Universidade de Santa Cruz do Sul – UNISC
Departamento de informática

COMPILADORES

Análise sintática

Parte 01

Prof. Geovane Griesang
geovanegriesang@unisc.br

Análise sintática

Continuação... Próxima aula

08/04/2013	3. Análise Sintática: 3.1 analisadores ascendentes e descendentes;
15/04/2013	3. Análise Sintática: 3.2 análise preditiva;
22/04/2013	3. Análise Sintática: 3.3 análise de precedência de operadores;
29/04/2013	3. Análise Sintática: 3.4 análise LR;
06/05/2013	3. Análise Sintática: 3.5 recuperação de erros;
13/05/2013	3. Análise Sintática: 3.6 geradores de analisadores sintáticos;
20/05/2013	- Prova Individual e sem consulta 2.

Análise sintática

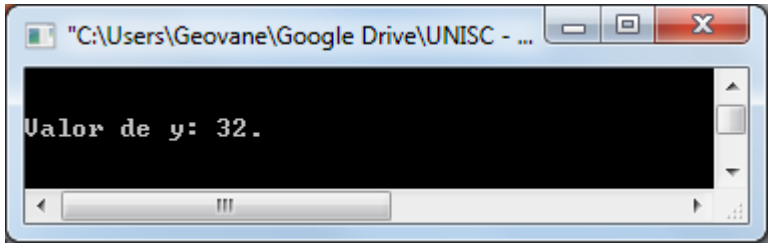
Análise léxica x Análise sintática

Exemplo de código-fonte em C:

```
y = x + 30;
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int x, y;
  x = 2;
  y = x + 30;
  printf("\n\nValor de y: %d.\n\n",y);
}
```



Análise sintática

Análise léxica x Análise sintática

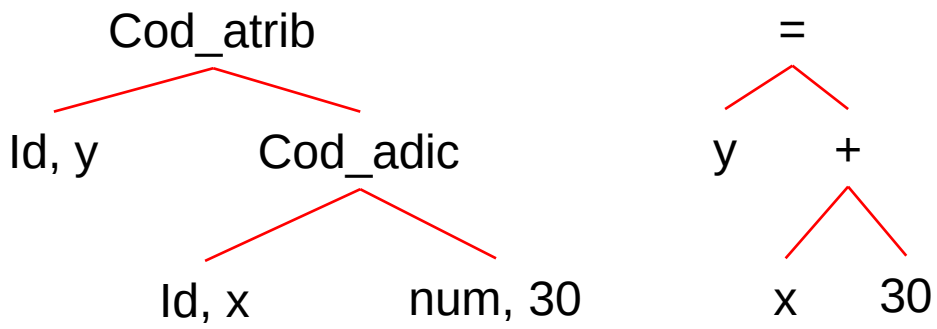
Exemplo de código-fonte em C:

Programa fonte: y = x + 30

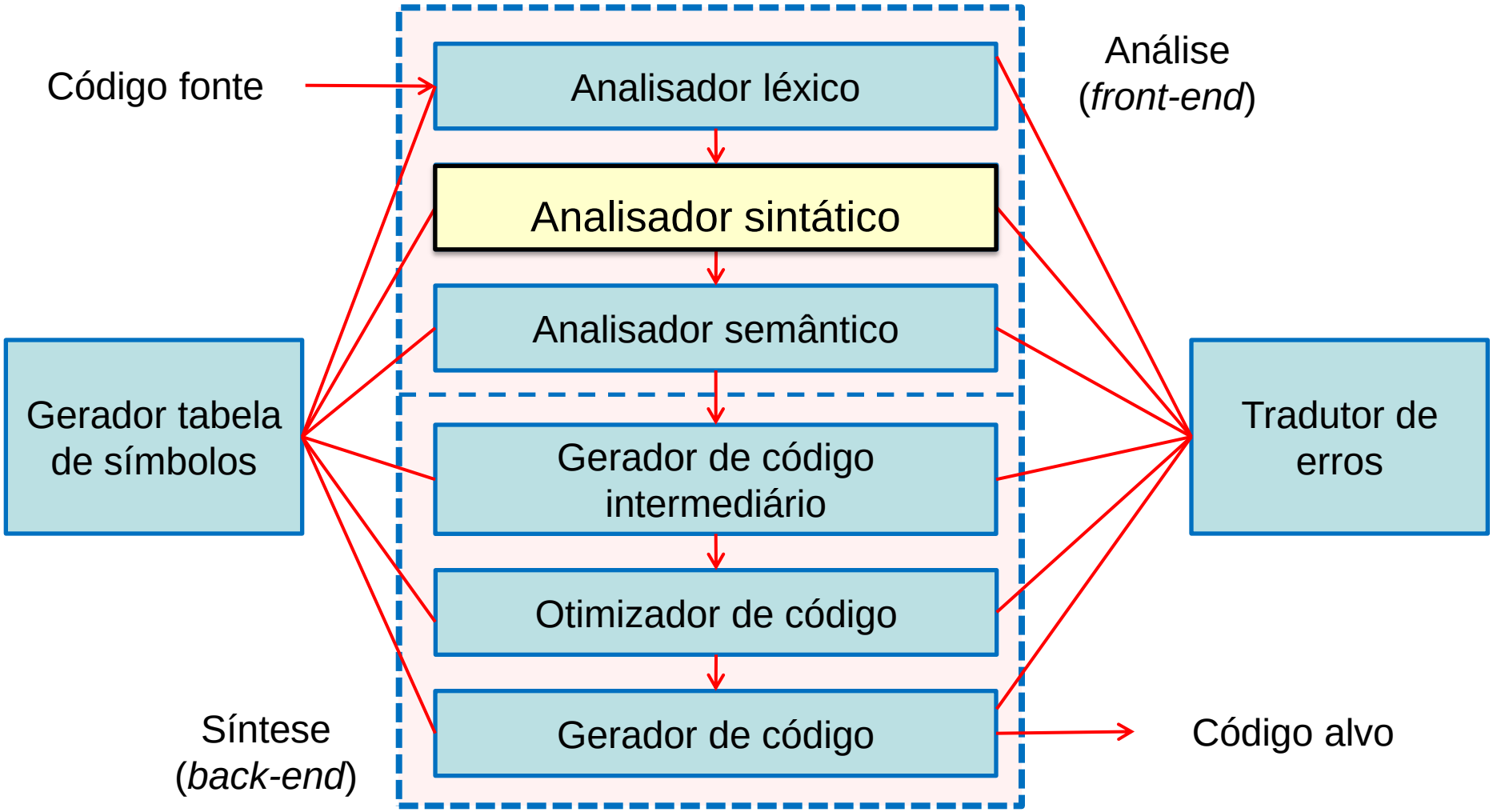
Análise léxica:

id	Cod_atrib	id	Cod_adic	num
Y	=	x	+	30

Análise sintática:



Análise sintática



Análise sintática

O Analisador Sintático (AS) obtém uma cadeia de *tokens* proveniente do analisador léxico e verifica se o mesmo pode ser gerado pela gramática da linguagem-fonte.

Espera-se que o analisador sintático relate quaisquer erros de sintaxe e também recupere erros que ocorrem mais comumente, afim de poder continuar processando o resto de sua entrada.

Portanto, a analisador sintático 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.

Verifica se as construções utilizadas no programa-fonte estão sintaticamente corretas.

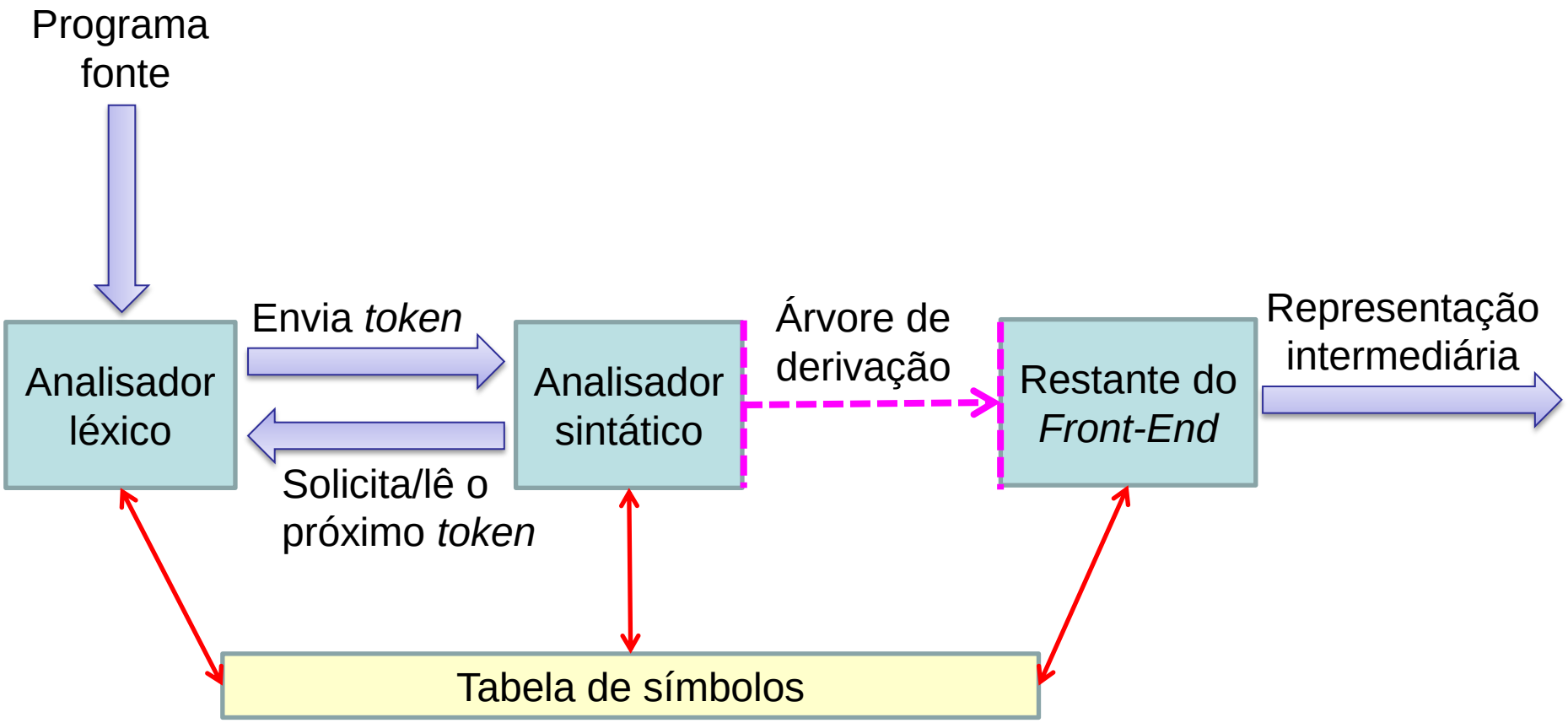
Análise sintática

Conceitualmente, para programas bem formados, o analisador sintático constrói uma árvore de derivação e a passa para o restante do *front-end* do compilador para mais processamento.

Na prática, não é necessário construir uma 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.

Análise sintática



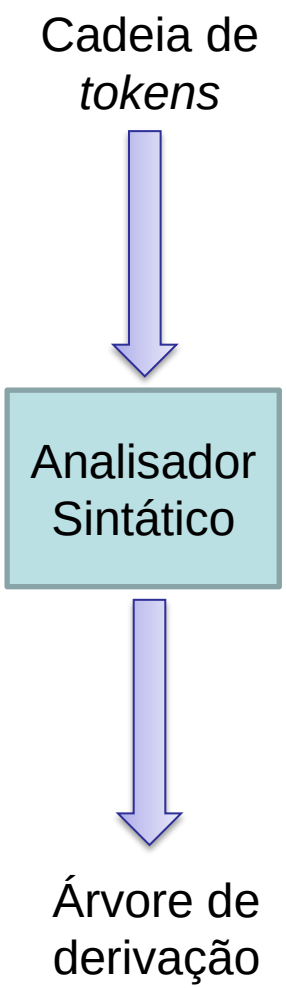
Análise sintática

Em aula, vamos considerar que a saída do analisador sintático é alguma representação da árvore de derivação para a cadeia de *tokens* reconhecidas pelo analisador léxico.

Na prática, existem várias tarefas que poderiam ser conduzidas durante a análise sintática, como:

- Coleta de dados sobre os *tokens* da tab. de símbolos.
- Verificação de tipo e outros tipos de análise semântica.
- Geração do código intermediário.

Todas essas atividades foram reunidas na caixa denominada "Restante do *front-end*".



Análise sintática

Tratamento de erro de sintaxe

Se um compilador tivesse de processar apenas programas corretos, o seu projeto e sua implementação seriam muito simplificados.

Mas espera-se que um compilador auxilie o programador na localização e rastreamento de erros que surgem nos programas, apesar de esforços do programador.

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 do compilador.

Um bom compilador deveria assistir o programa na identificação e localização dos erros.

Análise sintática

Tratamento de erro de sintaxe

Os erros mais comuns podem ocorrer em diferentes níveis:

Léxicos, tais como símbolos desconhecidos, erro de grafia de um identificador, palavra-chave ou operador.

Ex.: o uso de um identificador whule no lugar de while.

Ex.: esquecer de colocar aspas em um texto literal.

Sintáticos, tais como uma expressão aritmética com parênteses não balanceados.

Ex.: ausência de “;” e chaves faltando “{“ ”}”.

Ex.: o comando case sem switch (em Java ou C). É um erro sintático, mas apenas é identificada mais adiante (geração de código).

Análise sintática

Tratamento de erro de sintaxe

Os erros mais comuns podem ocorrer em diferentes níveis:

Semânticos, tais como um operador aplicado a um outro operador incompatível. Divergência de tipos entre operando e operador.

Ex.: um comando *return* em um método Java com tipo de resultado *void*.

Lógicos, tais como uma chamada infinitamente recursiva.

Ex.: incluem desde o raciocínio incorreto do programador ao uso, em um programa C, do operador de atribuição = em vez do operador de comparação ==. O programa com = pode estar bem formado, porém, pode não refletir a intensão do programador.

Análise sintática

Tratamento de erro de sintaxe

O tratador ou recuperador de erros num analisador sintático possui metas desafiadoras em sua implementação:

Informar a presença de erros de forma clara e precisa;

Recuperar-se de erros com rapidez suficiente para detectar erros subsequentes;

Não deve retardar o processamento, ou seja, deve acrescentar um custo mínimo no processamento de programas corretos.

Felizmente, os erros comuns não são complicados, portanto, um mecanismo de tratamento de erros relativamente simples muitas vezes é suficiente.

Análise sintática

Tratamento de erro de sintaxe

Existem muitas estratégias gerais diferentes que um analisador sintático pode empregar para se recuperar de um erro sintático.

Apesar de nenhuma delas ter provado ser universalmente aceitável, uns poucos métodos têm ampla aplicabilidade.

Estratégias de recuperação de erros:

- Modalidade do desespero (recuperação do modo pânico);
- Recuperação em nível de frase;
- Produções de erro;
- Correção global;

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Modalidade do desespero (recuperação do modo pânico):

É o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática.

Ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de *tokens de sincronização*.

Os *tokens de sincronização* são usualmente delimitadores, tais como o ponto-e-vírgula ou o **fim** (**}**, **.**, ...), cujo papel no programa-fonte seja claro.

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Modalidade do desespero (recuperação do modo pânico):

Naturalmente, o projetista do compilador precisa selecionar os *tokens de sincronização* apropriados à linguagem-fonte.

Embora a correção no modo de pânico (desespero) normalmente ignora uma quantidade considerável de símbolos terminais no programa fonte sem se preocupar com a busca de erros adicionais, ele tem a vantagem da simplicidade e, diferencialmente de alguns outros métodos, tem-se a garantia de não entrar em laço infinito (*loop* infinito).

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

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 de entrada restante por alguma cadeia que permita a continuação da análise.

Uma 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.

A escolha da correção local fica a critério do projetista do compilador.

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Recuperação em nível de frase:

Precisamos ter cuidado para que as substituições não provoquem *loops infinitos*, como aconteceria, por exemplo, se sempre inseríssemos algo na frente do símbolo de entrada corrente.

A substituição em nível de frase tem sido usada em diversos compiladores com recuperação de erro, pois pode corrigir qualquer cadeia de entrada.

Sua principal desvantagem é dificuldade de lidar com situações em que o erro real ocorreu antes do ponto de detecção.

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Produções de erro (regras de produções para erro):

Se tivéssemos uma boa ideia dos erros mais comuns que poderiam ser encontrados, poderíamos aumentar a gramática para a linguagem em exame com as produções que gerassem construções ilegais.

Portanto, podemos estender a gramática da linguagem com produções que geram construções erradas, antecipando os erros mais comuns que poderiam ser encontrados.

Se uma produção de erro for usada pelo AS, podemos gerar diagnósticos apropriados p/ indicar a construção ilegal que foi reconhecida na entrada.

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Correção global:

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 sequencia mínima de mudanças a fim de obter uma correção com custo global menor.

Análise sintática

Tratamento de erro de sintaxe

Estratégicas de recuperação de erros:

Correção global:

Ex.: Dado uma cadeia incorreta x na entrada e uma gramática G, esses algoritmos encontram uma árvore de derivação p/ uma cadeia relacionada y, tal que o número de inserções, exclusões e substituição de *tokens* necessários para transformar x em y seja o menor possível.

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

Análise sintática

Os métodos geralmente utilizados em compiladores são baseados em estratégias descendente ou ascendente.

Os métodos de análise descendente (*top-down*) constroem as árvores de derivação de cima (raiz) para baixo (folhas).

Os métodos de análise ascendente (*bottom-up*) constroem as árvores de derivação no sentido inverso, ou seja, começam das folhas e avançam até a raiz (de baixo para cima).

Análise sintática

Análise sintática descendente

A análise parte da raiz da árvore de derivação e segue em direção as folhas, ou seja, a partir do símbolo inicial da gramática vai-se procurando substituir os símbolos não-terminais de forma a obter nas folhas da árvore a cadeia desejada.

Essa análise pode ser vista como um método que produz uma derivação mais a esquerda para uma cadeira de entrada.

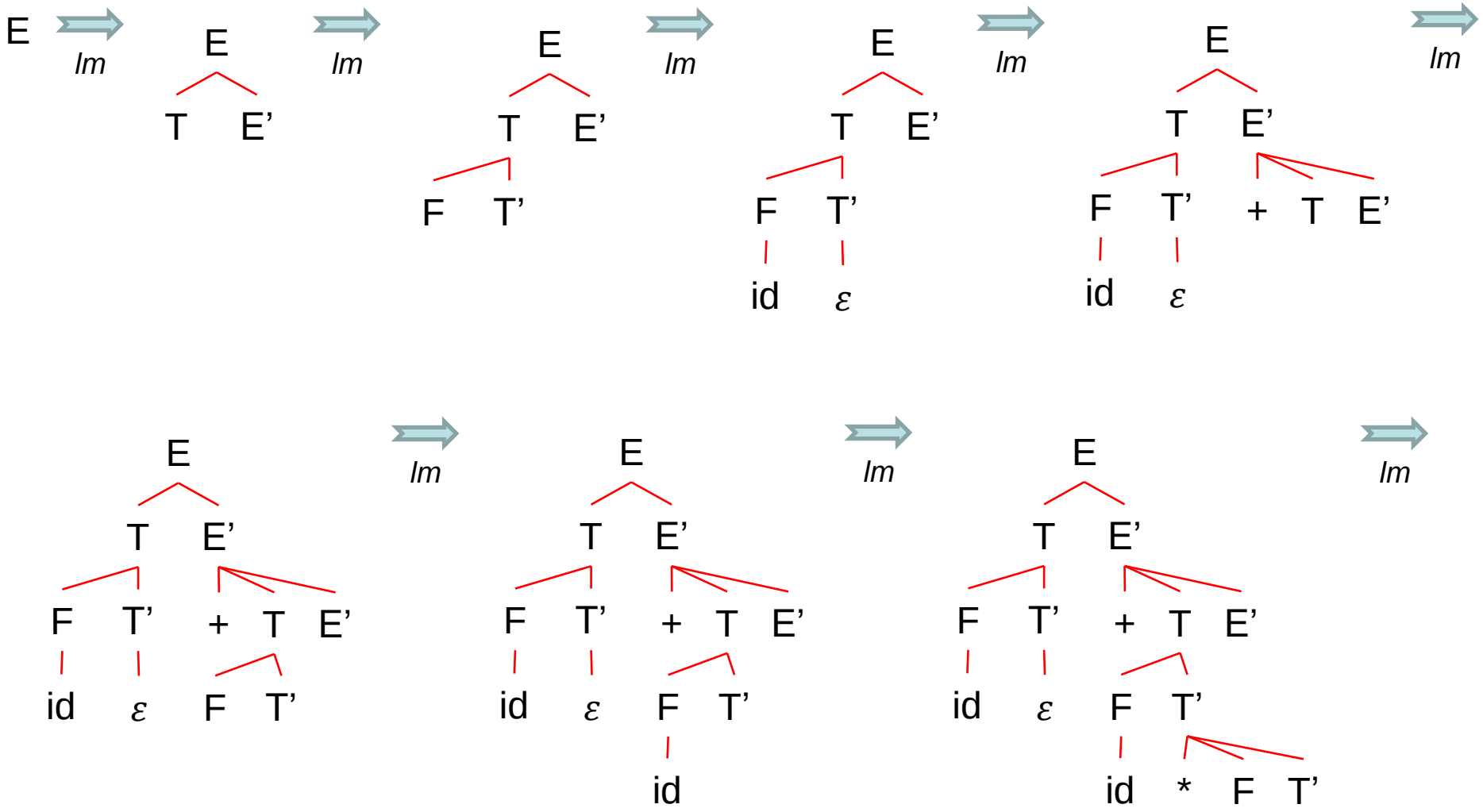
A sequencia de árvores de derivação do próximo *slide* para a entrada id+id*id representa uma análise sintática descendente de acordo com a gramática:

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

Análise sintática

Entrada: id+id*id

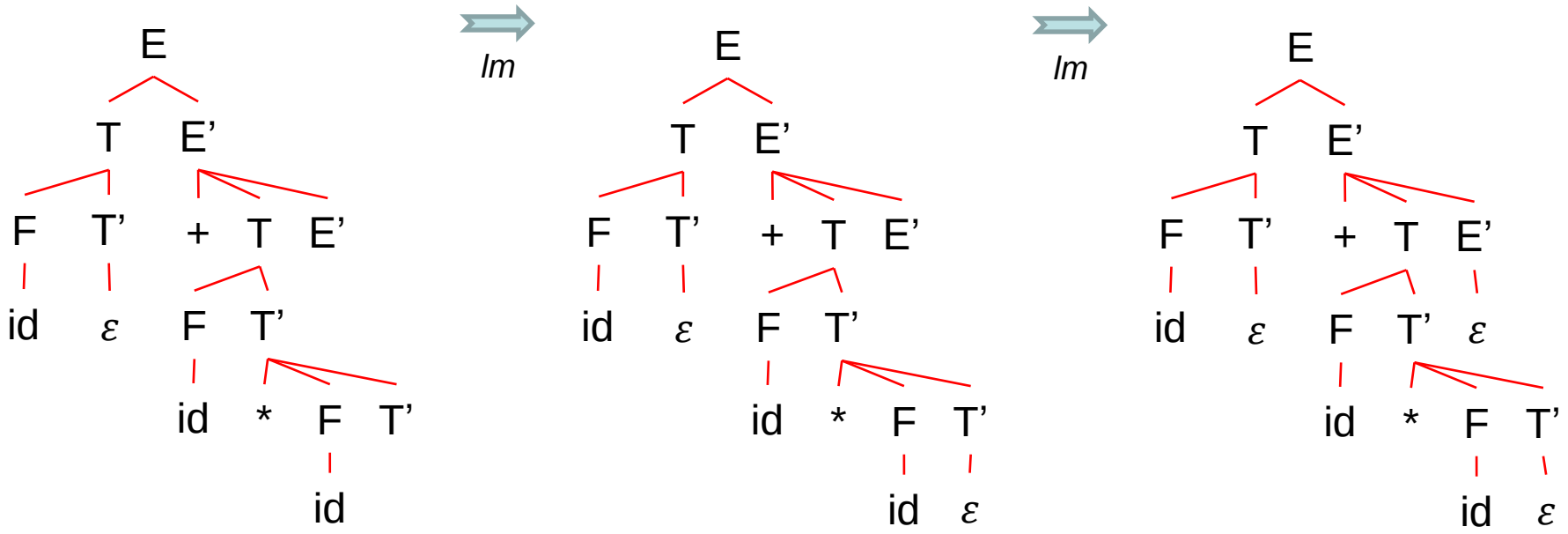
E	→	T E’
E’	→	+ T E’ | ε
T	→	F T’
T’	→	* F T’ | ε
F	→	(E) | id



Análise sintática

Entrada: id+id*id

E	→	T E’
E’	→	+ T E’ | ε
T	→	F T’
T’	→	* F T’ | ε
F	→	(E) | id



A análise sintática descendente estará completa se cada uma das folhas da árvore contiver símbolos terminais da cadeia de entrada.

Esta sequencia de árvores corresponde a uma derivação mais à esquerda da entrada.

Análise sintática

Análise sintática descendente

A cada passo de uma AS descendente, o problema principal é determinar a produção a ser aplicada para um não terminal, ex. 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.

Tipos de analisadores sintáticos descendentes:

Análise sintática descendente recursiva: com e sem retrocesso.
Análise sintática descendente preditiva.

Análise sintática

Análise sintática descendente recursiva

Um método de análise de descida recursiva consiste em um conjunto de procedimentos, um para cada conjunto não-terminal da gramática.

A execução começa com a ativação do procedimento referente ao símbolo inicial da gramática, que para e anuncia sucesso se seu corpo conseguir escandir toda cadeia de entrada.

Análise sintática

Análise sintática descendente recursiva

Pseudocódigo de um procedimento típico para um não-terminal em um analisador descendente:

```
    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 até o próximo símbolo terminal;
7)          else /* ocorreu um erro */
        }
    }
```

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

O método geral de análise sintática pode exigir retrocesso (*backtracking*).

Ou seja, pode voltar atrás no reconhecimento, fazendo repetidas leituras sobre a entrada.

Para permitir o retrocesso, o código anterior precisa ser modificado.

```
    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 até o próximo símbolo terminal;
7)          else /* ocorreu um erro */          }          }
```

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

Primeiro, não podemos escolher uma única produção-A na linha (1).

Devemos tentar cada uma das diversas alternativas da produção-A em alguma ordem.

```
    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 até o próximo símbolo terminal;
7)          else /* ocorreu um erro */          }          }
```

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

Segundo, um erro na linha (7) não é uma falha definitiva.

Apenas sugere que precisamos voltar para 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.

```
     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 até o próximo símbolo terminal;
7)           else /* ocorreu um erro */          }          }
```

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

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 com o objetivo armazenar esse apontador para o uso futuro.

```
    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 até o próximo símbolo terminal;
7)          else /* ocorreu um erro */          }          }
```

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

Foi um dos primeiros a serem utilizados.

Método recursivo com retrocesso. Tentativa e erro, se pegou o caminho errado, volta e pega outro caminho, até conseguir reconhecer a sentença correta.

Tempo de análise muito alto, justamente pelo número de tentativas.

Entretanto, este é um método muito ineficiente, não sendo visto com muita frequência.

Análise sintática

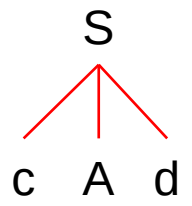
Análise sintática descendente recursiva (**com retrocesso**)

Para ilustrar este método, suponha uma gramática cujas produções são dadas a seguir:

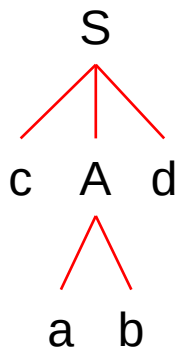
S → cAd
A → ab | a

Ex.: cadeia de entrada *cad*. Aplicando as regras de produção teríamos as seguintes árvores gramaticais para a sentença:

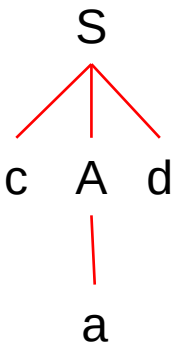
(a)



(b)



(c)



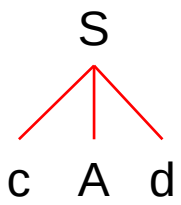
S → cAd
A → ab | a

Análise sintática

Análise sintática descendente recursiva (**com retrocesso**)

Observe-se que a árvore (a) corresponde a derivação S → cAd.

(a)



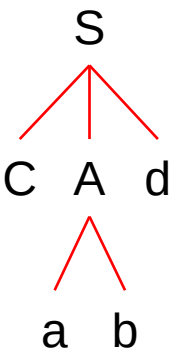
Para derivarmos o não terminal A existem duas alternativas.

Neste ponto o analisador deve escolher uma das alternativas, mas deverá se lembrar das próximas alternativas e qual a situação atual da derivação, criando assim um ponto de escolha.

Análise sintática

S → cAd
A → ab | a

(b)



Análise sintática descendente recursiva (**com retrocesso**)

Escolhendo-se a primeira alternativa, A → ab, obtemos a árvore (b), a qual corresponde à derivação S → cabd.

Neste ponto ocorre uma falha, pois a sentença gerada não corresponde à sentença de entrada.

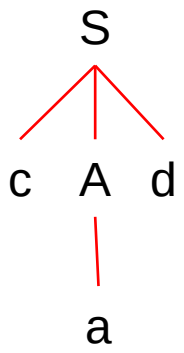
Deste modo, o analisador deve efetuar um retrocesso ao último ponto de escolha, restaurar a situação de derivação antes da escolha da alternativa que falhou, e continuar o processo de derivação a partir deste ponto.

Assim, a situação anterior era a forma sentencial cAd.

Análise sintática

S → cAd
A → ab | a

(c)



Análise sintática descendente recursiva (**com retrocesso**)

Como a primeira alternativa falhou, resta agora seguir a segunda, ou seja, A → a, o que produzirá a derivação S → cad, a qual corresponde a árvore (c).

Como foi possível gerar a sentença inicial a partir desta gramática, então tal sentença pode ser reconhecida por esta gramática.

O problema por trás deste tipo de implementação é a dificuldade de se restaurar a situação no ponto de escolha e o atraso que isto provoca.

Análise sintática

Análise sintática descendente recursiva **(sem retrocesso)**

O método de análise descendente recursiva necessita que a gramática não contenha recursões à esquerda e que esteja fatorada à esquerda.

O método se baseia na escrita de um procedimento recursivo para cada não-terminal da gramática.

Durante a derivação, quando um símbolo terminal for derivado, um novo item léxico deverá ser lido.

E quando um não terminal é encontrado, o procedimento que o define é chamado. As chamadas a estes procedimentos são recursivas e, portanto, a linguagem a ser utilizada na implementação deverá suportar esta característica de modo eficiente.

Análise sintática

Análise sintática descendente recursiva **(sem retrocesso)**

Ex.: analisador sintático recursivo para um reconhecedor de expressões aritméticas envolvendo identificadores, valores inteiros e reais.

As regras gramaticais:

```
<exp>     :   <termo> + <exp> | <termo> - <exp> | <termo>
<termo>   :   <fator> * <termo> | <fator> / <termo> | <fator>
<fator>   :   id | IntNum | RealNum | ( <exp> )
```

onde,

id	→	letra^{+} (letra | digito)*
IntNum	→	digito^{+}
RealNum	→	digito* . digito^{+}
letra	→	[A-Za-z]
digito	→	[0-9]

Análise sintática

```
<exp>    :   <termo> + <exp> | <termo> - <exp> | <termo>
<termo>  :   <fator> * <termo> | <fator> / <termo> | <fator>
<fator>  :   id | IntNum | RealNum | ( <exp> )
```

id	→	letra^{+} (letra | digito)*
IntNum	→	digito^{+}
RealNum	→	digito* . digito^{+}
letra	→	[A-Za-z]
digito	→	[0-9]

Análise sintática descendente recursiva
(sem retrocesso)

```
Rotina Exp
Inicio
        Chamar Termo
        Caso Lexema é
            '+', '-':  Chamar Lexico
                       Chamar Exp
Fim do Caso
Fim da Rotina Exp
```

```
Rotina Termo
Inicio
        Chamar Fator
        Caso Lexema é
            '*', '/': chamar Lexico
                      chamar Termo
Fim do Caso
Fim da Rotina Termo
```

Análise sintática

```
<exp>    :   <termo> + <exp> | <termo> - <exp> | <termo>
<termo>  :   <fator> * <termo> | <fator> / <termo> | <fator>
<fator>  :   id | IntNum | RealNum | ( <exp> )
```

id	→	letra^{+} (letra | digito)*
IntNum	→	digito^{+}
RealNum	→	digito* . digito^{+}
letra	→	[A-Za-z]
digito	→	[0-9]

Análise sintática descendente recursiva **(sem retrocesso)**

```
Rotina Fator
Inicio
        Se Lexema é '(' então
              chamar Lexico
              chamar Exp
              Se Lexema é ')' então
                    Chamar Lexico
              Senão
                    Erro(') esperado')
              Fim se
        Senão
              ...
        Fim se
Fim da Rotina Fator
```

```
Se Token ∈ {Id, IntNum, RealNum} então
        Chamar Lexico
Senão
        Erro('Operando esperado')
Fim se
```

Análise sintática

```
<exp>    :   <termo> + <exp> | <termo> - <exp> | <termo>
<termo>  :   <fator> * <termo> | <fator> / <termo> | <fator>
<fator>  :   id | IntNum | RealNum | ( <exp> )
```

id	→	letra^{+} (letra | digito)*
IntNum	→	digito^{+}
RealNum	→	digito* . digito^{+}
letra	→	[A-Za-z]
digito	→	[0-9]

Análise sintática descendente recursiva
(sem retrocesso)

```
Programa AnalisadorSintaticoRecursivo
Inicio
        Chamar Lexico
        Chamar Exp
        Se Token <> Eof então
             Erro('Fim de arquivo não esperado')
        Fim se
Fim do Programa AnalisadorSintaticoRecursivo
```

Análise sintática

Análise sintática descendente recursiva

Questão de prova

O que pode ocorrer com uma gramática com recursão a 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.

Ex.: S → ScAd.

Análise sintática

E	→	T E’
E’	→	+ T E’ | ε
T	→	F T’
T’	→	* F T’ | ε
F	→	(E) | id

Análise sintática descendente preditiva

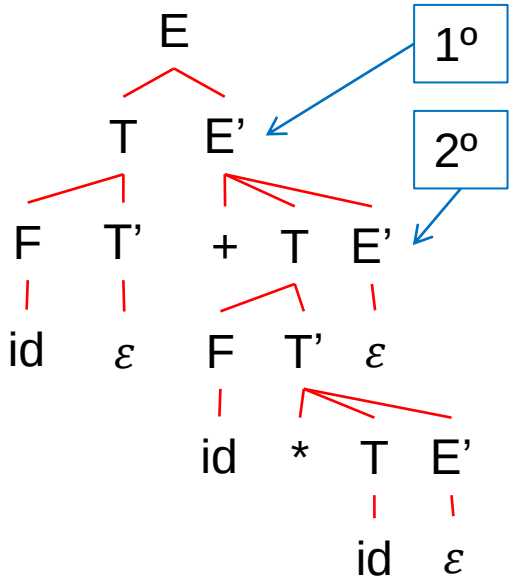
O método de reconhecimento sintático preditivo escolhe a produção-A correta examinando adiante na entrada um número fixo de símbolos.

Na prática, tipicamente se examina só um, ou seja, o próximo símbolo da entrada.

Exemplo:

No primeiro nó E’, a produção E’ → + T E’ é a escolhida.
No segundo nó E’, a produção E’ → ε é escolhida.

Então, uma analisador pode escolher entre as produções-E’ examinando o próximo símbolo da entrada.



Análise sintática

Análise sintática descendente preditiva

A classe de gramática para as quais podemos construir analisadores preditivos examinando k símbolos adiante na entrada às vezes é chamada de LL(k).

Assuntos da próxima aula!!

Com exercícios práticos.

A partir dos conjuntos FIRST e FOLLOW p/ uma gramática, construiremos "tabelas de reconhecimento preditivo", as quais tornam explícita a escolha da produção durante a análise descendente.

Análise sintática

Exercício:

Como ficaria o pseudocódigo abaixo para um analisador descendente com retrocesso? Altere o código para demonstrar sua sugestão.

```
    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 até o próximo símbolo terminal;
7)          else /* ocorreu um erro */
        }
    }
```

Próxima aula

Próxima aula será no laboratório 15.

Apenas a próxima aula, as demais serão em sala de aula.

Substituição com Cristiano Both (prova).

COMPILADORES

Obrigado!!

Prof. Geovane Griesang
geovanegriesang@unisc.br