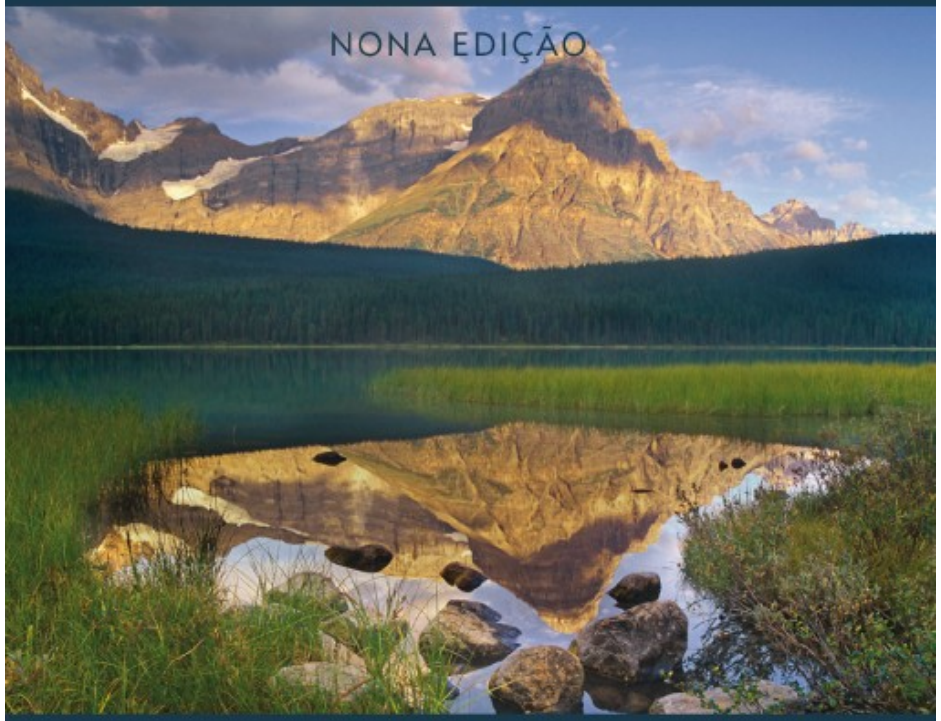
CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

NONA EDIÇÃO

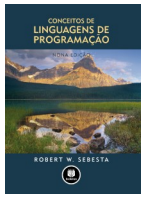


ROBERT W. SEBESTA

bookman

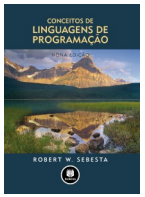
Capítulo 3

Descrevendo Sintaxe e Semântica



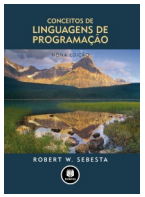
Tópicos do Capítulo 3

- Introdução
- O problema geral de descrever sintaxe
- Métodos formais para descrever sintaxe
- Gramáticas de atributos
- Descrevendo o significado de programas: semântica dinâmica



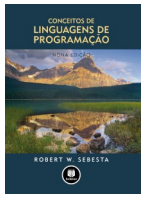
Introdução

- **Sintaxe:** a forma de suas expressões, sentenças e unidades de programas
- **Semântica:** o significado dessas expressões, sentenças e unidades de programas
- Sintaxe e semântica fornecem uma definição da linguagem
  - Usuários de uma definição de linguagem
    - Outros desenvolvedores de linguagem
    - Implementadores
    - Programadores (os usuários da linguagem)



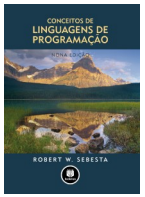
O problema geral de descrever sintaxe: terminologia

- Uma *sentença* é uma cadeia de caracteres formada a partir do alfabeto da linguagem
- Uma *linguagem* é uma cadeia de sentenças
- Um *lexema* é a unidade sintática de mais baixo nível de uma linguagem (por exemplo, `*`, `sum`, `begin`)
- Um *token* é uma categoria de lexemas (por exemplo, identificador)



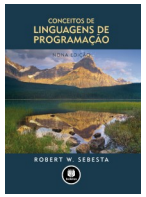
Definição formal de linguagens

- **Reconhecedores**
  - Um dispositivo de reconhecimento lê cadeias de caracteres a partir do alfabeto da linguagem e indica se a cadeia pertence ou não à linguagem
  - Exemplo: parte de análise sintática de um compilador
    - A discussão detalhada sobre análise sintática aparece no Capítulo 4
- **Geradores**
  - Um dispositivo que gera sentenças de uma linguagem
  - Pode determinar se a sintaxe de uma sentença em particular está sintaticamente correta a comparando à estrutura do gerador



BNF e gramáticas livres de contexto

- Gramáticas livres de contexto
  - Desenvolvido por Noam Chomsky no meio dos anos 1950
  - Geradores de linguagem, feitos para descrever a sintaxe de linguagens naturais
  - Define uma classe de linguagens chamadas de livres de contexto
- Forma de *Backus-Naur* (1959)
  - Inventada por John Backus para descriver Algol 58
  - BNF é equivalente às gramáticas livres de contexto

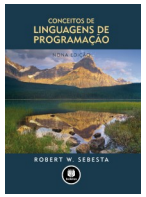


Fundamentos de BNF

- Em BNF, abstrações são usadas para representar classes de estruturas sintáticas – elas agem como variáveis sintáticas (também chamadas de símbolos não terminais, ou simplesmente não terminais)
- Terminais são lexemas ou tokens
- Uma regra tem um lado esquerdo (LHS), que é um não terminal, e um lado direito (RHS), que é uma cadeia de terminais e não terminais
  - Exemplos das regras de BNF:

    `<ident_list> → identifier | identifier, <ident_list>`

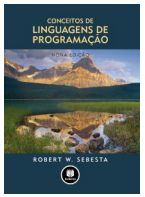
    `<if_stmt> → if <logic_expr> then <stmt>`
- Gramática: coleção de regras
- Um símbolo inicial é um elemento especial dos não terminais de uma gramática



Regras de BNF

- Uma abstração (ou símbolo não terminal) pode ter mais de um RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

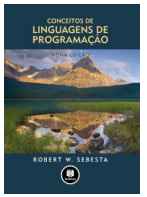


Descrevendo listas

- Listas sintáticas são descritas usando recursão

```
<ident_list> → ident
             | ident, <ident_list>
```

- Uma derivação é uma aplicação de regras repetida, começando com um símbolo inicial e terminando com uma sentença (todos símbolos terminais)



Um exemplo de gramática

<program> → <stmts>

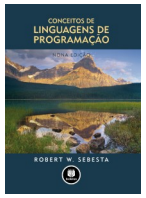
<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

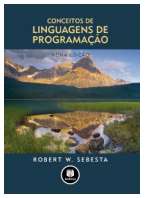
<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const



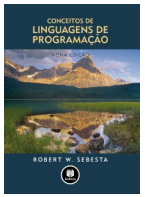
Um exemplo de derivação

```
<program> => <stmts>  => <stmt>
                      => <var> = <expr>
                      => a = <expr>
                      => a = <term> + <term>
                      => a = <var> + <term>
                      => a = b + <term>
                      => a = b + const
```



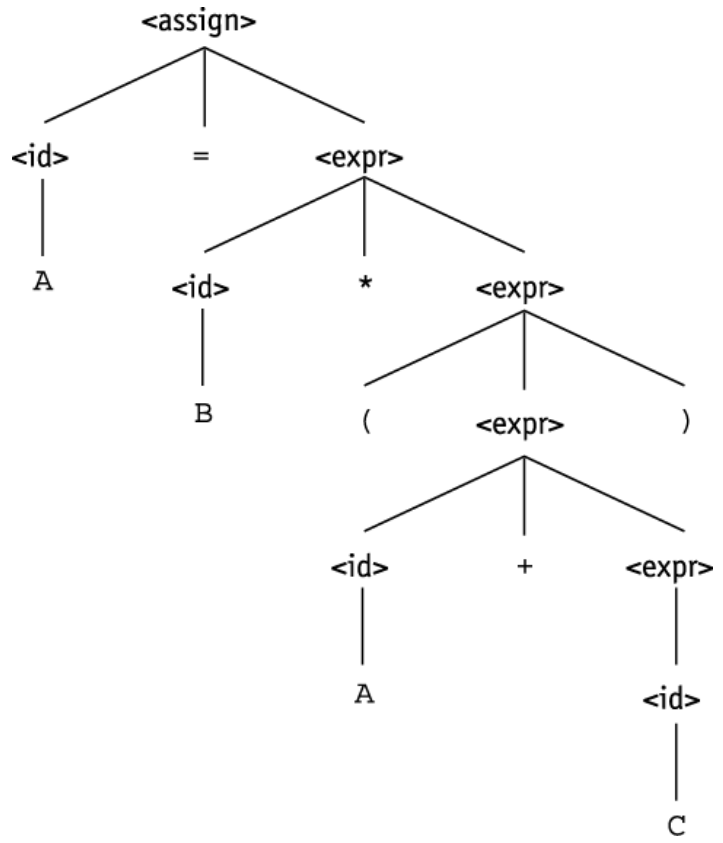
Derivações

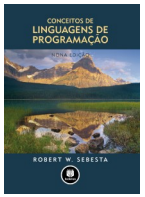
- Cada cadeia de símbolos em uma derivação é chamada de *forma sentencial*
- Uma *sentença* é uma forma sentencial que tem apenas símbolos terminais
- Uma *derivação mais à esquerda* é uma em que o não terminal mais à esquerda em cada forma sentencial é expandido
- Uma derivação pode ser nem mais à esquerda nem mais à direita



Árvore de análise sintática

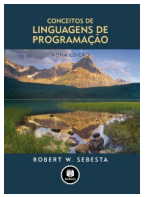
- Representação hierárquica de uma derivação





Ambiguidade em gramáticas

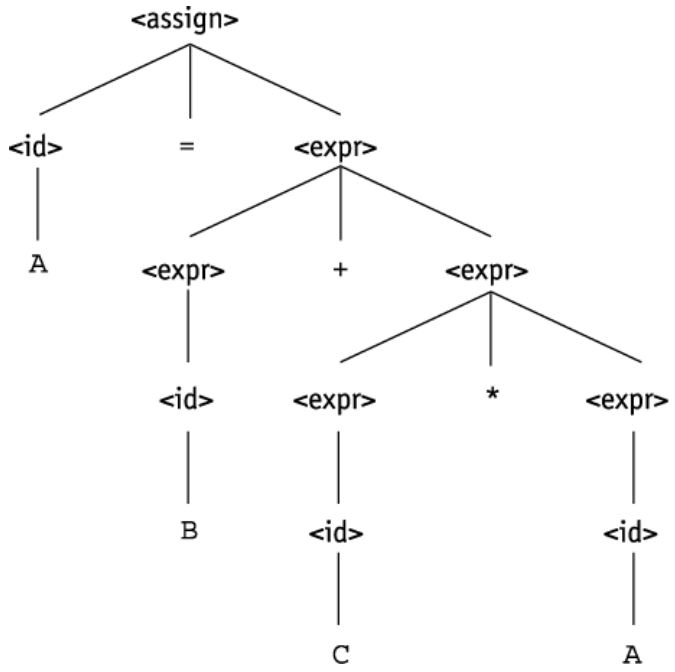
- Uma gramática é *ambígua* se gera uma forma sentencial com duas ou mais árvores de análise sintática distintas

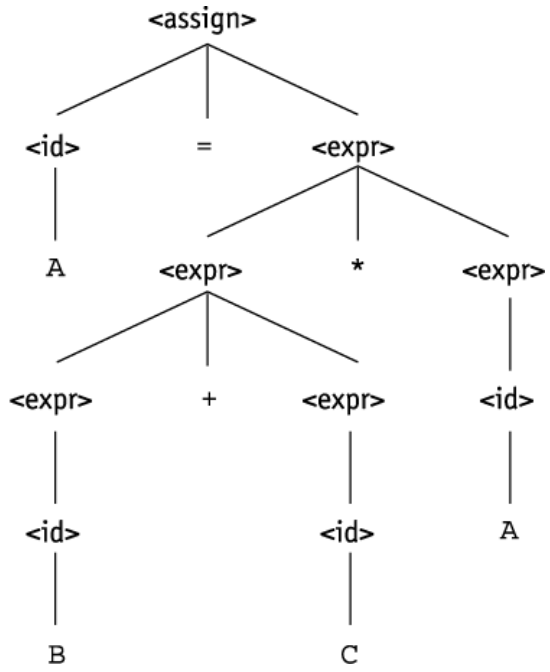


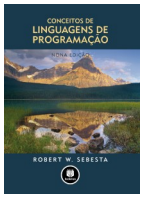
Uma gramática de expressão ambígua

<expr> → <expr> <op> <expr> | const

<op> → / | -



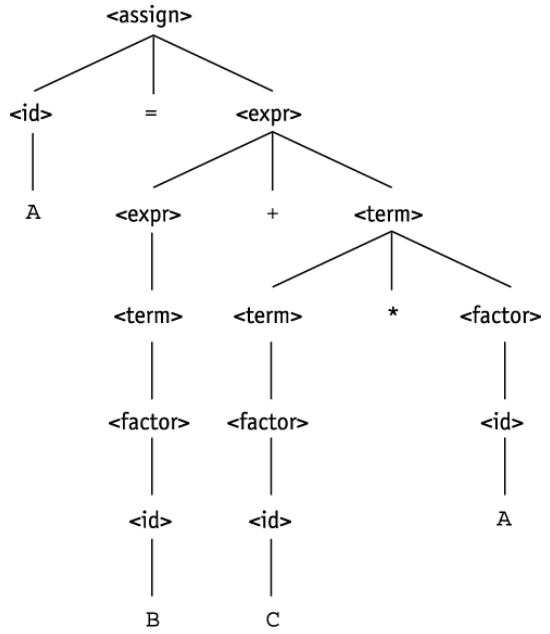


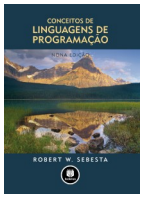


Uma gramática de expressão não ambígua

- Se usarmos a árvore de análise sintática para indicar níveis de precedência dos operadores, não podemos ter a ambiguidade

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```

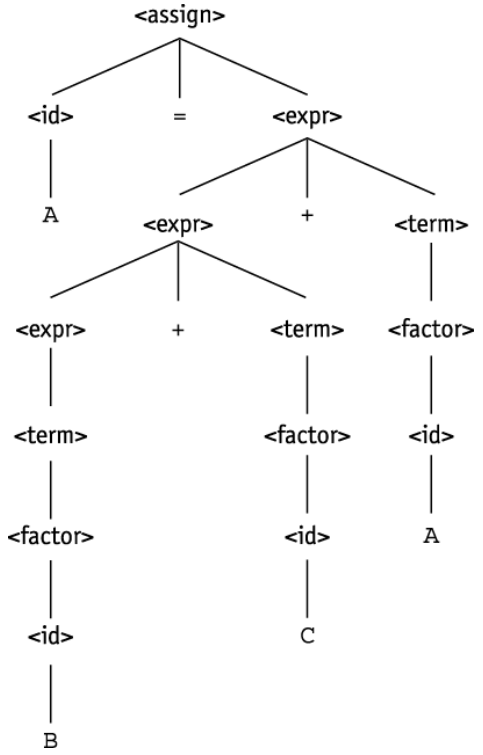


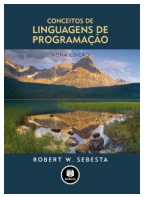


Associatividade de operadores

- Associatividade de um operador também pode ser indicada por uma gramática

```
<expr> -> <expr> + <expr> |  const  (ambígua)
<expr> -> <expr> + const  |  const  (não ambígua)
```





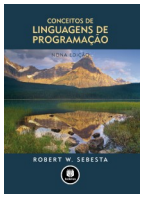
BNF estendida

- Partes opcionais são delimitadas por colchetes [ ]

  `<proc_call> -> ident [(<expr_list>)]`
- Partes alternativas de RHSs são colocadas entre parênteses e separadas com barras verticais

  `<term> → <term> (+|-) const`
- Repetições (0 ou mais) são colocadas entre chaves { }

  `<ident> → letter {letter|digit}`



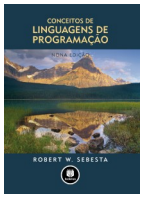
BNF e EBNF

- BNF

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

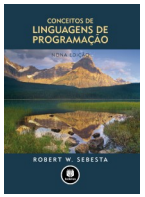
- EBNF

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



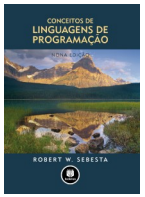
Recentes variações em EBNF

- RHSs alternativas são colocadas em linhas separadas
- Uso de dois pontos em vez de =>
- Uso de `opt` para partes opcionais
- Uso de `oneof` para escolhas



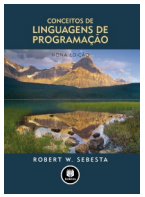
Semânticas estáticas

- Não têm a ver com o significado
- Gramáticas livres de contexto (CFGs) não podem descrever todas as sintaxes de linguagens de programação
- Categorias de construtores que são problemas:
  - Livres de contexto (por exemplo, tipos de operandos em expressões)
  - Não livres de contexto (por exemplo, variáveis precisam ser declaradas antes de serem usadas)



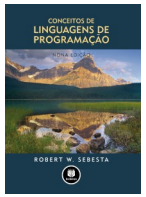
Gramáticas de atributos

- Uma gramática de atributos (AG) é uma extensão de uma gramática livre de contexto (CFG)
- Valores primários de gramáticas de atributos:
  - Especificação de semânticas estáticas
  - Compilação (verificação de semânticas estáticas)



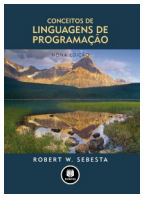
Gramáticas de atributos: definição

- Def: Uma gramática de atributo é uma gramática com os seguintes recursos adicionais:
  - Associado a cada símbolo X da gramática está um conjunto de atributos A (X)
  - Associado a cada regra gramatical está um conjunto de funções semânticas e um conjunto possivelmente vazio de funções de predicado sobre os atributos dos símbolos na regra gramatical



Gramáticas de atributos: definição

- Deixe $X_0 \rightarrow X_1 \ldots X_n$ ser uma regra
- Funções semânticas da forma $S(X_0) = f(A(X_1), \ldots, A(X_n))$ definem *atributos sintetizados*
- Funções da forma $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $i <= j <= n$, definem *atributos herdados*
- *Atributos intrínsecos* são atributos sintetizados de nós folha cujos valores são determinados fora da árvore de análise sintática



Gramáticas de atributos: um exemplo

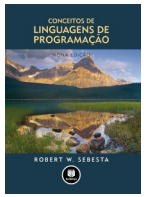
- Sintaxe

  `<assign> -> <var> = <expr>`

  `<expr> -> <var> + <var> | <var>`

  `<var> A | B | C`

- `actual_type`: sintetizado para `<var>` e `<expr>`
- `expected_type`: herdado para `<expr>`



Gramática de atributos

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

  Regra semântica:

  `<expr>.actual_type ← <var>[1].actual_type`

  Predicado:

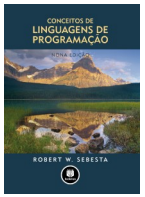
  `<var>[1].actual_type == <var>[2].actual_type`

  `<expr>.expected_type == <expr>.actual_type`

- Regra sintática : `<var> → id`

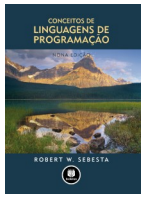
  Regra semântica:

  `<var>.actual_type ← lookup (<var>.string)`



Gramática de atributos

- Como são computados os valores dos atributos?
  - Se todos os atributos forem herdados, a árvore pode ser decorada na ordem decrescente.
  - Se todos os atributos forem sintetizados, a árvore pode ser decorada na ordem crescente.
  - Em muitos casos, ambos os tipos de atributos são usados, e uma combinação de crescente e decrescente precisa ser usada.

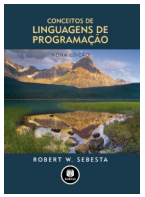


Gramática de atributos

<expr>.expected_type ← inherited from parent

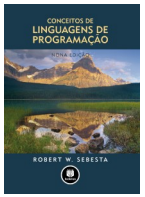
<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

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



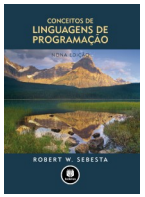
Semânticas

- Não há nenhuma notação única amplamente aceitável ou formalismo para descrever semântica
- Diversas necessidades de uma metodologia e notação para a semântica
  - Programadores precisam saber o que a sentença de uma linguagem significa
  - Desenvolvedores de compiladores devem saber exatamente o que as construções da linguagem significam
  - Geradores de compilação seriam possíveis
  - Projetistas poderiam detectar ambiguidades e inconsistências



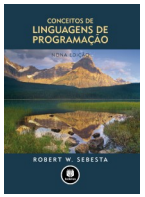
Semântica operacional

- Semântica Operacional
  - Descrever o significado de uma sentença ou programa pela especificação dos efeitos de rodá-lo em uma máquina. A mudança no estado de máquina (memória, registros, etc.) define o significado da sentença
- Para usar semântica operacional em uma linguagem de alto nível, é necessária uma máquina virtual



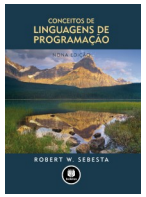
Semântica operacional

- Um interpretador puro de *hardware* seria muito caro
- Um interpretador puro de *software* também tem problemas
  - As características detalhadas de um computador em particular deixaria ações difíceis de entender
  - Uma definição semântica seria dependente da máquina



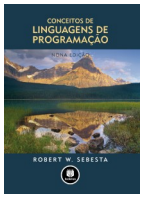
Semântica operacional

- Uma melhor alternativa: uma completa simulação
- O processo:
  - Construir um interpretador (transforma código de origem em código de máquina de um computador idealizado)
  - Construir um simulador para o computador idealizado



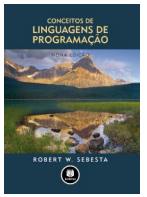
Semânticas operacionais

- Usos de semânticas operacionais:
  - Manuais de linguagem e livros-texto de programação
  - Ensino de linguagens de programação
- Dois níveis diferentes de uso de uma semântica operacional:
  - Semântica operacional natural
  - Semântica operacional estrutural
- Avaliação
  - Boa se usada informalmente
  - Extremamente complexa se usada formalmente



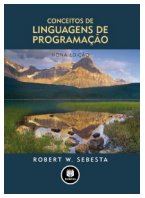
Semânticas denotacionais

- Baseada na teoria de funções recursivas
- O método de descrição semântica mais abstrato
- Originalmente desenvolvida por Scott e Strachey (1970)



Semânticas denotacionais

- O processo de construir uma especificação denotacional para uma linguagem:
  - Define um objeto matemático para cada entidade da linguagem
  - Define uma função que mapeia as instâncias dessa entidade de linguagem para instâncias do objeto matemático correspondente
- O significado da construção de linguagem é definido apenas pelos valores das variáveis de programas



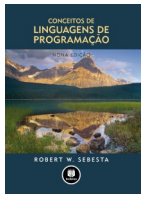
Semânticas denotacionais: o estado de um programa

- O estado de um programa são os valores atuais de todas as suas variáveis

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- Considere **VARMAP** uma função que, quando dado um nome de variável e estado, retorna o atual valor da variável

$$\text{VARMAP}(i_j, s) = v_j$$



Números decimais

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

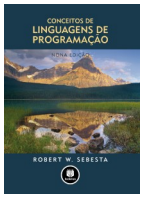
M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

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

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

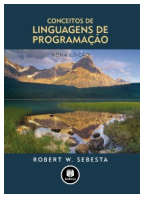
…

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



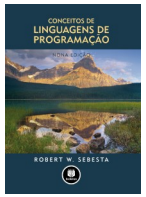
Expressões

- Expressões formam o conjunto Z ∪ {error}
- Assumimos que expressões são números decimais, variáveis ou expressões binárias tendo um operador aritimético e dois operando, cada um pode ser uma expressão



Expressões

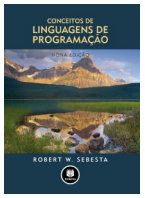
```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
      <binary_expr> =>
           if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                                 undef)
               then error
           else
           if (<binary_expr>.<operator> == '+' then
              Me(<binary_expr>.<left_expr>, s) +
                     Me(<binary_expr>.<right_expr>, s)
           else Me(<binary_expr>.<left_expr>, s) *
               Me(<binary_expr>.<right_expr>, s)
       ...
```



Sentenças de atribuição

- Funções de mapeamento mapeiam listas de sentenças e estados para estados ∪ {error}

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

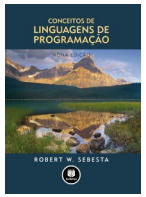


Laços lógicos com pré-teste

- Funções de mapeamento mapeiam listas de sentenças e estados para estados U {error}

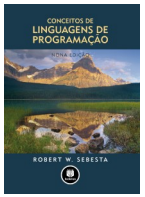
M_l(while B do L, s) Δ=

```
    if Mb(B, s) == undef
        then error
        else if Mb(B, s) == false
            then s
            else if Msl(L, s) == error
                  then error
                  else Ml(while B do L, Msl(L, s))
```



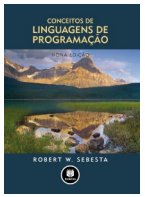
Significado do laço

- O significado do laço é o valor das variáveis do programa após as sentenças no laço terem sido executadas o número de vezes necessário, assumindo que não ocorreram erros
- Essencialmente, o laço foi convertido de iteração para recursão, onde o controle da recursão é definido matematicamente por outras funções recursivas de mapeamento de estado
  - A recursão é mais fácil de descrever com rigor matemático do que a iteração



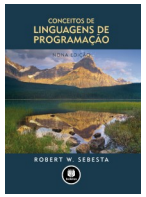
Avaliação de semântica denotacional

- Pode ser usada para provar a corretude de programas
- Sugere uma maneira rigorosa para pensar nos programas
- Pode ser uma ajuda ao projeto de linguagem
- Tem sido usada em sistemas de geração de compilador
- Devido a sua complexidade, é de pouco uso para usuários de linguagem



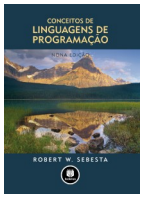
Semântica axiomática

- Baseada em lógica matemática
- Propósito original: verificação de programas
- Axiomas ou regras de inferência são definidas para cada tipo de sentença da linguagem (a fim de permitir transformações de expressões de lógica em expressões de lógica mais formais)
- As expressões lógicas são chamadas de *asserções*



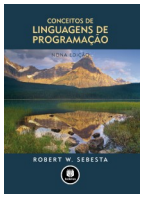
Semântica axiomática

- Uma asserção que precede imediatamente uma sentença (uma *pré-condição*) descreve as restrições nas variáveis do programa naquele ponto.
- Uma asserção imediatamente após uma sentença é uma *pós-condição*
- Uma *pré-condição mais fraca* é a menos restritiva que garantirá a validade da pós-condição associada



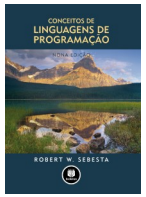
Forma da semântica axiomática

- Forma sentencial: `{P} statement {Q}`
- Um exemplo
  - `a = b + 1  {a > 1}`
  - Uma pré-condição possível: `{b > 10}`
  - Pré-condição mais fraca: `{b > 0}`



Provas de programas

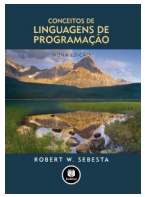
- A pós-condição para o programa completo é o resultado desejado
  - Volte até a primeira sentença do programa. Se a pré-condição na primeira sentença é a mesma da especificação do programa, ele está correto.



Semântica axiomática: axiomas

- Um axioma para sentença de atribuição (x = E): $\{Q_{x->E}\}$ x = E {Q}
- A Regra de Consequência:

$$\frac{\{P\}\ S\ \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$



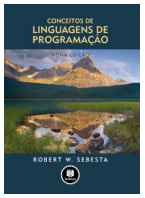
Semântica axiomática: axiomas

- Uma regra de inferência para sequências da forma S1; S2

  {P1} S1 {P2}

  {P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$



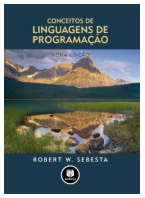
Semântica axiomática: axiomas

- Uma regra de inferência para um laço de repetição

  {P} while B do S end {Q}

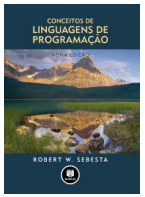
$$\frac{(\text{I and B})\ (\text{S}\ \{\text{I}\}}{\{\text{I}\}\ \texttt{while}\ \text{B}\ \texttt{do}\ \text{S}\ \texttt{end}\ \{\text{I and (not B)}\}}$$

onde I é a invariante de laço de repetição



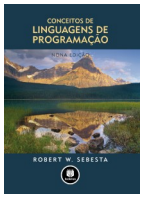
Semântica axiomática: axiomas

- Características da invariante de laço de repetição: I deve satisfazer os seguintes requisitos:
  - P => I -- a invariante de repetição precisa ser verdadeira inicialmente
  - {I} B {I} -- avaliação de B não pode mudar a validade de I
  - {I and B} S {I} -- I não é mudado pela execução do corpo da repetição
  - (I and (not B)) => Q -- se I é verdadeiro e B é falso, Q é incluído
  - A repetição termina -- pode ser difícil de provar



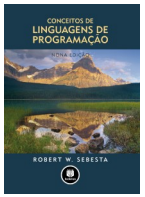
Invariante de repetição

- A invariante de repetição é uma versão enfraquecida da pós-condição de repetição e é também uma pré-condição.
- I deve ser fraco o suficiente para estar preenchido antes do início do ciclo, mas quando for combinado com a condição de saída do laço deve ser suficientemente forte para forçar a verdade da pós-condição



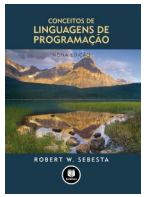
Avaliação da semântica axiomática

- Desenvolver axiomas ou regras de inferência para todas as sentenças de uma linguagem é difícil
- É uma boa ferramenta para provas de corretude, mas não é útil para usuários de linguagens e escritores de compiladores



Semântica denotacional × semântica operacional

- Na semântica operacional, as mudanças de estado são definidas por algoritmos codificados
- Na semântica denotacional, as mudanças de estado são definidas por funções matemáticas rigorosas



Resumo

- BNF e gramáticas livres de contexto são metalinguagens equivalentes
  - Bastante adequadas para a tarefa de descrever a sintaxe de linguagens de programação
- Uma gramática de atributos e um formalismo descritivo que pode descrever tanto a sintaxe quanto a semântica estática de uma linguagem
- Três métodos principais de descrição semântica
  - Operacional, denotacional e axiomática