**1)** Explica detalhadamente e exemplifica como funciona os analisadores sintáticos descendentes:

**Prefácio da resposta:**

Os analisadores sintáticos descendentes, podem ser separados em duas categorias, **ASD** com retrocesso e **ASD** preditiva. A **ASD** preditiva pode ser ou não recursiva

**ASD retrocesso:**

Significa que, se uma derivação de uma produção falhar, o analisador de sintaxe reinicia o processo usando regras diferentes da mesma produção. Esta técnica pode processar a string de entrada mais de uma vez para determinar a produção correta. O analisador superior começa a partir do nó raiz (símbolo de início) e compara a string de entrada com as regras de produção para substituí-las (se houver). Entretanto, há desvantagens relacionadas ao mesmo, já que, quanto ao seu processamento é igualado a um processo exponencial, isto é, explora todas as regras, portanto este processo é concluído quando a cadeia é reconhecida ou quando as não há mais regras e a cadeia não é reconhecida.

**ASD Preditivo não recursivo:**

Esta forma de análise que não requer nenhum rastreamento posterior e é conhecida como técnica de análise de tabela, uma vez que que é construído uma tabela para a string a ser analisada. O **ASD** Preditivo não recursivo tem capacidade de prever qual produção será usada para substituir a sequência de entrada. Para realizar suas tarefas, o analisador preditivo usa um ponteiro de antecipação, que aponta para os próximos símbolos de entrada. Para tornar o rastreamento retroativo do analisador, o analisador preditivo impõe algumas restrições à gramática e aceita apenas uma classe de gramática conhecida como gramática LL (left-left), ou seja mais a esquerda.

**ASD Preditivo Recursivo:**

Refere-se a uma análise **de cima para baixo** em uma sintaxe de um conjunto de procedimentos recursivos usados para processar a entrada. Um procedimento está associado a cada não terminal de uma gramática. Uma forma simples de **ASD** preditivo recursivo, em que o símbolo de antecipação determina de forma inadequada o fluxo de controle através do corpo do procedimento para cada não terminal. A sequência de chamadas de procedimento durante a análise de uma string de entrada define indiretamente uma árvore de análise para entrada e pode ser usada para construir uma árvore de análise explícita, caso necessário. Neste momento, o analisador pode ter mais de uma produção para escolher para uma única instância de entrada, onde o conceito de retrocesso tem início.

**Exemplo preditivo:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Antes de remover a recursão à esquerda | |  |  | | --- | --- | | Depois de remover a recursão | | | à esquerda |  | |
| **E** ➝ **E** + **T** | **T**  **T** ➝ **T** \* **F** | **F**  **F** ➝ (**E**) | id | **E** ➝ **TE'**  **E'** ➝ +**TE**' | Є  **T** ➝ **FT'**  **T'** ➝ \***FT** ' | Є  **F** ➝ (**E**) | id |

**2)** Sobre análise sintática descendente, explique como funciona e escreva 3 exemplos com o uso do algoritmo First e Follow

**Prefácio da resposta**:

Quando uma string de entrada (código-fonte ou um programa em alguma linguagem) é fornecida a um compilador, o compilador a processa em várias fases, começando com a [análise lexical](http://quiz.geeksforgeeks.org/compiler-lexical-analysis/) (verifica a entrada e a divide em tokens) para gerar o código de destino.

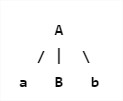
A análise sintática é a segunda fase, ou seja, após a análise lexical. Ele verifica a estrutura sintática da entrada fornecida, ou seja, se a entrada fornecida está na sintaxe correta (da linguagem em que a entrada foi escrita) ou não. Ele faz isso construindo uma estrutura de dados, chamada de árvore de análise ou árvore de sintaxe. A árvore de análise é construída usando a gramática predefinida do idioma e a string de entrada. Se a string de entrada fornecida pode ser produzida com a ajuda da árvore de sintaxe (no processo de derivação), a string de entrada está na sintaxe correta, caso contrário, o erro é relatado pelo analisador de sintaxe.

O algoritmo First torna-se necessário, já que, simplifica o processo de retrocesso de um **AS**. Isto é, “o compilador pode identificar o primeiro símbolo", logo pode definir sabiamente qual regra aplicar com base no token da cadeia de símbolos de entrada.

Entretanto ainda há mais uma falha, mesmo utilizando First. Ex. Considerando a gramática a seguir:

1. ➝ a**B**b
2. ➝ c | ε

E suponha que a string de entrada seja “**ab**” para analisar.



Agora, o analisador verifica o segundo caractere da string de entrada, que é **b**, e o não terminal a ser derivado é **B**, mas o analisador não pode obter nenhuma string derivável de **B** que contenha um símbolo de entrada que preencha o espaço entre ‘**a’** e ‘**b’,** ou seja **acb**. Está gramática contém uma regra de produção **B** ➝ ε, se for aplicada então **B** desaparecerá, e o analisador obterá a entrada “**ab**”, como mostrado abaixo. Mas o analisador pode aplicá-lo apenas quando sabe que o caractere que segue **B** na regra de produção é o **(b)** o mesmo caractere atual na entrada. Ou seja, **FOLLOW**(**B**) = {**b**}, e o caractere de entrada atual lido também é **b**. Portanto, o analisador aplica esta regra. E é capaz de obter a string “ab” da gramática fornecida.

**1° Exemplo:**

O algoritmo **First** analisa uma variável, e a partir dessa variável, se tentarmos controlar todas as strings, o símbolo do terminal inicial é chamado de primeiro.

Então se assumirmos que **FI(X)** para um símbolo gramatical **X** é o conjunto de terminais que começam as cadeias deriváveis de **X**.

**Regras do FI:**

* 1. **Se x** for um terminal, então **FI(x)** = {'x'}
  2. **Se x**➝ Є, é uma regra de produção, então adicione Є a **FI(x)**.
  3. **Se X**➝ **Y1, Y2, Y3… Yn** for uma produção, **FI(X)** = **FI(Y1)**
  4. **Se FI(Y1)** contém Є, então **FI(X)** = {**FI(Y1)** - Є} **U** {**FI(Y2)**}
  5. **Se FI(Yi)** contém Є para todo i = 1 a n, então adicione Є a **FI(X).**

**Regras da gramática:**

E ➝ TE'

1. '➝ +TE' | Є T ➝ FT'

T' ➝ \* FT' | Є

1. ➝ (E) | id

**Então:**

**FI** (E) = **FI** (T) = { ( , id } Nota-se que '(' existe por causa da 5ª regra

**FI** (E') = { + , Є }

**FI** (T) = **FI** (F) = { ( , id }

**FI** (T') = { \* , Є } **FI** (F) = { ( , id }

**2° Exemplo:**

O algoritmo **Follow** analisa o símbolo terminal que segue uma variável no processo de derivação.

Então se assumirmos que **FO(X)** para ser o conjunto de terminais que podem aparecer imediatamente à direita do Não-Terminal **X** em alguma forma sentencial.

**Regras do FO:**

* 1. **FO(S**) = {**$**} onde **$** representa os símbolos iniciais.
  2. **Se** A -> p**B**q é uma produção, onde p, **B** e q são quaisquer símbolos gramaticais, então tudo em **FI**(q) exceto Є está em **FO(B)**.
  3. **Se A** ➝ p**B** for uma produção, então tudo em **FO(A)** está em **FO(B)**.
  4. **Se A** ➝ p**B**q é uma produção e **FI**(q) contém Є, então **FO(B)** contém { **FI**(q) Є} **U** {**FO(A)**}

**Regras da gramática:**

E ➝ TE'

1. '➝ +TE' | Є T ➝ FT'

T' ➝ \* FT' | Є

1. ➝ (E) | id **Então:**

**FO**(E) = { **$** , ) } Nota-se que ')' existe por causa da 5ª regra

**FO**(E’) = **FO**(E) = { **$**, ) }

**FO**(T) = { **FI**(E’) – Є } U **FO**(E’) **U FO**(E) = { + , **$** , ) }

**FO**(T’) = **FO**(T) = { + , **$** , ) }

**FO**(F) = { **FI**(T’) – Є } U **FO**(T’) U **FO**(T) = { \*, +, **$**, ) }

Observando as duas funções na mesma gramática é possível notar que ambas complementam-se para que as regras sejam aplicadas de forma assertiva.

**3° Exemplo:**

**Regras da gramática:**

**S** ➝ **ACB**| **C**bb | **B**a

1. ➝ da | **BC**
2. ➝ g | Є
3. ➝ h | Є

**Então:**

**First:**

**FI(S)** = **FI(A) U FI(B) U FI(C)** = { d, g, h, Є, b, a}

**FIFI** = { d } **U** { **FI(B)** - Є } **U FI(C)** = { d, g, h, Є }

**FI(B)** = { g, Є }

**FI(C)** = { h, Є }

**Follow:**

**FO(S)** = { **$** }

**FO(A)** = { h, g, **$** }

**FO(B)** = { a, **$**, h, g }

**FO(C)** = { b, g, **$**, h }