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

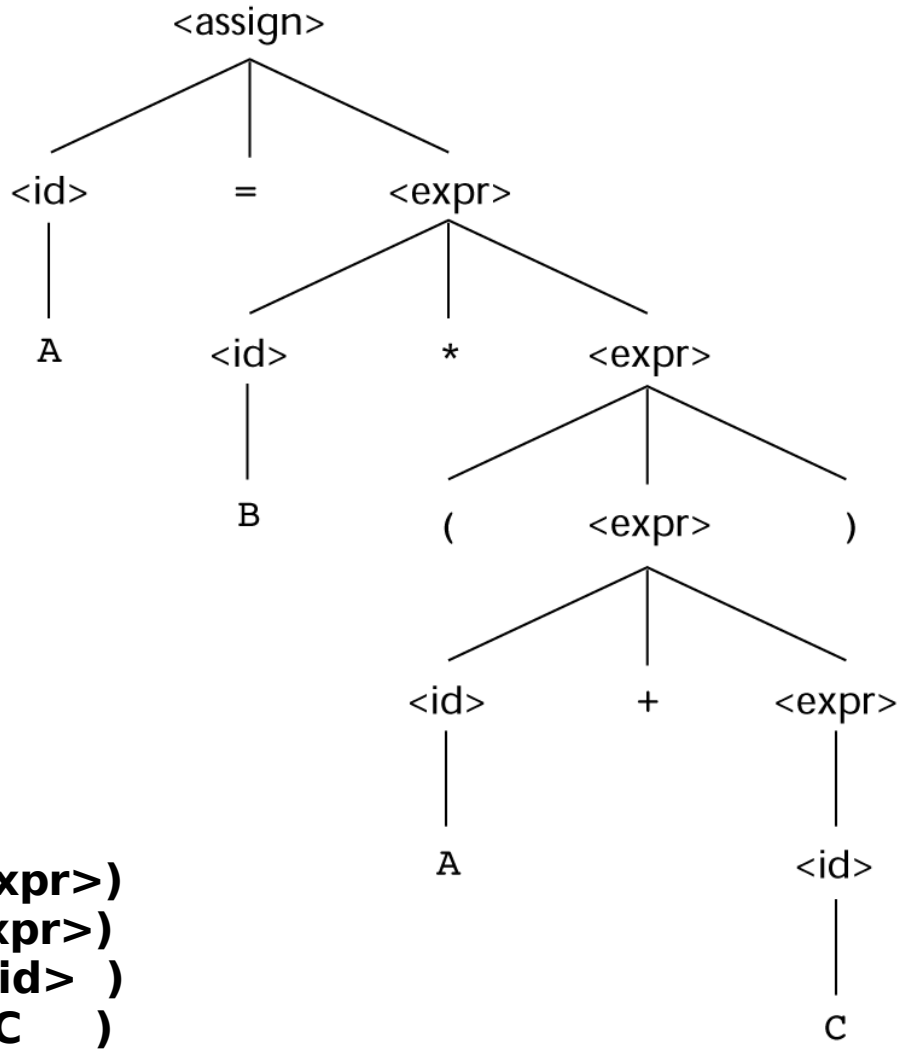
- Uma árvore de análise (parse tree) é uma representação hierárquica de uma derivação
- Uma gramática é *ambígua* se ela gerar uma forma sentencial que tem duas ou mais diferentes árvores de análise
- Exemplo:

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

Árvores de análise

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

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

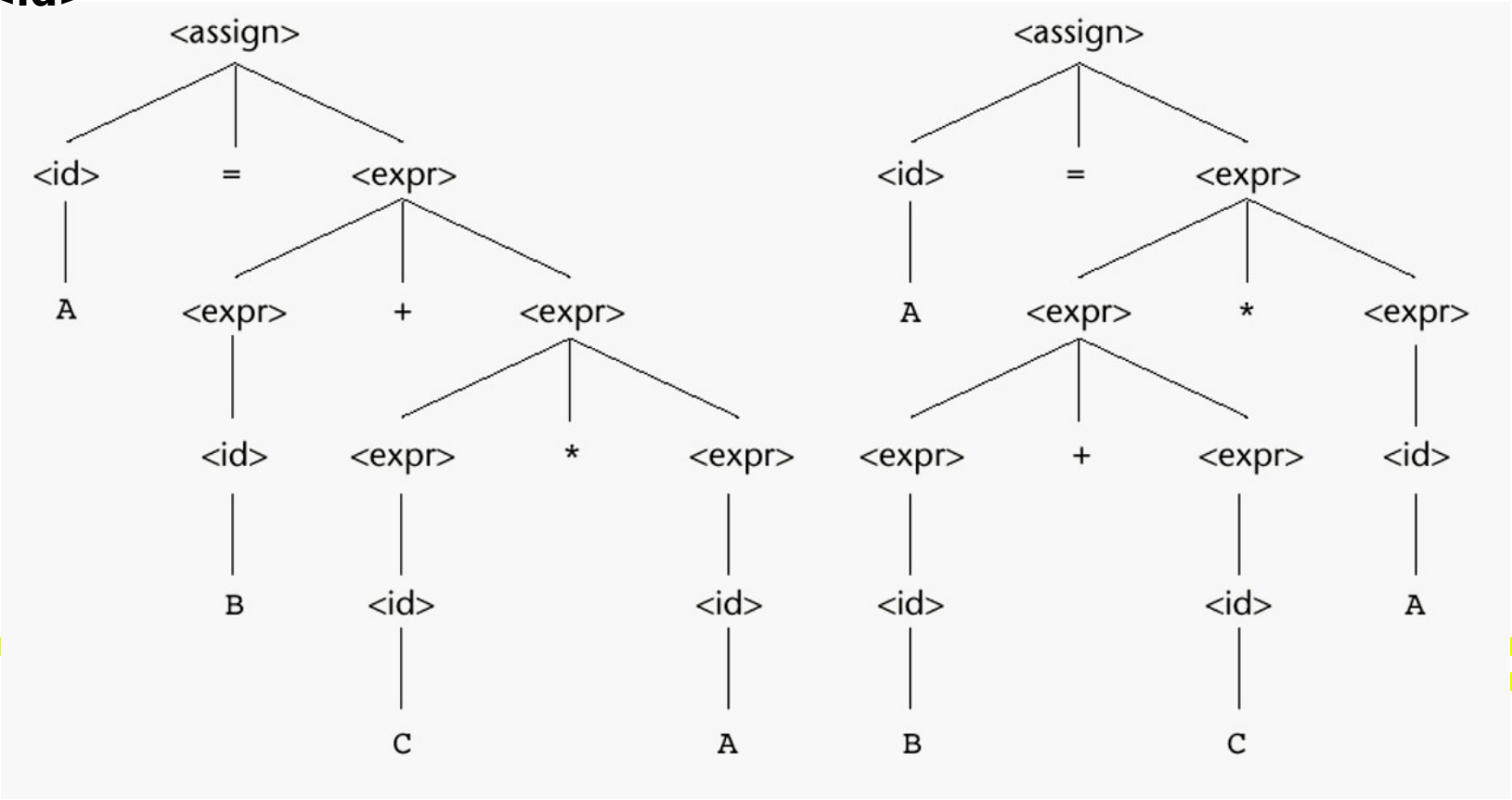


A = B *(A+C)
<atribuição> -> <id> = <expr>
A = <expr>
A = <id> * <expr>
A = B * <expr>
A = B * (<expr>)
A = B * (<id> + <expr>)
A = B * (A + <expr>)
A = B * (A + <id>)
A = B * (A + C)

Ambiguidade

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

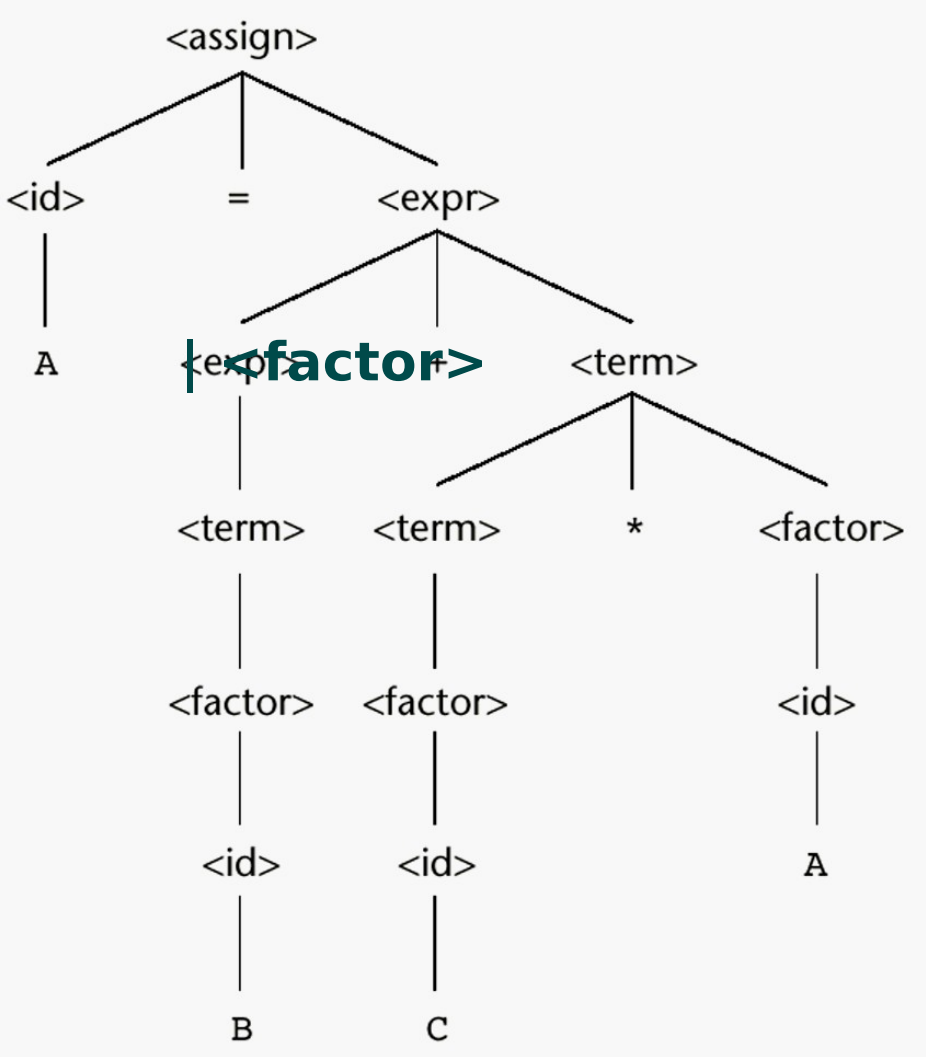
<atribuição> -> <id> = <expr>
<id> ->A | B | C
<expr> -> <expr> + <expr>
| <expr> * <expr>
| <(<expr>)
| <id>



Precedência

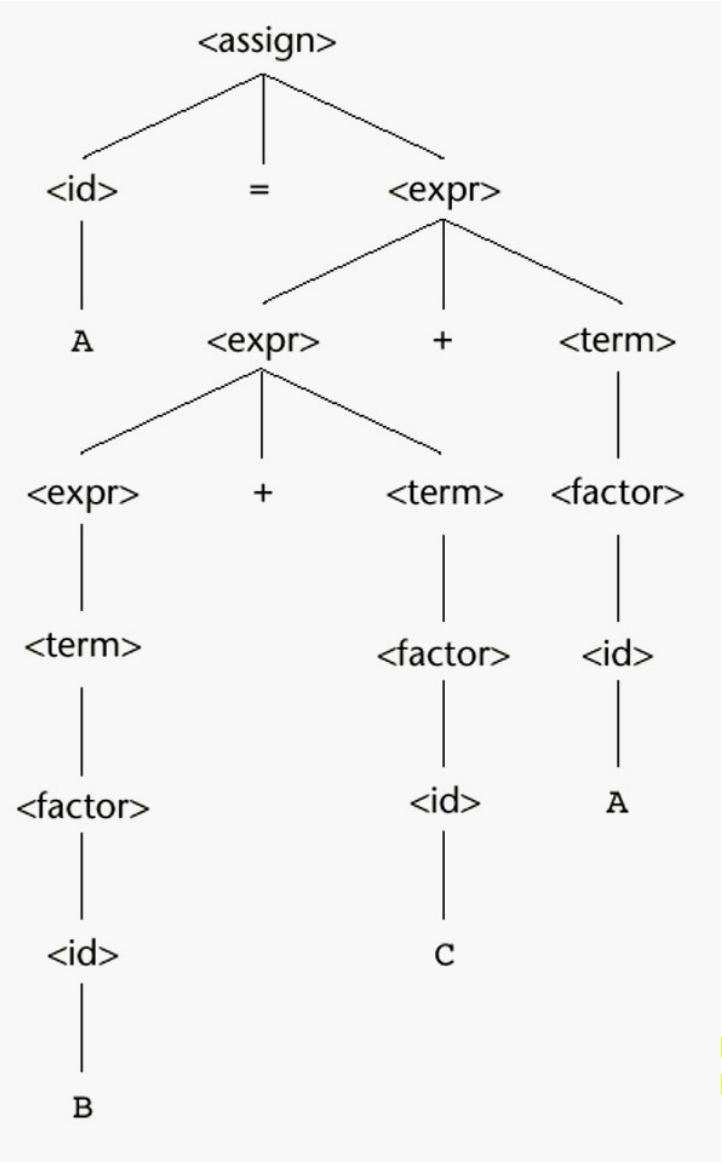
- Pode-se utilizar a gramática de forma a indicar a precedência dos operadores, sem ambigüidades
- Uma única árvore para a expressão A = B + C * A usando uma gramática não ambígua

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



Associatividade

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



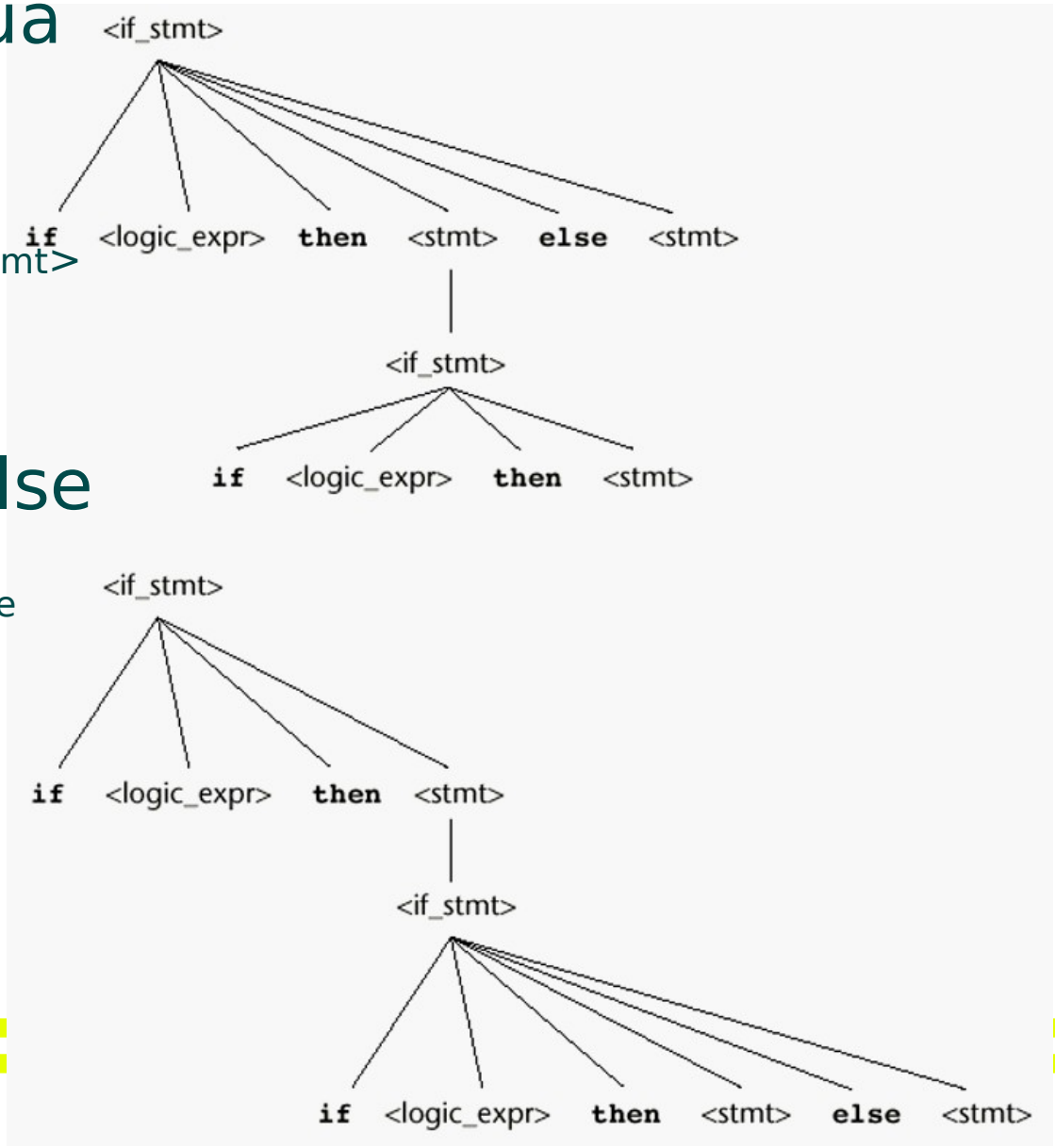
If then else

- 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:

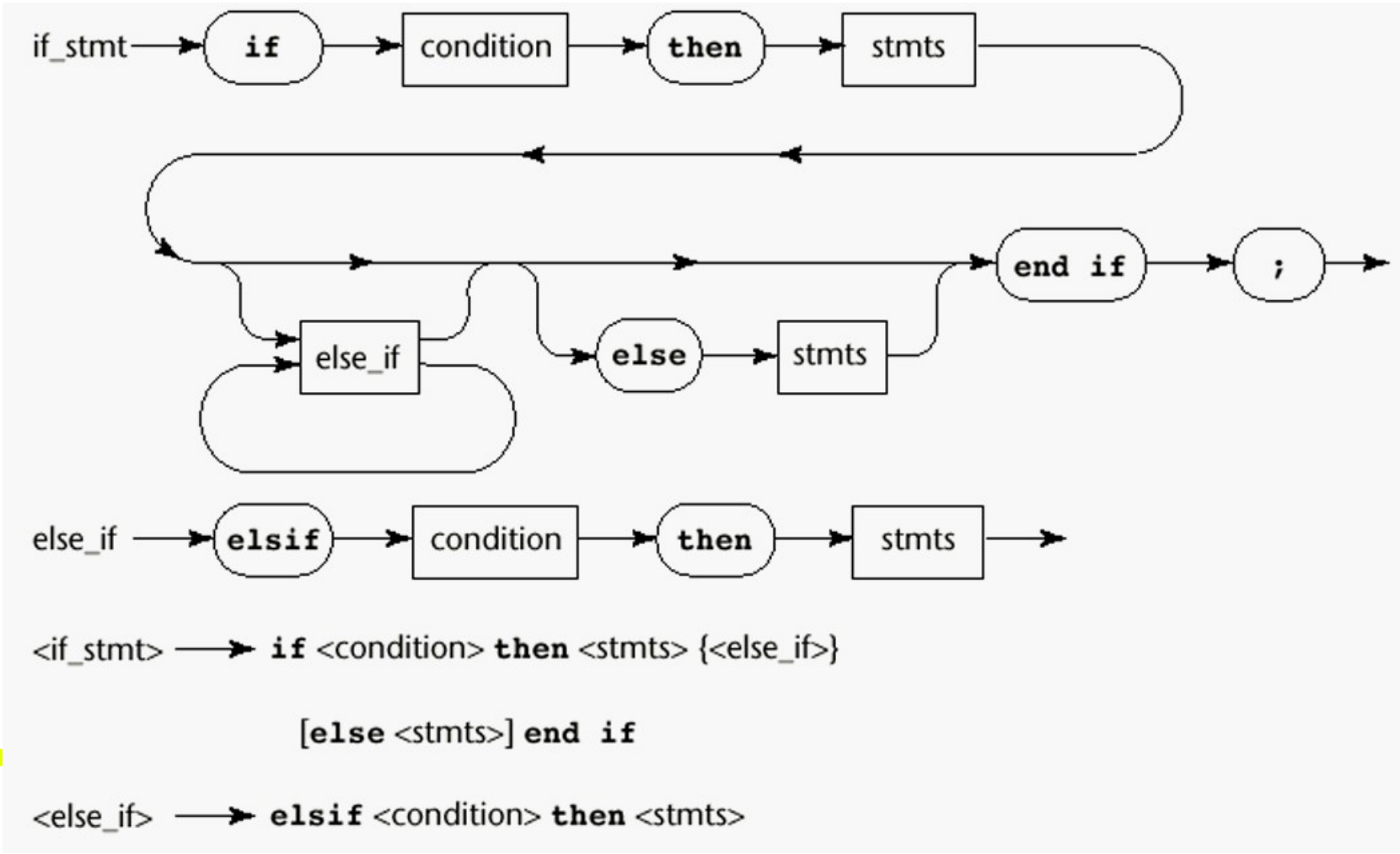
```
<expr> -> <expr> + <term>
        | <expr> - <term>
        | <term>
<term> -> <term> * <factor>
        | <term> / <factor>
        | <factor>
```

EBNF:

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

Grafos de sintaxe

Os grafos de sintaxe e a descrição EBNF do comando if



Exercícios

- Usando a gramática mostrada abaixo, mostre uma árvore de análise e uma derivação à esquerda das instruções:
- A = A * (B + (C * A))
- B = C * (A * C + B)

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

Exercícios

- Usando a gramática mostrada abaixo, mostre uma árvore de análise e uma derivação à esquerda das instruções:
- A = (A + B) * C
- A = B + C + A
- A = A * (B + C)

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

Exercícios

- Considerando a gramática mostrada abaixo, quais das seguintes sentenças pertencem a linguagem gerada por ela:
- baab
- bbbab
- bbaaaaa
- bbaab

<S> -> <A>a<B>b
<A> -> <A>b | b
<B> -> a<B> | a

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.

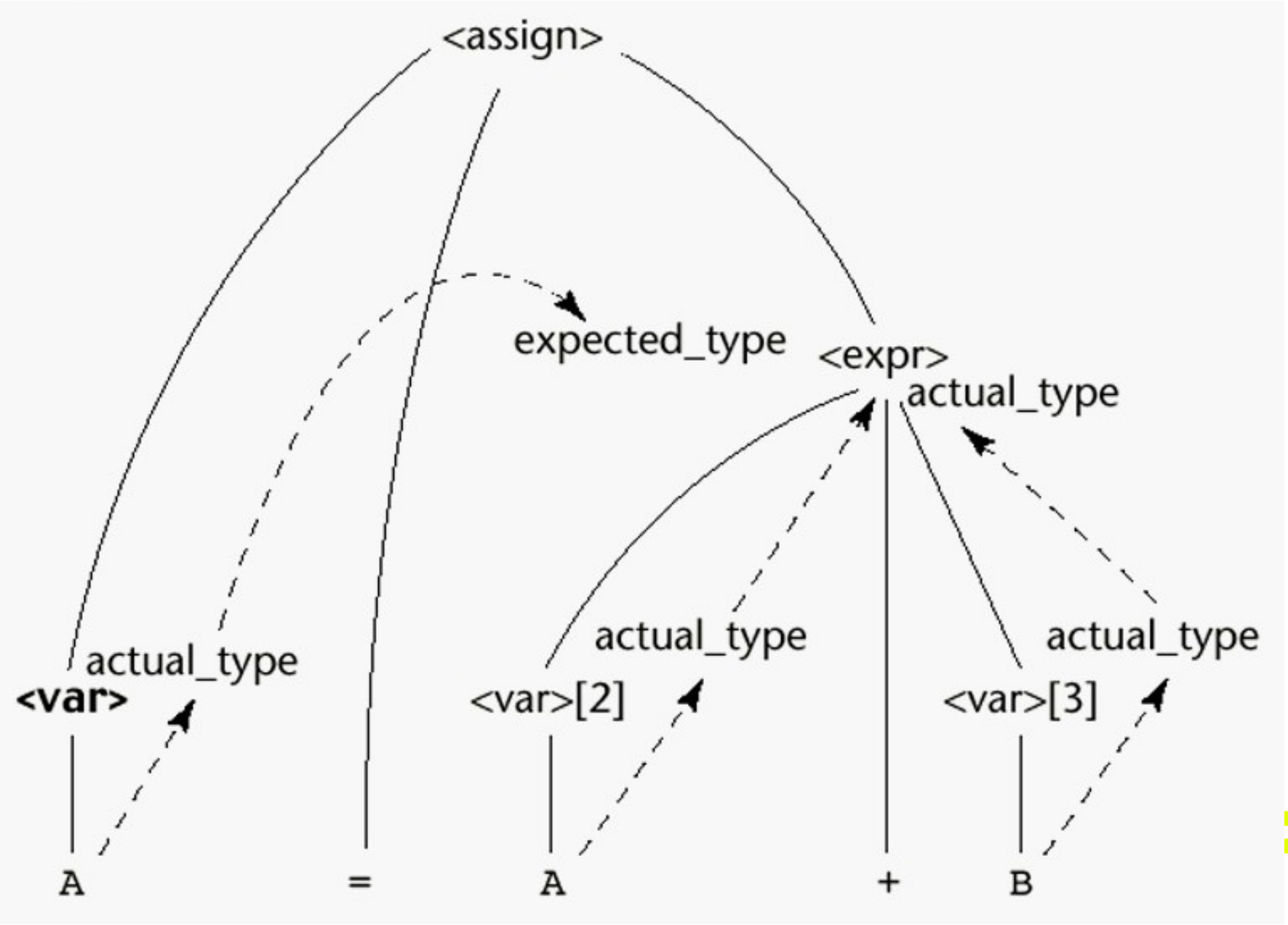
Gramática de atributos

- 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 axiomática

- *Processo de prova de programas:* a pós-condição do programa inteiro é o resultado desejado. Executar o program de trás para frente até a primeira instrução.Se a pré-condição do primeiro comando é a mesma que o programa espera, o programa está correto.
- *Um axioma para o comando de atribuição:*

  {Qx->E} x := E {Q}
- *A regra de conseqüência:*

  {P} S {Q}, P' => P, Q => Q'

  {P'} S {Q'}
- *Uma regra de inferência para seqüências:*
  - Para uma seqüência S1;S2:

    {P1} S1 {P2}

    {P2} S2 {P3}
  - A regra de inferência é:

    {P1} S1 {P2}, {P2} S2 {P3}

    {P1} S1; S2 {P3}

Semântica axiomática

- *Uma regra de inferência para loops*
  - Para o comando de loop:

    {P} while B do S end {Q}
  - A regra de inferência é:

    (I and B) S {I}

    {I} while B do S {I and (not B)}
  - onde I é a invariante do loop.
- Caracteristicas do loop invariant
- I must meet the following conditions:
- 1. P => I (the loop invariant must be true initially)
- 2. {I} B {I} (evaluation of the Boolean must not
- change the validity of I)
- 3. {I and B} S {I} (I is not changed by executing
- the body of the loop)
- 4. (I and (not B)) => Q (if I is true and B is false,
- Q is implied)
- 5. The loop terminates (this can be difficult to
- prove)

Semântica axiomática

-

- A*valiação da semântica axiomática:*

1. A definição de axiomas e regras de inferência para todos os comando de uma linguagem é uma tarefa complexa

2. é uma boa ferramenta para para pesquisar provas de exatidão de programas e uma excelente estrutura sobre a qual argumentar a respeito dos programas, mas não é tão útil para usuários da linguagem e construtores de compiladores

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

Semântica denotacional

O significado dos constructos da linguagem são definidos apenas pelos valores das variáveis do programa

A diferença entre a semântica denotacional e operacional: Na semântica operacional, as mudanças de estado são definidas pela codificação de algoritmos, enquanto que na semântica denotacional elas são definidas por funções matemáticas rigorosas

O estado de um programa é o valor de todas as suas variáveis

$$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$$

Seja VARMAP a fun'dão que, quando dado um nome de variável e um estado, retorna o valor corrente da variável

$$VARMAP(i_j, s) = v_j$$

Semântica denotacional

Numeros binários

<num_bin> → 0

| 1

| <num_bin> 0

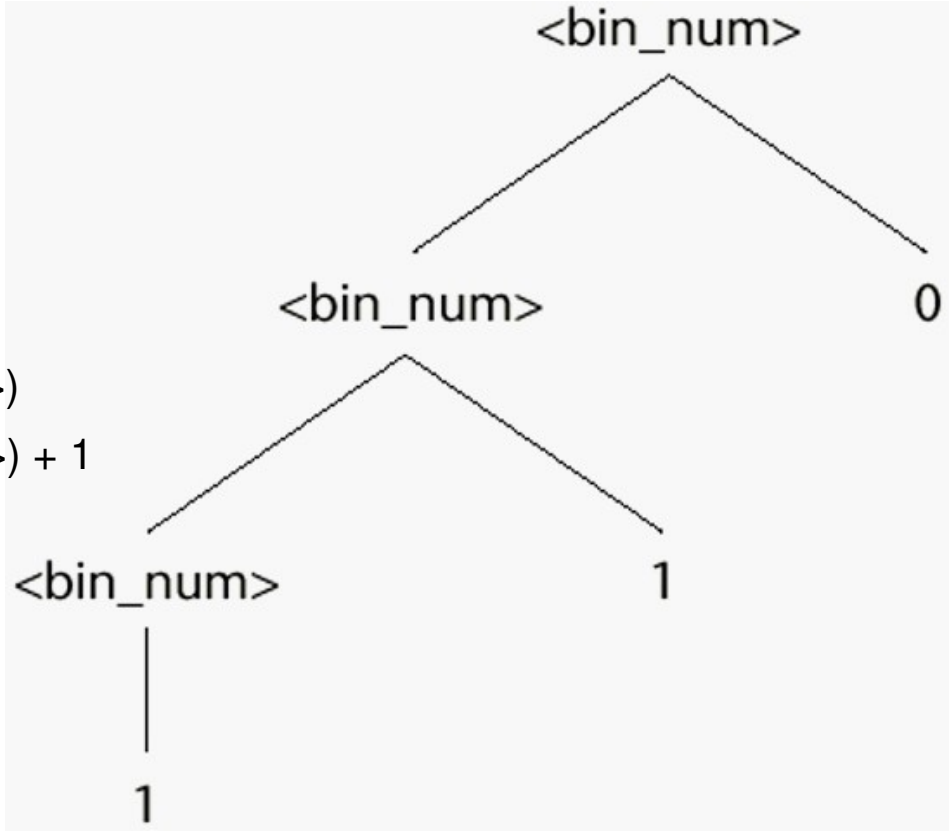
| <num_bin> 1

M_{bin}('0') = 0

M_{bin} ('1') = 1

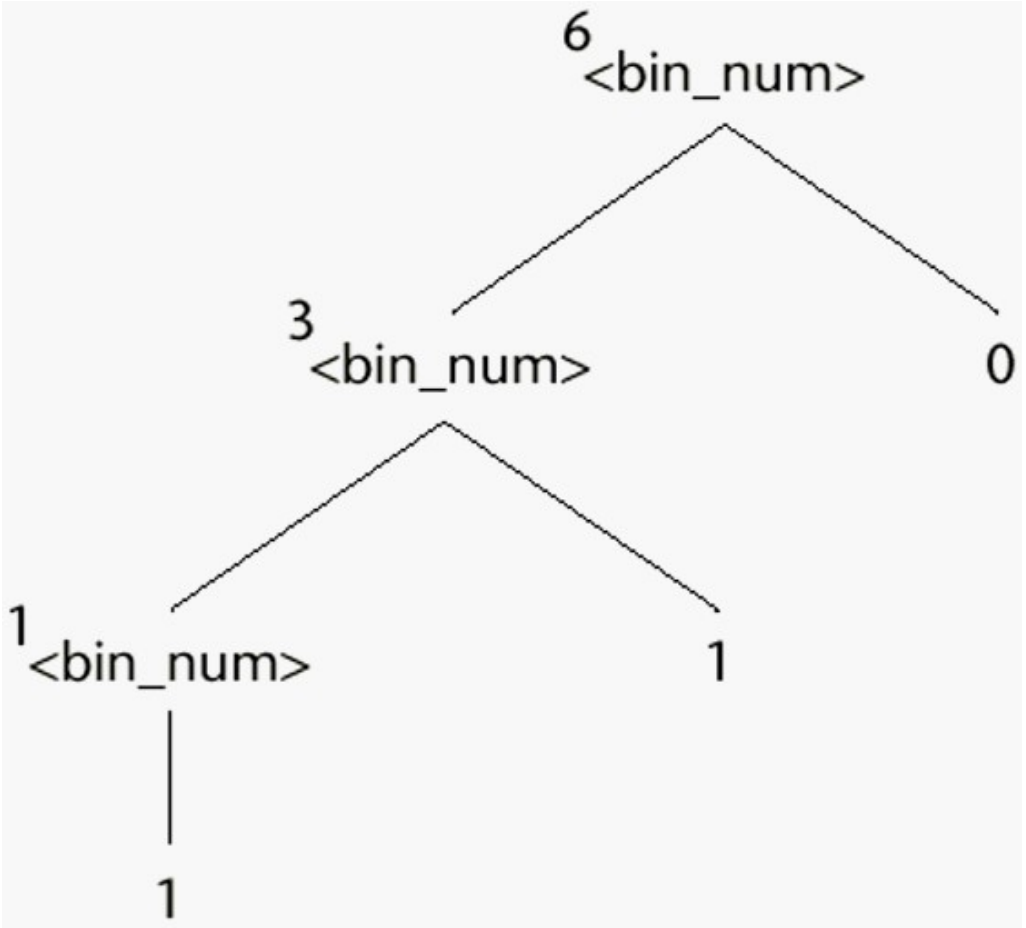
M_{bin} (<num_bin> '0') = 2 * M_{bin} (<num_bin>)

M_{bin} (<num_bin> '1') = 2 * M_{bin} (<num_bin>) + 1



Semântica denotacional

Uma árvore de análise com os objetos denotados para 110



Semântica denotacional

1. Decimal Numbers

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 |
5 | 6 | 7 | 8 | 9)

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Semântica denotacional

2. Expressões

M_e(<expr>, s) Δ=
  case <expr> of
    <dec_num> => M_{dec}(<dec_num>, s)
    <var> =>
        if VARMAP(<var>, s) = undef
            then error
            else VARMAP(<var>, s)
    <binary_expr> =>
        if (M_e(<binary_expr>.<left_expr>, s) = undef
            OR M_e(<binary_expr>.<right_expr>, s) =
                    undef)
            then error
            else
              if (<binary_expr>.<operator> = '+' then
                M_e(<binary_expr>.<left_expr>, s) +
                      M_e(<binary_expr>.<right_expr>, s)
              else M (<binary_expr> <left_expr> s) *

Semântica denotacional

3 Assignment Statements

$M_a(x := E, s)\ \Delta =$
  if $M_e(E, s) = error$
    then error
    else $s' = \{<i_1', v_1'>, <i_2', v_2'>, \ldots, <i_n', v_n'>\}$,
      where for $j = 1, 2, \ldots, n$,
        $v_j' = VARMAP(i_j, s)$ if $i_j <> x$
          $= M_e(E, s)$ if $i_j = x$

4 Logical Pretest Loops

$M_l(\text{while B do L}, s)\ \Delta =$
  if $M_b(B, s) = undef$
    then error
    else if $M_b(B, s) = false$
      then s
      else if $M_{sl}(L, s) = error$

Semântica denotacional

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors

- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

  - Recursion, when compared to iteration, is easier to describe with mathematical rigor

Semântica denotacional

-Avaliação da semântica denotacional

-Pode ser utilizada para prova da correção de programas

-Provê um modo rigoroso para discutir linguagens

-É um bom auxílio no projeto de linguagens

-Tem sido utilizado na pesquisa de sistemas geradores de compiladores

Chapter 3

- **Recursive Descent Parsing**

- **- Parsing is the process of tracing or constructing**
- **a parse tree for a given input string**
-
- **- Parsers usually do not analyze lexemes; that is**
- **done by a lexical analyzer, which is called by the**
- **parser**
-
- **- A *recursive descent parser* traces out a parse**
- **tree in top-down order; it is a top-down parser**
-
- **- Each nonterminal in the grammar has a**
- **subprogram associated with it; the subprogram**
- **parses all sentential forms that the nonterminal**
- **can generate**
-
- **- The recursive descent parsing subprograms are**
- **built directly from the grammar rules**
-
- **- Recursive descent parsers, like other top-down**
- **parsers, cannot be built from left-recursive**
- **grammars**

- Example: For the grammar:

- <term> -> <factor> {(* | /) <factor>}

- We could use the following recursive descent
- parsing subprogram (this one is written in C)

```
void term() {
  factor();  /* parse the first factor*/
  while (next_token == ast_code ||
        next_token == slash_code) {
    lexical();  /* get next token */
    factor();  /* parse the next factor */
  }
}
```

- Static semantics (have nothing to do with
- meaning)
- Categories:

1. Context-free but cumbersome (e.g. type checking)
2. Noncontext-free (e.g. variables must be declared before they are used)