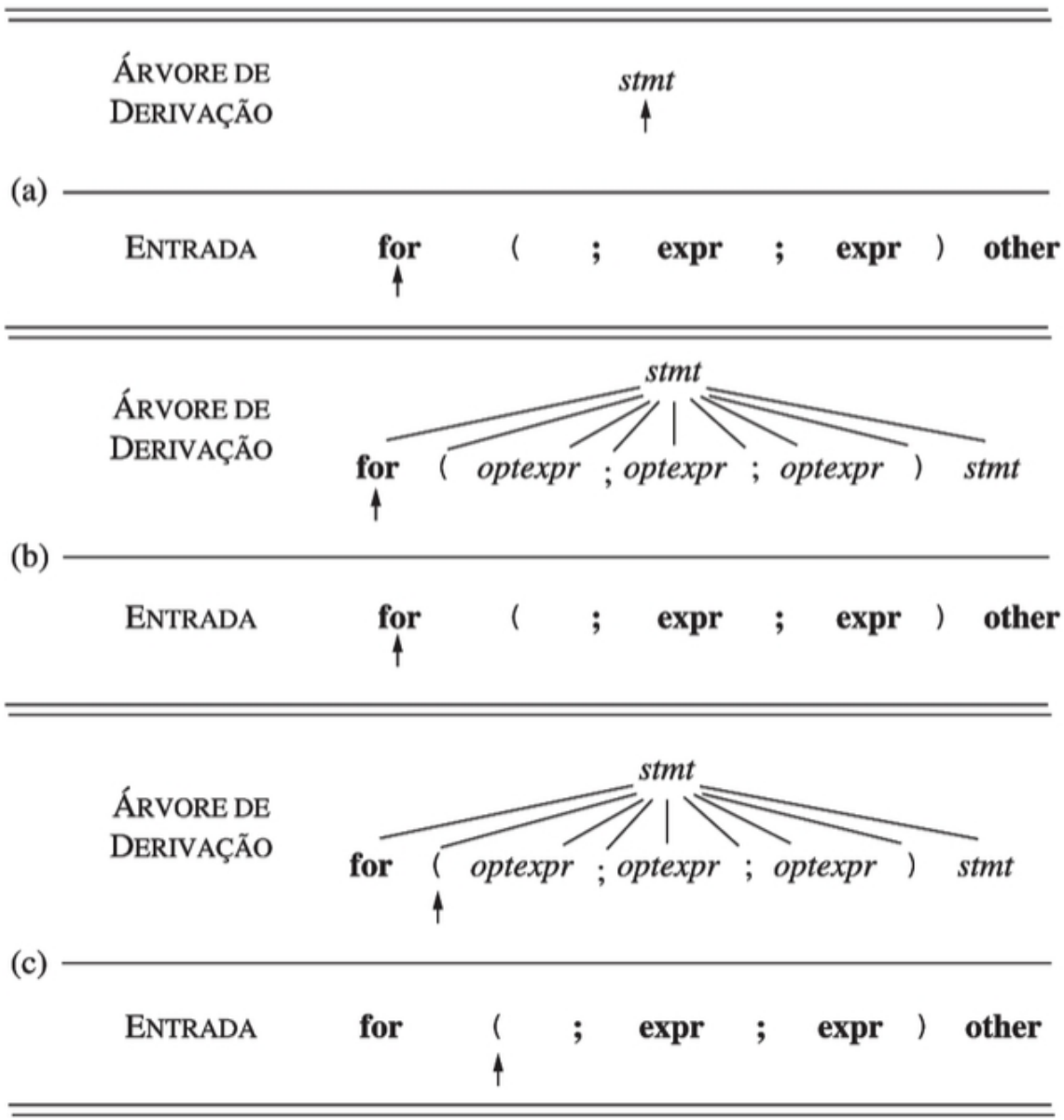
* O terminal corrente sendo lido na entrada é frequentemente referenciado como o símbolo *lookahead*. Inicialmente o símbolo *lookahead* é o primeiro terminal, ou seja, o mais à esquerda, da sequência de entrada.

Análise Descendente

* A Fig. 14 a seguir mostra a construção da árvore de derivação da Fig. 13 para a cadeia de terminais da entrada:

for (; expr ; expr) other

for (; expr ; expr) other



Análise sintática descendente processando a entrada da esquerda para a direita.

* Fig. 14

Análise Descendente

* Inicialmente, o terminal for é o símbolo *lookahead*, e a parte conhecida da árvore de derivação consiste na raiz, rotulada com o não-terminal inicial *stmt* na Fig. 14 (a).

* O objetivo é construir o restante da árvore de derivação de modo que a sequência de terminais gerada pela árvore de derivação case com a sequência da entrada.

* Para que haja um casamento, o não-terminal *stmt* na Fig. 14 (a) precisa derivar uma cadeia que comece com o símbolo *lookahead* **for**.

Análise Descendente

* Na gramática da Fig. 12, existe apenas uma produção para *stmt* que pode derivar tal cadeia, portanto a selecionamos e construímos os filhos da raiz rotulados com os símbolos no corpo da produção.

* Essa expansão da árvore de derivação aparece na Fig. 14 (b).

Análise Descendente

* Cada um dos três instantâneos da Fig. 14 possui setas identificando o símbolo *lookahead* na entrada e o nó na árvore de derivação que está sendo considerado.

* Quando os filhos são construídos em um nó, em seguida consideramos o filho mais à esquerda.

* Na Fig. 14 (b), os filhos foram construídos na raiz, e o filho mais à esquerda, rotulado com **for**, está sendo analisado.

Análise Descendente

* Quando o nó sendo considerado na árvore de derivação se refere a um terminal, e o terminal casa com o símbolo *lookahead*, avançamos na árvore de derivação e na entrada.

* Na Fig. 14 (c), a seta na árvore de derivação avançou para o próximo filho da raiz, e a seta na entrada avançou para o próximo terminal, que é (. Outro avanço levará a seta na árvore de derivação para o filho rotulado com o não-terminal *optexpr* e leva a seta na entrada para o terminal ;.

Análise Descendente

* No nó não-terminal rotulado com *optexpr*, repetimos o processo de seleção de uma produção para um não-terminal.

* As produções com λ como corpo exigem tratamento especial. Por enquanto, nós as usamos como um padrão quando nenhuma outra produção puder ser usada.

* Com o não-terminal *optexpr* e o *lookahead* ;, a produção λ é usada, pois ; não casa com a única outra produção para *optexpr*, que possui o terminal **expr** como seu corpo.

Análise Descendente

* Em geral, a seleção de uma produção para um não-terminal pode envolver tentativa-e-erro; ou seja, podemos ter de tentar uma produção e recuar para tentar outra produção se a primeira não for adequada.

* Uma produção não é adequada se, depois de usar a produção, não pudermos completar a árvore para casar com a sequência de terminais da entrada.

Análise Descendente

* Definindo formalmente, o método de análise sintática descendente constrói a árvore de derivação para a cadeia de entrada de cima para baixo, ou seja, da raiz para as folhas, criando os nós da árvore em pré-ordem (busca em profundidade, conforme discutimos anteriormente).

* Neste processo, a análise sintática descendente pode ser vista como o método que produz uma derivação mais à esquerda para uma cadeia da entrada.

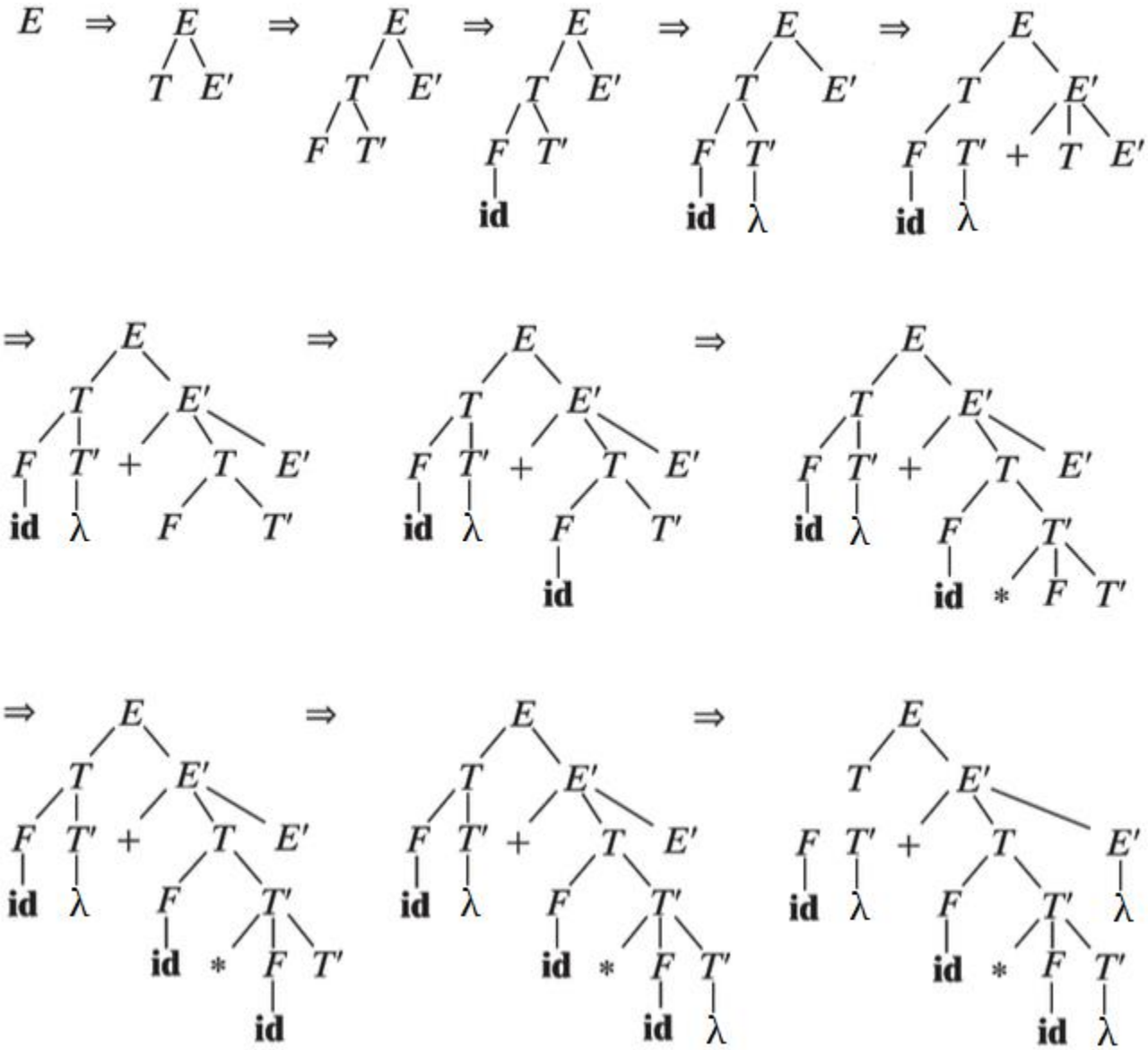
Análise Descendente

* A sequência de árvores de derivação da Fig. 15 a seguir, para a entrada **id** + **id** * **id** representa uma **análise sintática descendente** de acordo com a gramática vista anteriormente (exemplo sem recursão à esquerda), repetida aqui.

* Gramática:

$$
\begin{aligned}
E &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \lambda \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \lambda \\
F &\rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
$$

* Fig. 15
* Essa sequência de árvores corresponde a uma derivação mais à esquerda da entrada **id + id * id**.



Análise sintática descendente para **id + id * id**.

Análise Descendente

* A cada passo de uma análise sintática descendente, o problema principal é determinar a produção a ser aplicada para um não-terminal, digamos *A*.

* Quando uma produção-*A* é escolhida, o restante do processo de análise consiste em "casar" os símbolos terminais do corpo da produção com a cadeia de entrada.

* Estudaremos agora, de maneira aprofundada, um método de análise sintática descendente. Veremos uma forma geral, denominada "análise sintática de descida recursiva", que pode exigir o retrocesso para encontrar a produção-*A* correta a ser aplicada.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Um método de análise sintática de descida recursiva consiste em um conjunto de procedimentos, um para cada não-terminal da gramática.
  * A execução começa com a ativação do procedimento referente ao símbolo inicial da gramática, que pára e anuncia sucesso se o seu corpo conseguir escandir toda a cadeia de entrada.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * O pseudocódigo para um não-terminal típico aparece na Fig. 16 abaixo. Observe que esse pseudocódigo é não determinístico, pois começa escolhendo a produção-*A* a ser aplicada de uma maneira arbitrária.

```
        void A() {
1)          Escolha uma produção-A, A → X1 X2 ··· Xk;
2)          for ( i = 1 até k ) {
3)                  if ( Xi é um não-terminal )
4)                          ativa procedimento Xi();
5)                  else if ( Xi igual ao símbolo de entrada a )
6)                          avance na entrada para o próximo símbolo terminal;
7)                  else /* ocorreu um erro */;
            }
        }
```

Um procedimento típico para um não-terminal em um analisador descendente.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * O método geral de análise sintática de descida recursiva pode exigir retrocesso; ou seja, pode demandar voltar atrás no reconhecimento, fazendo repetidas leituras sobre a entrada.
  * As construções presentes nas linguagens de programação raramente necessitam de retrocesso durante suas análises, de modo que os analisadores com retrocesso não são usados com frequência.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Para permitir o retrocesso, o código da Fig. 16 anterior precisa ser modificado.
  * Primeiro, não podemos escolher uma única produção-*A* na linha (1); devemos tentar cada uma das várias alternativas da produção-*A* em alguma ordem.
  * Segundo, um erro na linha (7) não é uma falha definitiva; apenas sugere que precisamos retornar à linha (1) e tentar outra opção da produção-*A*.
  * Somente quando não houver mais produções-*A* a serem tentadas é que declaramos que foi encontrado um erro na entrada.
  * Terceiro, para tentar outra opção da produção-*A*, é necessário colocar o apontador da entrada onde ele estava quando atingimos a linha (1) pela primeira vez. Assim, é necessária a declaração de uma variável local a fim de armazenar esse apontador para uso futuro.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Vejamos um exemplo de retrocesso:
  * Considere a gramática:

$$\begin{array}{lcl} S & \rightarrow & c\,A\,d \\ A & \rightarrow & a\,b \mid a \end{array}$$

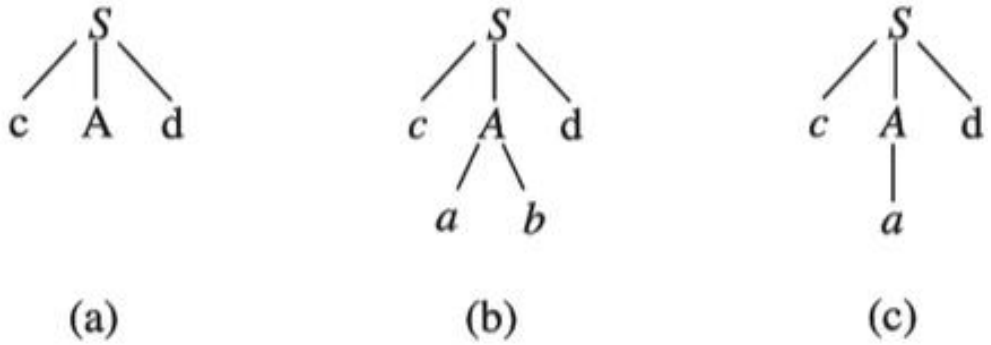
  * Para construir uma árvore de derivação descendente para a cadeia de entrada $w = cad$, comece com uma árvore consistindo em um único nó rotulado com S, e com o apontador de entrada apontando para c, o primeiro símbolo de w.

  * Como S possui apenas uma produção, nós a expandimos para obter a árvore da Fig. 17 (a) a seguir.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Vejamos um exemplo de retrocesso:
  * Fig. 17:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$



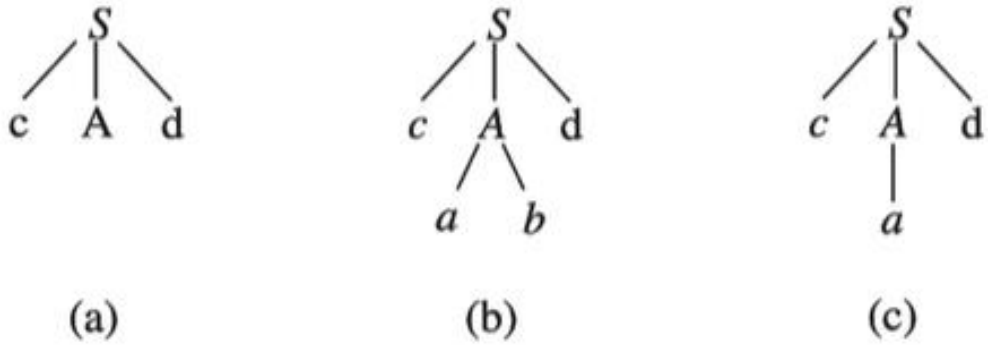
Passos de uma análise sintática descendente.

  * A folha mais à esquerda, rotulada com c, casa com o primeiro símbolo da entrada w, de modo que avançamos o apontador para a, o segundo símbolo de w, e consideramos a próximo folha, rotulada com A.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Vejamos um exemplo de retrocesso:
  * Fig. 17:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$



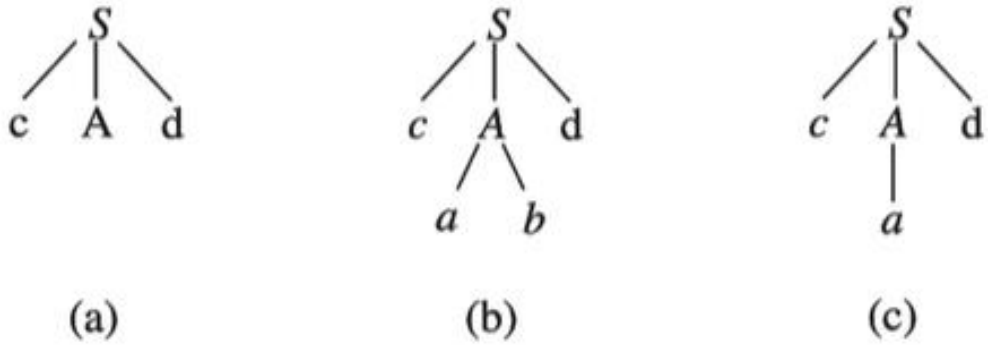
Passos de uma análise sintática descendente.

  * Agora, expandimos A usando a primeira alternativa $A \rightarrow ab$ e obtemos a árvore da Fig. 17 (b). Como temos um casamento com o segundo símbolo de entrada a, avançamos o apontador para d, o terceiro símbolo de entrada, e comparamos d com a próxima folha da árvore, rotulada com b.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Vejamos um exemplo de retrocesso:
  * Fig. 17:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$



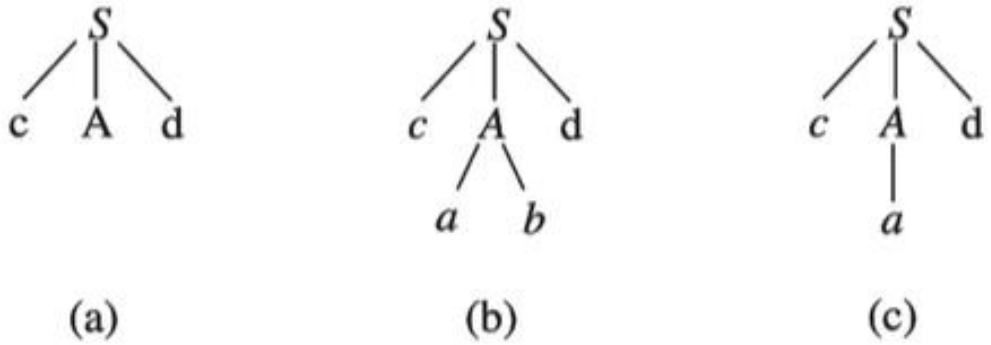
Passos de uma análise sintática descendente.

  * Como b não casa com d, informamos a falha e voltamos para A na tentativa de encontrar uma alternativa de A **ainda não explorada**, mas que poderia produzir um casamento. Ao voltar em A, é necessário reiniciar o apontador da entrada para a posição 2, que corresponde à posição em que estava quando atingimos a primeira alternativa de A.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Vejamos um exemplo de retrocesso:
  * Fig. 17:

$$S \rightarrow c\,A\,d$$
$$A \rightarrow a\,b \mid a$$



Passos de uma análise sintática descendente.

  * Isto significa que o procedimento para A precisa armazenar o apontador da cadeia de entrada em uma variável local. A segunda alternativa para A produz a árvore da Fig. 17 (c). A folha a casa com o segundo símbolo de w, e a folha d casa com o terceiro símbolo da entrada. Como produzimos uma árvore de derivação para w, **paramos e anunciamos o sucesso da análise**.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * Uma gramática recursiva à esquerda pode fazer com que um analisador de descida recursiva, até mesmo aquele com retrocesso, entre em **loop infinito**.
  * Ou seja, ao tentar expandir um não-terminal A, podemos eventualmente nos encontrar novamente tentando expandir A sem ter consumido nenhuma entrada.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * **Diagramas de Transição**
    * Os diagramas de transição são úteis para visualizar analisadores sintáticos.
    * Os diagramas de transição para analisadores sintáticos diferem daqueles para os analisadores léxicos.
    * Os analisadores sintáticos possuem um diagrama para cada não-terminal. Os rótulos das arestas podem ser tokens ou não-terminais.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * **Diagramas de Transição**
    * Uma transição sob um token (terminal) significa que efetuaremos essa transição se esse token for o próximo símbolo de entrada.
    * Uma transição sob um não-terminal A representa a ativação do procedimento A.
    * Os diagramas de transição podem ser simplificados, desde que a sequência de símbolos da gramática ao longo dos caminhos seja preservada.

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * **Diagramas de Transição**
    * Também podemos substituir o diagrama de um não-terminal A por uma aresta rotulada com A.

    * Vejamos a seguir, na Fig. 18, um exemplo de diagrama de transição para os não-terminais E e E' da gramática:

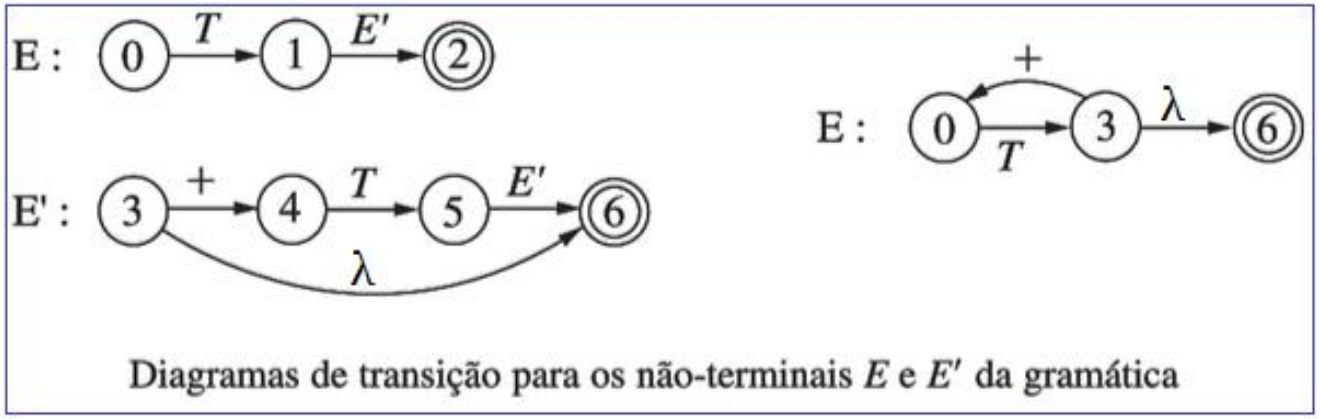
$$
\begin{array}{lll}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \lambda \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \lambda \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

Análise Descendente

* **Análise Sintática de Descida Recursiva**
  * **Diagramas de Transição**

$$E \rightarrow T\,E'$$
$$E' \rightarrow +T\,E' \mid \lambda$$

* Abaixo, Fig. 18:



Diagramas de transição para os não-terminais E e E' da gramática

Análise Descendente

- **Analisadores Sintático LL**
  - Os analisadores sintáticos de descida recursiva que não precisam de retrocesso (também chamados de analisadores sintáticos *preditivos*), podem ser construídos para uma classe de gramáticas chamada LL(1):
    - O primeiro "L" em LL significa que a cadeia de entrada é escandida da esquerda para a direita (L = *Left-to-right*).
    - O segundo "L" em LL representa uma derivação à esquerda (L = *Leftmost*).
    - O "1" pelo uso de um símbolo à frente na entrada utilizado em cada passo para tomar as decisões quanto à ação de análise.

Análise Descendente

* **Analisadores Sintático LL**
  * A classe de gramáticas LL(1) é rica o suficiente para reconhecer a maioria das construções presentes nas linguagens de programação, mas escrever uma gramática adequada para a linguagem fonte não é uma tarefa simples.
  * Por exemplo, nenhuma gramática com recursão à esquerda ou ambígua pode ser LL.
  * A vantagem é que um analisador LL(1) é bastante fácil de construir, e muito eficiente. Por esse motivo, será nosso foco.

Análise Descendente

* **Analisadores Sintático LL**
  * Embora a eliminação de recursão à esquerda e a fatoração à esquerda sejam fáceis de fazer, devemos observar que algumas gramáticas livres de contexto não tem gramáticas equivalentes LL(1), e nesse caso nenhuma alteração produzirá uma gramática LL(1).

A linguagem gerada pela gramática do "else vazio", por exemplo, ***não*** *possui uma gramática LL(1) para ela. Apesar da recursão à esquerda ter sido eliminada e a gramática ter sido fatorada à esquerda, a gramática é ambígua.*

$$
\begin{array}{lcl}
S & \rightarrow & iEtSS' \mid a \\
S' & \rightarrow & eS \mid \lambda \\
E & \rightarrow & b
\end{array}
$$

Análise Descendente

* **Analisadores Sintático LL**
  * Gramáticas LL(1) podem ser analisadas por um simples *parser* descendente recursivo:
    * Sem recursão à esquerda
    * Fatorada à esquerda
    * Com um símbolo *lookahead*
  * Nem todas as linguagens podem ser tornadas LL(1).
  * Ainda se pode usar um mecanismo mais poderoso para reconhecer tais gramáticas:
    * Eliminar a recursividade.

Análise Descendente

* **Analisadores Sintático LL**
  * Os analisadores sintáticos de descida recursiva que não precisam de retrocesso podem ser construídos para as gramáticas LL, pois é possível selecionar a produção apropriada a ser aplicada para um não-terminal examinando-se apenas o símbolo corrente da entrada restante.

  * Construções de fluxo de controle, com suas distintas palavras-chave, geralmente satisfazem as restrições das gramáticas LL.

Análise Descendente

* **Analisadores Sintático LL**
  * Por exemplo, se tivermos as produções:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if} \ (\ \textit{expr}\ )\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
& | & \textbf{while}\ (\ \textit{expr}\ )\ \textit{stmt} \\
& | & \{\ \textit{stmt\_list}\ \}
\end{array}
$$

  * então as palavras-chave **if**, **while** e o símbolo **{** nos dizem qual é a única alternativa que possivelmente poderia ser bem-sucedida se estivermos procurando por um *stmt*.

Análise Descendente

* **Analisadores Sintático LL**
  * Conceitualmente, o analisador LL(1) constrói uma derivação mais à esquerda para o programa, partindo do símbolo inicial.
  * A cada passo da derivação, o prefixo de terminais da forma sentencial tem que casar com um prefixo da entrada.
  * Caso exista mais de uma regra para o não-terminal que vai gerar o próximo passo da derivação, o analisador usa o primeiro token após esse prefixo para escolher qual regra usar.
  * Esse processo continua até todo o programa ser derivado ou acontecer um erro (o prefixo de terminais da forma sentencial não casa com um prefixo do programa).

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Vejamos uma gramática LL(1) simples:

```
PROG -> CMD ; PROG
PROG -> λ
CMD -> id = EXP
CMD -> print EXP
EXP -> id
EXP -> num
EXP -> ( EXP + EXP )
```

  * Vamos analisar: **id = (num + id) ; print num ;**

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * O terminal entre || é o *lookahead*, usado para escolher qual regra usar.

  * *PROG* (Variável de Partida)

|id| = (num + id) ; print num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * O terminal entre || é o *lookahead*, usado para escolher qual regra usar.

  * *PROG -> CMD ; PROG*

|id| = (num + id) ; print num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG*

id = |(| num + id) ; print num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ; *PROG*

id = (|num| + id) ; print num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ;
    PROG -> id = (num + *EXP*) ; *PROG*

id = (num + |id|) ; print num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ;
    PROG -> id = (num + *EXP*) ; *PROG* -> id = (num + id) ; *PROG*

id = (num + id) ; |print| num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ; *PROG* -> id = (num + *EXP*) ; *PROG* -> id = (num + id) ; *PROG* -> id = (num + id) ; *CMD* ; *PROG*

id = (num + id) ; |print| num ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * Um prefixo de terminais na forma sentencial desloca o *lookahead*.

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ; *PROG* -> id = (num + *EXP*) ; *PROG* -> id = (num + id) ; *PROG* -> id = (num + id) ; *CMD* ; *PROG* -> id = (num + id) ; print *EXP* ; *PROG*

id = (num + id) ; print |num| ;

Análise Descendente

* **Analisadores Sintático LL**
  * Exemplo:
  * O *lookahead* agora está no final da entrada (EOF – End of File)

  * *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ; *PROG* -> id = (num + *EXP*) ; *PROG* -> id = (num + id) ; *PROG* -> id = (num + id) ; *CMD* ; *PROG* -> id = (num + id) ; print *EXP* ; *PROG* -> id = (num + id) ; print num ; *PROG*

id = (num + id) ; print num ; ||

Análise Descendente

- **Analisadores Sintático LL**
  - Exemplo:
  - Chegamos em uma derivação para o programa, sucesso!

  - *PROG* -> *CMD* ; *PROG* -> id = *EXP* ; *PROG* -> id = (*EXP* + *EXP*) ; *PROG* -> id = (num + *EXP*) ; *PROG* -> id = (num + id) ; *PROG* -> id = (num + id) ; *CMD* ; *PROG* -> id = (num + id) ; print *EXP* ; *PROG* -> id = (num + id) ; print num ; *PROG* -> id = (num + id) ; print num ;

id = (num + id) ; print num ; ||

Análise Descendente

* **Gerador de Analisador Sintático LL**
* **JavaCC**
  * JavaCC (Java Compiler Compiler) é um gerador de analisador sintático aberto para a linguagem Java.
  * É similar ao YACC na medida em que gera um analisador sintático a partir de uma gramática fornecida, exceto pelo fato da saída ser em Java.

Análise Descendente

* **Gerador de Analisador Sintático LL**
* **JavaCC**
  * O JavaCC gera analisadores sintáticos descendentes, o que o limita às classes de gramáticas LL (excluindo, por exemplo, recursividade à esquerda).
  * Além do próprio gerador de analisador sintático, o JavaCC fornece outros recursos padrões relacionados à geração de analisadores, como a construção de árvore (por meio de uma ferramenta chamada JJTree incluída no JavaCC), ações, depuração, etc.

Análise Ascendente

* A análise sintática **ascendente** (bottom-up) corresponde à construção de uma árvore de derivação para uma cadeia de entrada a partir das folhas (a parte de baixo) em direção à raiz (o topo) da árvore.

* Embora a árvore de derivação seja utilizada para descrever os métodos de análise, um *front-end* pode executar uma tradução diretamente, portanto, na prática, ela nunca é efetivamente construída.

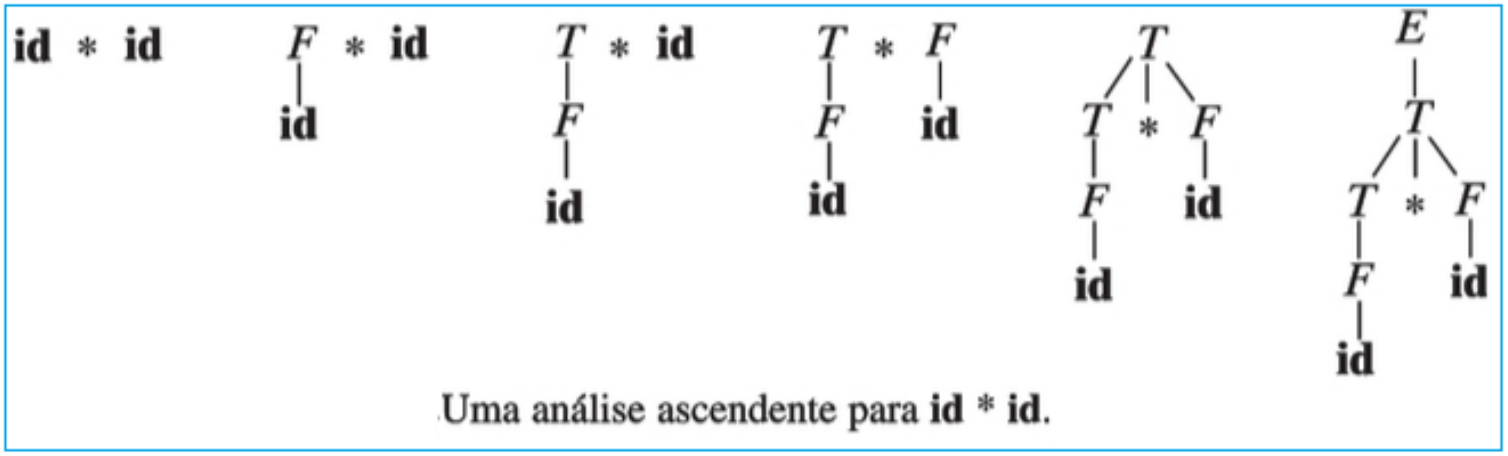
Análise Ascendente

* A sequência de instantâneos de árvore na Fig. 19 ilustra o reconhecimento ascendente da sequência de tokens **id** * **id**, com respeito à gramática de expressão vista anteriormente:

Gramática:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

* Abaixo, Fig. 19:



Uma análise ascendente para **id** * **id**.

Análise Ascendente

* A maior classe de gramáticas para a qual os analisadores sintáticos ascendentes podem ser construídos, as gramáticas LR, será discutida adiante.

* Embora seja muito trabalhoso construir um analisador sintático LR à mão, ferramentas chamadas geradores automáticos de analisadores sintáticos facilitam a construção de analisadores LR eficientes a partir de gramáticas adequadas.

* Veremos alguns conceitos importantes para a definição de gramáticas apropriadas para fazer uso efetivo de um gerador de analisadores LR.

Análise Ascendente

* **Reduções**
  * Podemos pensar na análise ascendente como o processo de "reduzir" uma cadeia w para o símbolo inicial da gramática.
  * Em cada passo da *redução*, uma subcadeia específica, casando com o lado direito de uma produção, é substituída pelo não-terminal na cabeça dessa produção.
  * As principais decisões relacionadas com a análise ascendente em cada passo do reconhecimento são: determinar quando reduzir e determinar a produção a ser usada para que a análise prossiga.

Análise Ascendente

* **Reduções**
  * Por definição, uma redução é o inverso de um passo em uma derivação (lembre-se, em uma derivação, um não-terminal em uma forma sentencial é substituído pelo lado direito de uma de suas produções).

  * O objetivo do método de análise ascendente é, portanto, construir uma derivação ao reverso.

  * A derivação a seguir corresponde à análise da Fig. 19:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

  * Essa derivação é, na verdade, uma derivação mais à direita.

Análise Ascendente

* **Analisadores Sintático LR**
  * Atualmente, o tipo mais prevalente de analisadores sintáticos ascendentes é baseado em um conjunto chamado reconhecedores LR(k):
    * O "L" em LR significa que a cadeia de entrada é escandida da esquerda para a direita (L = *Left-to-right*).
    * O "R" em LR representa uma derivação à direita ao reverso (R = *Rightmost*).
    * O "k" representa os símbolos à frente no fluxo de entrada que auxiliam nas decisões de análise.

Análise Ascendente

* **Analisadores Sintático LR**
  * Na prática, $k = 0$ ou $k = 1$ é suficiente, e só vamos considerar analisadores sintáticos LR com $k <= 1$.
  * Quando (k) é omitido, k é considerado como sendo 1.

Análise Ascendente

* **Analisadores Sintático LR**
  * LR = (**L**eft to right with **R**igthmost derivation)
  * Informalmente, podemos dizer que os analisadores LR:

"São analisadores redutores que leem a sentença de entrada da esquerda para a direita e produzem uma derivação mais à direita ao reverso, considerando k símbolos sob o cabeçote de leitura."

Análise Ascendente

* **Analisadores Sintático LR**
  * Os reconhecedores sintáticos LR são atraentes por diversos motivos:

  * 1) Os analisadores LR são capazes de reconhecer praticamente todas as construções sintáticas definidas por gramáticas livres de contexto da maioria das linguagens de programação. Existem gramáticas livres de contexto que não são LR, mas geralmente elas podem ser evitadas para construções típicas das linguagens de programação.
  * 2) O LR, além de ser o método de análise sem retrocesso mais geral, pode ser implementado com o mesmo grau de eficiência, espaço e tempo, que outros métodos.
  * 3) Um analisador sintático LR detecta um erro sintático tão logo ele aparece na cadeia de entrada em uma escansão da entrada da esquerda para a direita.

Análise Ascendente

* **Analisadores Sintático LR**
  * Os reconhecedores sintáticos LR são atraentes por diversos motivos:

  * 4) A classe de gramáticas que podem ser reconhecidas usando os métodos LR é um superconjunto próprio da classe de gramáticas que podem ser reconhecidas com os métodos que não precisam de retrocesso ou LL. Para uma gramática ser LR(k), ela deve ser capaz de reconhecer a ocorrência do lado direito de uma produção em uma forma sentencial mais à direita, com k símbolos à frente na entrada. Esse requisito é muito menos rigoroso do que aquele para as gramáticas LL(k), onde; para reconhecer o uso de uma produção o analisador vê apenas os k primeiros símbolos que o seu lado direito deriva. Assim, não é surpresa que as gramáticas LR possam descrever mais linguagens do que as gramáticas LL.

Análise Ascendente

* **Analisadores Sintático LR**
  * A principal desvantagem do método LR está relacionada com a geração do analisador: sua construção à mão, para uma gramática de linguagem de programação típica, é muito trabalhosa.
  * Portanto, é necessário o uso de uma ferramenta especializada, um gerador de analisadores LR.
  * Felizmente, muitos desses geradores estão disponíveis, como o muito utilizado gerador "Yacc" (*yet another compiler-compiler*).
    * Esse gerador recebe como entrada uma gramática livre de contexto e produz automaticamente como saída um analisador sintático para essa gramática. Se a gramática possui ambiguidades ou outras construções difíceis de analisar em uma escansão da esquerda para a direita, então o gerador de analisador sintático localiza essas construções e disponibiliza mensagens com diagnósticos detalhados.

Análise Ascendente

* **Gerador de Analisador Sintático LR**
* **Yacc – Yet Another Compiler-Compiler**
  * Trata-se de um gerador de analisador sintático, no qual gera o analisador sintático baseado na gramática livre de contexto.
  * Como este foi projetado inicialmente para usar no sistema operacional Unix, o YACC costumava ser o gerador de analisador sintático padrão na maioria dos sistemas Unix, mas acabou sendo suplantado por versões mais modernas ainda que compatíveis.

Análise Ascendente

* **Gerador de Analisador Sintático LR**
* **Yacc – Yet Another Compiler-Compiler**
  * O analisador sintático gerado pelo Yacc requer um analisador léxico, que pode ser fornecido externamente através de geradores de analisadores léxicos como o *Lex* ou o *Flex*.
  * A norma POSIX (sugestão de pesquisa) define a funcionalidade e os requisitos tanto para *Lex* quanto para Yacc.

Análise Ascendente

* **Gerador de Analisador Sintático LR**
* **Yacc – Yet Another Compiler-Compiler**
  * A utilização do Yacc geralmente é em conjunto com o gerador de analisador léxico *Lex*, no qual o Yacc usa uma gramática formal para analisar sintaticamente uma entrada, algo que o *Lex* não consegue fazer somente com expressões regulares (o *Lex* é limitado a simples máquinas de estado finito).

Análise Ascendente

* **Gerador de Analisador Sintático LR**
* **Yacc – Yet Another Compiler-Compiler**
  * Entretanto, o Yacc não consegue ler a partir de uma simples entrada de dados, ele requer uma série de tokens, que são geralmente fornecidos pelo *Lex*.
  * O *Lex* age como um pre-processador do Yacc.

Análise Ascendente

* **LL Versus LR**
  * Se você pretende escrever o analisador sintático (*parser*) você mesmo, sem a ajuda de nenhuma ferramenta, tenda imediatamente a escolher um LL, especialmente se a linguagem for regular ou simples o suficiente para LL(1). É um modelo fácil de programar e na maioria dos casos a diferença de performance é muito pequena para ser relevante. É uma boa escolha para quem está começando por permitir entender o que está acontecendo no código com facilidade.

  * Mas, se você planeja um *parser* de maior complexidade, ou que a eficiência é importante, o ideal é confiar em uma ferramenta para gerar o *parser* para você. Na maioria dos casos ela será capaz de fazer simplificações na gramática que a tornam ilegível no código final do *parser*, porém mais performática.

Análise Ascendente

* **LL Versus LR**
  * A primeira forma de análise LL:
    * Lê a entrada da esquerda para a direita;
    * Realiza uma busca *top-down*, isto é, parte-se do símbolo inicial (raiz) em direção às folhas (terminais) da árvore de derivação;
    * Constrói uma derivação mais à esquerda.

  * Já a segunda forma de análise LR:
    * Também lê a entrada da esquerda para a direita;
    * Realiza uma busca *bottom-up*, isto é, parte-se das folhas em direção à raiz da árvore;
    * Constrói uma derivação mais à direita, embora a leitura também seja feita a partir da esquerda.

Resumo

* ***Analisadores sintáticos***: Um analisador sintático recebe como entrada tokens do analisador léxico e trata os nomes de tokens como símbolos terminais de uma gramática livre de contexto. O analisador, então, constrói uma árvore de derivação para as sequências de tokens da entrada; a árvore de derivação pode ser construída de modo figurativo (passando pelos passos de derivação correspondentes) ou literal.

* ***Gramáticas livres de contexto***: Uma gramática especifica um conjunto de símbolos terminais (entradas), um conjunto de não-terminais (símbolos representando construções sintáticas) e um conjunto de produções, cada qual especificando uma forma com que cadeias representadas por um não-terminal podem ser construídas a partir dos símbolos terminais e cadeias representadas por certos não-terminais. Uma produção consiste em um lado esquerdo ou cabeça (o não-terminal a ser substituído) e um lado direito, também denominado corpo da produção (a cadeia de substituição de símbolos da gramática).

Resumo

* ***Derivações***: O processo que começa com o não-terminal inicial de uma gramática e o substitui sucessivamente pelo lado direito de uma de suas produções é chamado de derivação. Se o não-terminal mais à esquerda (ou mais à direita) sempre for substituído, então a derivação é chamada derivação mais à esquerda (respectivamente, derivação mais à direita).

* ***Árvores de derivação***: Uma árvore de derivação é uma imagem de uma derivação, na qual existe um nó para cada não-terminal que aparece na derivação. Os filhos de um nó são os símbolos pelos quais esse não-terminal é substituído no processo de derivação. Existe uma correspondência um-para-um entre as árvores de derivação, derivações mais à esquerda e derivações mais à direita da mesma cadeia de terminais.

Resumo

* ***Ambiguidade***: Uma gramática para a qual a mesma cadeia de terminais possui duas ou mais árvores de derivação diferentes, ou, de forma equivalente, duas ou mais derivações mais à esquerda ou duas ou mais derivações mais à direita, é considerada ambígua. Na prática, na maioria dos casos é possível reprojetar uma gramática ambígua de modo que ela se torne uma gramática não-ambígua para a mesma linguagem. No entanto, as gramáticas ambíguas, com certos truques aplicados, às vezes produzem analisadores sintáticos mais eficientes.

* ***Análise descendente e ascendente***: Os analisadores sintáticos geralmente são classificados em analisadores sintáticos descendentes (começando com o símbolo inicial da gramática e construindo a árvore de derivação de cima para baixo) ou ascendentes (começando com os símbolos terminais que formam as folhas da árvores de derivação construindo a árvore de baixo para cima). Os analisadores descendentes incluem analisadores de descida recursiva e LL, enquanto as formas mais comuns de analisadores ascendentes são os analisadores LR.

Resumo

* ***Projeto de gramáticas***: As gramáticas adequadas para a análise descendente normalmente são mais difíceis de projetar do que aquelas usadas por analisadores ascendentes. É necessário eliminar a recursão à esquerda, uma situação na qual um não-terminal deriva uma cadeia de símbolos da gramática que começa com o mesmo não-terminal. Também devemos fatorar à esquerda – agrupar produções para o mesmo não-terminal que possuem um prefixo comum no corpo.

* ***Analisadores de Descida Recursiva***: Esses analisadores utilizam um procedimento para cada não-terminal. O procedimento examina sua entrada e decide qual produção aplicar ao não-terminal. Os terminais do lado direito da produção são casados com a entrada no momento apropriado, enquanto os não-terminais do lado direito resultam em chamadas ao seu procedimento. O retrocesso é usado em situações nas quais uma produção errada foi escolhida, representando uma possibilidade de retorno.

Resumo

* ***Analisadores LL(1)***: Uma gramática tal que seja possível escolher a produção correta com a qual expandimos determinado não-terminal, examinando apenas o próximo símbolo da entrada, é chamada de LL(1). Essas gramáticas nos permitem construir uma tabela de reconhecimento preditivo que fornece, para cada não-terminal e cada símbolo *lookahead*, a escolha correta da produção. A correção do erro pode ser facilitada colocando-se rotinas de erro em algumas ou todas as entradas da tabela que não possuem uma produção legítima.

* ***Analisadores LR***: Cada um dos vários tipos de analisadores LR opera construindo primeiro o conjunto de itens válidos (chamados estados LR) para todos os prefixos viáveis possíveis e registrando o estado de cada prefixo na pilha. O conjunto de itens válidos orienta a decisão de análise.

Resumo

* ***Análise sintática ascendente de gramáticas ambíguas***: Em muitas situações importantes, como a análise de expressões aritméticas, podemos usar uma gramática ambígua e explorar informações adicionais, como a precedência de operações, para resolver conflitos entre transferência e redução, ou entre redução por duas produções diferentes. Assim, as técnicas e análise LR se estendem a muitas gramáticas ambíguas.

* ***Yacc***: O gerador de analisador sintático Yacc recebe como entrada uma gramática (possivelmente) ambígua e informações sobre a solução de conflitos, e constróis os estados LR. Depois, ele produz uma função que usa esses estados para realizar uma análise ascendente e chamar uma função associada toda vez que uma redução for realizada.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos