INE5318
Construção de Compiladores

Ricardo Azambuja Silveira
INE-CTC-UFSC
E-Mail: silveira@inf.ufsc.br
URL: www.inf.ufsc.br/~silveira

Conceitos de Linguagens de Programação

Prof. Ricardo A. Silveira

Conceitos

- ***Sintaxe*** **- a forma ou estrutura das expressões, instruções e unidades de programas.**
- ***Semântica*** **– o significado das expressões, instruções e unidades de programas.**
- **Quem deve usar uma definição de linguagem?**
  - **Outros projetistas**
  - **Implementadores**
  - **Programadores (os usuários da linguagem)**

- **Uma *sentença* é uma cadeia de caracteres sobre algum alfabeto**
- **Uma *linguagem* é um conjunto de sentenças**
- **Um *lexema* é a unidade de mais baixo nível sintáticos de uma linguagem. Incluem:**
  - **Identificadores**
  - **Literais**
  - **Operadores**
  - **Palavras reservadas**
- **Um *token* (símbolo)n é uma categoria de lexemas (e.g., identifier)**

Abordagens formais para descrever LPs

- **Sintaxe:**
  - **Reconhecedores – (máquinas de estados) usada em compiladores**
  - **Geradores – formalismo usado para gerar sentenças na linguagem**
    - **Gramáticas**
    - **Expressões**

Gramática livre de contexto

- Desenvolvida por Noam Chomsky em meados dos anos 50 como um gerador de linguagens com o propósito de descrever a sintaxe das linguagens naturais
- Define uma classe de linguagens denominada *linguagens livres de contexto*
- Backus Naur Form - BNF
  - Metalinguagem inventada em 1959 por John Backus para descrever a linguagem Algol 58 e aperfeiçoada por Peter Naur em 1960
  - Uma *metalinguagem* é uma linguagem usada para descrever outra linguagem.
  - A BNF é equivalente a gramática livre de contexto
  - Em BNF, *abstrações* são usadas para representar classes de estruturas sintáticas, na forma
    <abstração> -> descrição da abstração
  - Que funcionam como variáveis sintáticas (também chamadas *símbolos não-terminais*) que derivam dos lexemas (também chamadas *símbolos terminais*)
  - Exemplos:
    - <atribuição> -> <variável> = <expressão>
    - Isto é uma regra, que descreve a estrutura de um comando de atribuição

BNF

- Uma regra tem um lado esquerdo (left-hand side - LHS) e um lado direito (right-hand side - RHS), e consiste em símbolos *terminais* e *não-terminais*
- Uma *gramática* é um conjunto finito e não vazio de regras
- Uma abstração (ou símbolo não-terminal) pode ter mais que um RHS

  <stmt> -> <single_stmt>
  | begin <stmt_list> end
- Uma lista sintatica é descrita em BNF usando recursão

  <ident_list> -> ident
  | ident, <ident_list>
- uma *derivação* é a aplicação repetida de regras, a partir do símbolo de início e terminando com uma sentença formada apenas com símbolos terminais

Um exemplo de gramática

<program> -> begin <stmts> end

<stmts> -> <stmt>

| <stmt> ; <stmts>

<stmt> -> <var> = <expr>

<var> -> a | b | c | d

<expr> -> <term> + <term>

| <term> - <term>

<term> -> <var>

| const

Um exemplo de derivação

<program> => <stmts> => <stmt>

=> <var> = <expr> => a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

=> a = b + const

- Cada cadeia de símbolos na derivação é uma *forma sentencial*
- Uma *sentença* é a forma sentencial que tem apenas símbolos terminais
- uma *derivação a esquerda* é aquela em que o símbolo não-terminal mais a esquerda em cada forma sentencial é escolhida para expansão
- Uma derivação pode ser mais a esquerda, mais a direita ou mixta

Árvores de análise

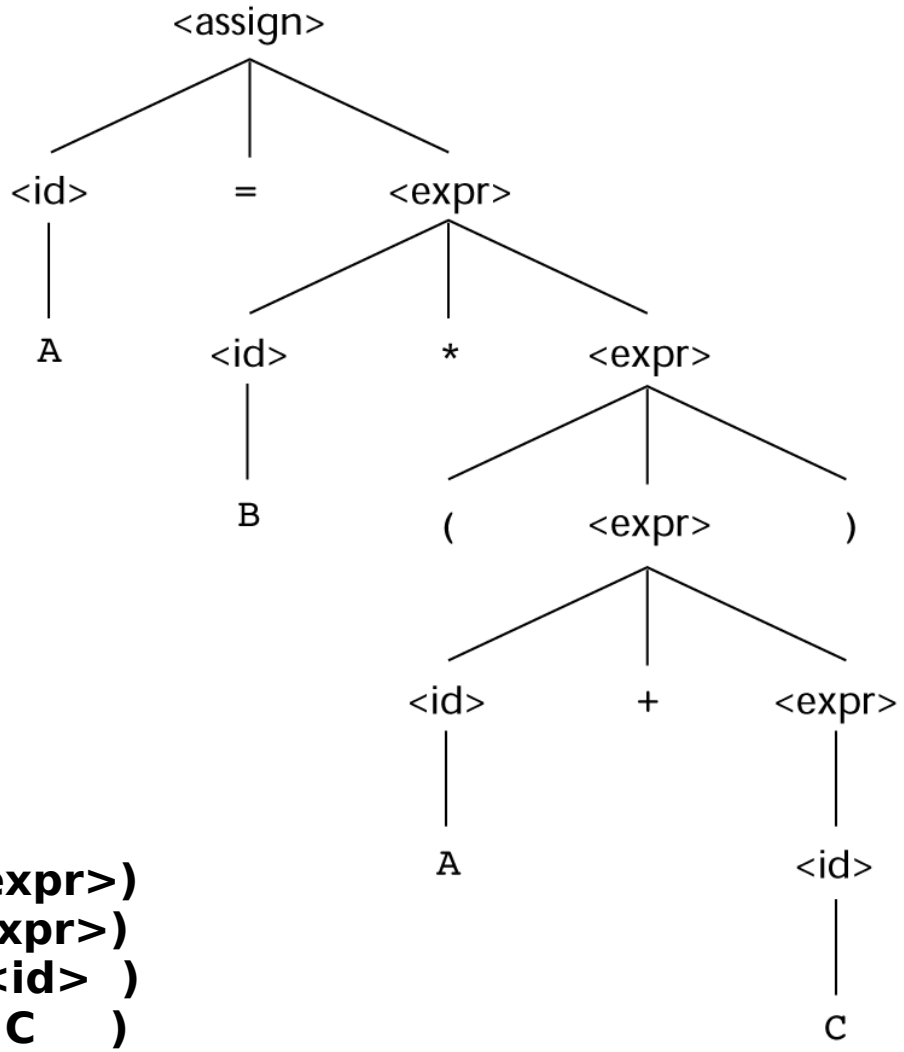
- Uma árvore de análise (parse tree) é uma representação hierárquica de uma derivação
- Uma gramática é *ambígua* se ela gerar uma forma sentencial que tem duas ou mais diferentes árvores de análise
- Exemplo:

```
<expr> -> <expr> <op> <expr>
        |  const
<op> -> /  |  -
```

Árvores de análise

Uma árvore de análise para o comando A = B * (A + C)

<atribuição> -> <id> = <expr>
<id> ->A | B | C
<expr> -> <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

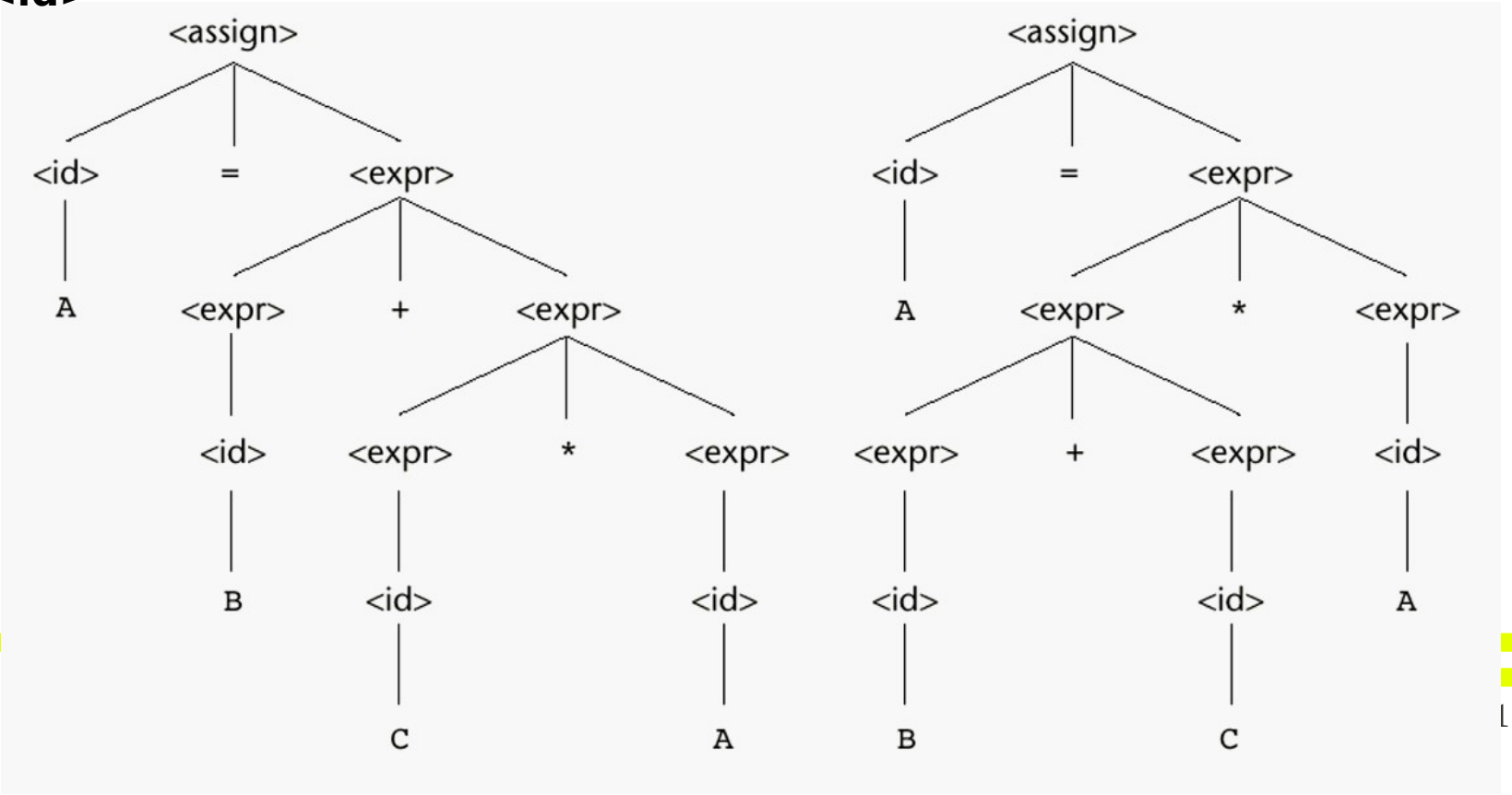


A = B *(A+C)
<atribuição> -> <id> = <expr>
A = <expr>
A = <id> * <expr>
A = B * <expr>
A = B * (<expr>)
A = B * (<id> + <expr>)
A = B * (A + <expr>)
A = B * (A + <id>)
A = B * (A + C)

Ambiguidade

Dado a gramática abaixo, a expressão A = B + C * A tem duas árvores distintas

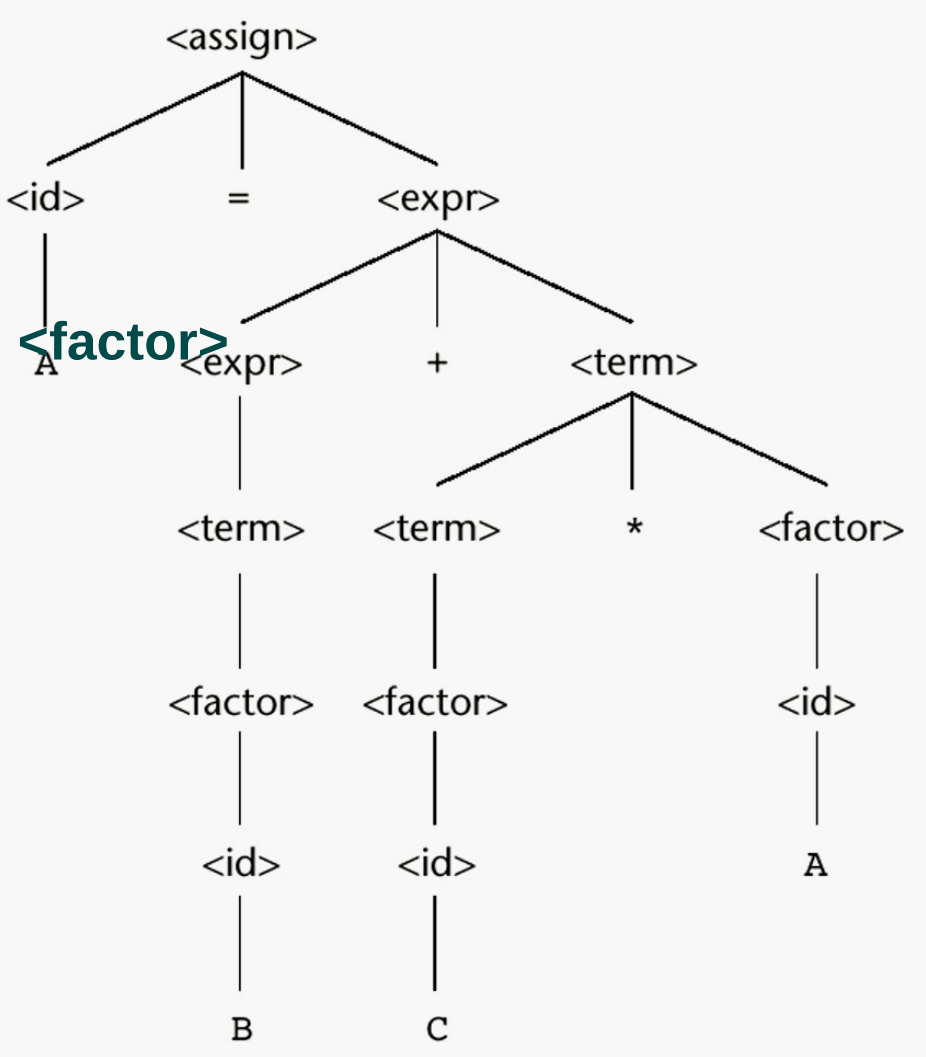
<atribuição> -> <id> = <expr>
<id> ->A | B | C
<expr> -> <expr> + <expr>
| <expr> * <expr>
| <(<expr>)
| <id>



Precedência

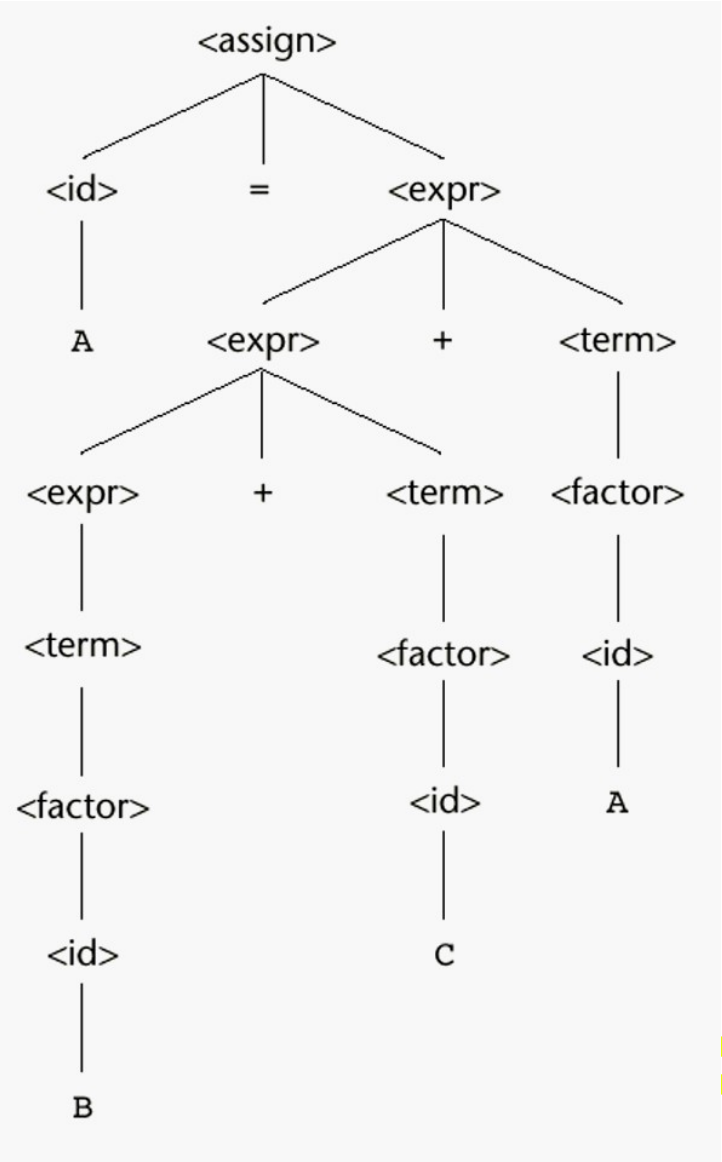
- Pode-se utilizar a gramática de forma a indicar a precedência dos operadores, sem ambigüidades
- Uma única árvore para a expressão A = B + C * A usando uma gramática não ambígua

<assingn> -> <id> = <expr>
<id> ->A | B | C
<expr> -> <expr> + <term>
|<term>
<term> -> <term> * <factor>
| <factor>
<factor> -> (<expr>)
| <id>



Associatividade

- Uma árvore de análise para a expressão A = B + C + A ilustrando a associatividade da adição



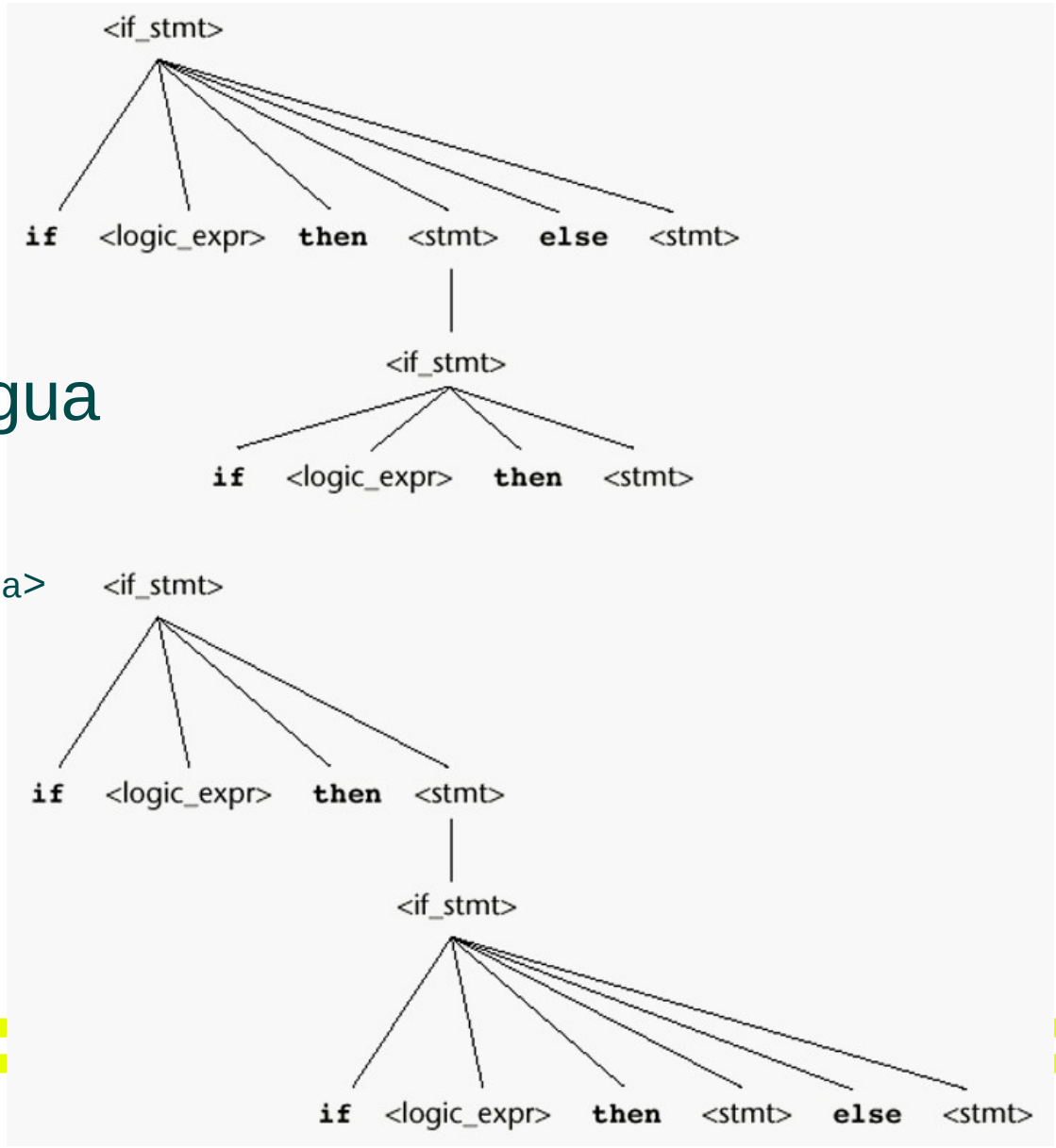
If then else

- Uma gramática ambígua para if then else

<if_stmt> -> if <expr_logica> then <stmt>
| if <expr_logica> then <stmt> else <stmt>

- Uma gramática não-ambígua para if then else

<stmt> -> <casada> |<livre>
<casada> -> if <expr_logica> then <casada> else <casada>
| qualquer instrução não-if
<livre> -> if <expr_logica> then <stmt>
| if <expr_logica> then <casada> else <livre>



BNF extendida (EBNF)

Serve apenas para abreviar a notação da BNF

- **Partes opcionais são colocadas entre colchetes ([])**

```
<proc_call> -> ident [ ( <expr_list>)]
```

- **Partes alternativas das RHSs entre parênteses e separadas por barras verticais**

```
<term> -> <term> (+ | -) const
```

- **Repetições (0 or mais) entre chaves ({})**

```
<ident> -> letter {letter | digit}
```

BNF:

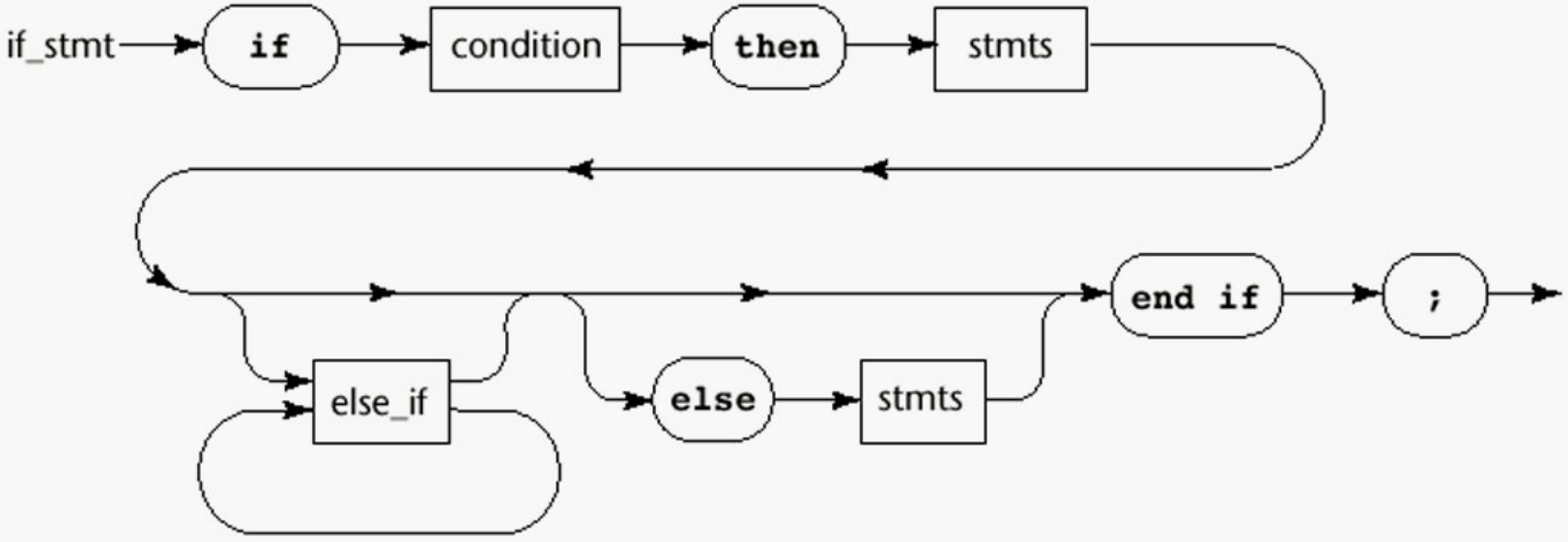
```
<expr> -> <expr> + <term>
        | <expr> - <term>
        | <term>
<term> -> <term> * <factor>
        | <term> / <factor>
        | <factor>
```

EBNF:

```
<expr> -> <termo> {(+ | -) <term>}
<term> -> <fator> {(* | /) <factor>}
```

Grafos de sintaxe

Os grafos de sintaxe e a descrição EBNF do comando if



<if_stmt> ⟶ **if** <condition> **then** <stmts> {<else_if>}

[**else** <stmts>] **end if**

<else_if> ⟶ **elsif** <condition> **then** <stmts>

Gramática de atributos

- Desenvolvida por Knuth, 1968
- Gramáticas livre de contexto não tem capacidade para descrever completamente a sintática de linguagens de programação
- Mecanismos adicionados a GLC pata tratar algumas informações semânticas relacionadas as formas legais do programa na construção das árvores de análise
- Valor primário da gramática de atributos:
  - Especificação da semântica estática
  - Projeto de compiladores (verificação da semântica estática)
- Definição:
- Uma *gramática de atributo* é a gramática livre de contexto com as seguintes adições:
  - Para cada símbolo gramatical x há um conjunto A(x) de atributos
  - Cada regra tem um conjunto de funções que definem certos atributos dos símbolos não-terminais em uma regra
  - Cada regre tem um conjunto (possivelmente vazio) de predicados para checar a consistência dos atributos

Gramática de atributos

- Seja a regra X0 -> X1 ... Xn
- Funções na forma S(X0) = f(A(X1), ... A(Xn)) definem, *atributos sintetizados*
- Funções na forma I(Xj) = f(A(X0), ... , A(Xn)), para i <= j <= n, definem *atributos herdados*
- Inicialmente, existem *atributos intrínsecos* nas folhas
- *Exemplo:* expressões na forma id + id
- - id's podem ser tipo int ou real
- - tipos dos dois id's devem ser os mesmos
- - tipos de expressão devem ser o mesmo que o tipo esperado
- *BNF:*
- <expr> -> <var> + <var>
- <var> -> id
- *Attributos:*
- *tipo_efetivo* – sintetizado para <var> e <expr>
- *tipo_esperado* – herdado para <expr>

Gramática de atributos

Regra sintática: <expr> -> <var>[1] + <var>[2]

Regra semantica: <expr>.actual_type ← <var>[1].actual_type

Predicado:

<var>[1].actual_type = <var>[2].actual_type

<expr>.expected_type = <expr>.actual_type

Regra sintática: <var> -> id

Regra semantica: <var>.actual_type ← lookup (id, <var>)

Gramática de atributos

- *Como os valores dos atributos são computados?*
- 1. Se todos os atributos foram herdados, a árvore é decorada em ordem top-down.
- 2. Se todos os atributos foram sintetizados, a árvore é decorada em ordem bottom-up.
- 3. Em muitos casos, os dois tipos de atributos são usados e uma combinação de top-down e bottom-up é usada.

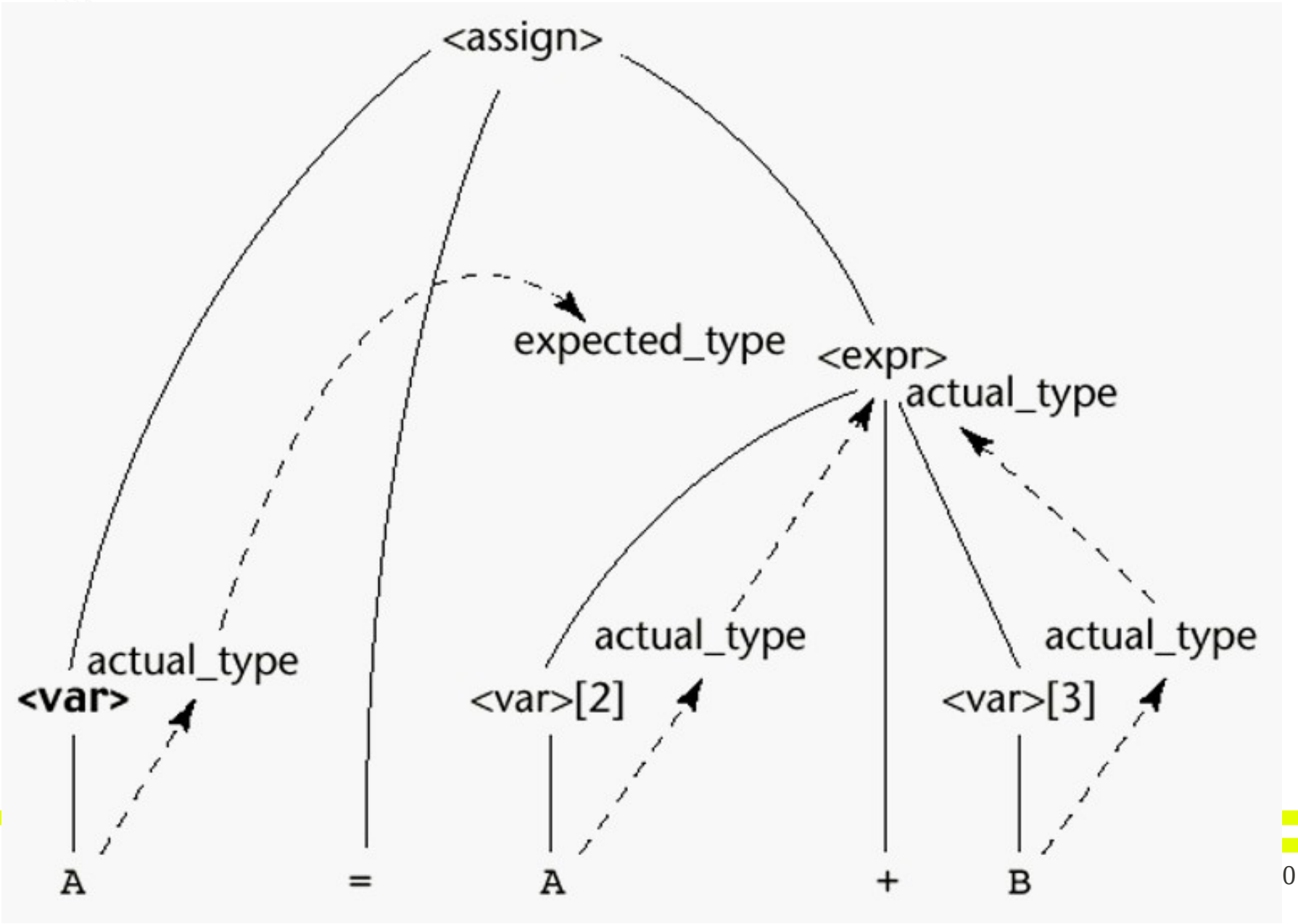
Gramática de atributos

- 1. <expr>.expected_type ← inherited from parent

- 2. <var>[1].actual_type ← lookup (A, <var>[1])
- <var>[2].actual_type ← lookup (B, <var>[2])
- <var>[1].actual_type =? <var>[2].actual_type

- 3. <expr>.actual_type ← <var>[1].actual_type
- <expr>.actual_type =? <expr>.expected_type

Gramática de atributos



Semântica dinâmica

- Denota o significado das expressões, das instruções e das unidades de programas
- Nenhuma notação ou formalismo simples para descrição semântica é aceito largamente para descrever a semântica dinâmica das LPs
- Utilidade:
  - Conhecimento da linguagem pelos programadores
  - Construção de compiladores
  - Geração automática de compiladores
  - Prova de exatidão de programas
- Principais métodos:
  - Semântica operacional
  - Semântica axiomática
  - Semântica denotacional

Semântica operacional

- Descreve o significado de um programa através da execução de seus comandos em uma máquina real ou virtual. As mudanças no estado da máquina (memória, registradores, etc.) definem o significado de cada comando
- Seu uso para descrever a semântica de linguagens de alto nível exige a construção de uma máquina virtual
  - Um interpretador implementado em hardware seria muito caro
  - Um interpretador implementado em software apresenta problemas:
    - Os detalhes característicos de um computador em particular tornam difícil a compreensão das ações
    - Uma semântica definida desta forma é altamente dependente da máquina
- Alternativa viável: uma simulação completa de um computador
- O processo:
  - Costruir um tradutor para converter as instruções no código fonte pra um código de máquina do computador simulado
  - Construir um simulador do computador idealizado (máquina virtual)
- Avaliação da semântica operacional:
  - Boa se usada informalmente
  - Extremamente complexa se for usada formalmente

Semântica operacional

Exemplo

Instrução C

```
for(expr1; expr2; expr3) {
        …
        }
```

Semântica operacional

```
        expr1;
loop:   if expr2 = 0 goto out
        ….
        expr3;
        goto loop
out:    …
```

Semântica axiomática

Baseada em lógica formal (cálculo de predicados de primeira ordem)

Propósito original: verificação formal de programas

Abordagem: Define axiomas ou regras de inferência para cada tipo de comando da linguagem para permitir a transformação de expressão em outras expressões

As expressões são chamadas *asserções*

Uma asserção anterior a um comando (uma *pré-condição*) define o relacionamento e as restrições entre as variáveis que são verdadeiras naquele ponto de execução

Uma asserção posterior a um comando é uma *pós-condição*

Uma *precondição mais fraca* é a menos restritiva precondição que garante a pós-condição

- - Pre-post form: {P} statement {Q}

um exemplo: a := b + 1 {a > 1}

Uma possível pré-condição: {b > 10}

Pré-condição mais fraca: {b > 0}

Semântica denotacional

- Baseada na teoria da função recursiva
- O método mais abstrato de descrição semântica
- Originalmente desenvolvida por Scott e Strachey (1970)
- O processo de construção de uma especificação denotacional para uma linguagem
  - Definir um objeto matemático para cada entidade da linguagem
  - Definir uma função que mapeie instâncias das entidades da linguagem em instâncias dos objetos matemáticos correspondentes