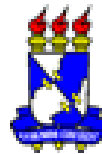


# Processamento e Especificação de Linguagens de Programação

Prof. Alberto Costa Neto  
alberto@ufs.br

Linguagens de Programação



Departamento de Computação  
Universidade Federal de Sergipe

# Introdução

- LPs são ferramentas-chave dos programadores
- LP = notação formal para expressar algoritmos
- Algoritmo = conceito abstrato independente
  - linguagem natural, diagramas, rabiscos, etc...
- Entretanto, sem uma notação formal não podemos compartilhar, verificar a corretude, e... ... principalmente, fazer a máquina entender as instruções para executar o que queremos



# Introdução

- **Processador de Linguagem** = traduz a notação da linguagem usada para expressar um algoritmo em um conjunto de instruções primitivas da máquina (código de máquina)



# Introdução

- Exemplo: Programa para calcular área de um triângulo

$$area = \sqrt{(s(s-a)(s-b)(s-c))}$$

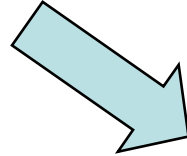
$$\text{onde, } s = \frac{(a+b+c)}{2}$$



# Introdução

## código de máquina

0000 0001 0110 1110  
0100 0000 0001 0010  
1100 0000 0000 1101



## código assembly

...

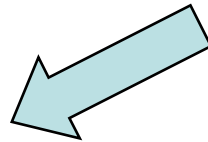
LOAD R1 a; ADD R1 b; ADD R1 c; DIV R1 #2;  
LOAD R2 R1;  
LOAD R3 R1; SUB R3 a; MULT R2 R3;  
LOAD R3 R1; SUB R3 b; MULT R2 R3;  
LOAD R3 R1; SUB R3 c; MULT R2 R3;  
LOAD R0 R2; CALL sqrt

Processador



## código alto-nível

let s = (a+b+c)/2  
in sqrt(s\*(s-a)\*(s-b)\*(s-c))



# Introdução

## linguagens de alto-nível

- Conceitos típicos:
  - Expressões
  - Tipos
  - Comandos
  - Declarações
  - Abstração
  - Encapsulamento
  - ...



# Introdução

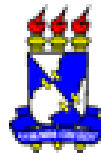
- Tipos de Processadores de Linguagem
  - **Tradutor**: traduz o código de uma linguagem para outra
    - **Caso particular: Compilador**
      - Traduz o código escrito numa linguagem de alto nível para uma linguagem de baixo nível (usualmente, ling. de máquina)
  - **Interpretador**: executa diretamente o código expresso numa determinada linguagem sem estágio intermediário
  - **Híbrido**: compila e interpreta
    - Ex: Java



# Especificação de Linguagens

Prof. Alberto Costa Neto  
alberto@ufs.br

Linguagens de Programação



Departamento de Computação  
Universidade Federal de Sergipe



# Especificação de Linguagens

- **O que é?**
  - A "cara" de um programa "bem formado"
    - **Sintaxe:** quais os símbolos existentes e como podem ser combinadas
    - **Restrições de contexto:** determinar se uma expressão é bem formada depende do contexto
  - Qual o significado de um programa "bem formado"
    - **Semântica:** qual o significado real dos símbolos



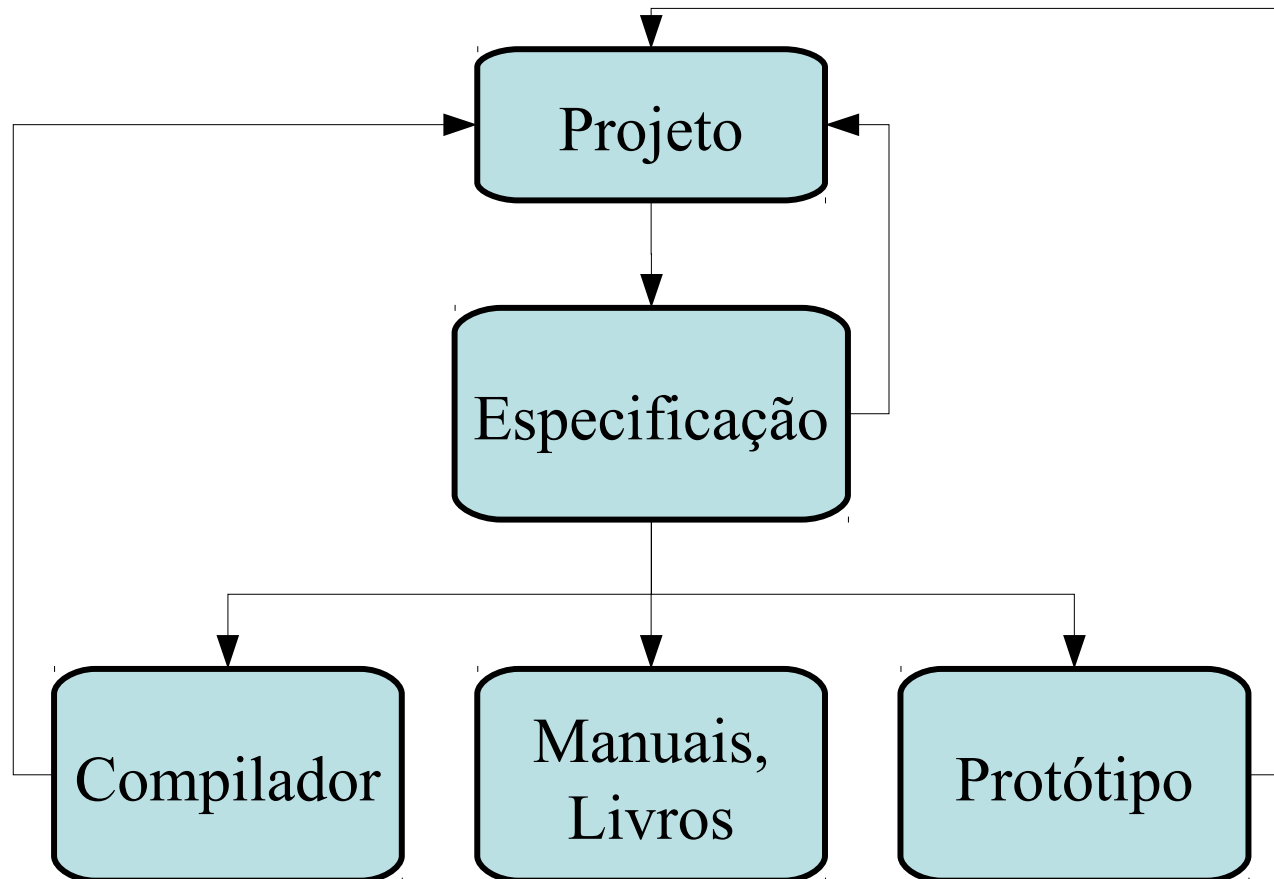
# Especificação de Linguagens

- **Por quê?**

- Dispositivo de comunicação entre pessoas que precisam ter um entendimento comum de uma determinada linguagem de programação
  - Projetista da linguagem
  - Implementador da linguagem
  - Usuário da linguagem (programadores)



# Ciclo de Vida de uma LP



# Especificação de Linguagens

- **Como especificar?**
  - Especificação Formal: uso de um tipo de formalismo preciso
  - Especificação Informal: descrição em uma língua qualquer (ex: inglês)
  - Geralmente, uma mistura (ex: Java)
    - Sintaxe: especificação formal com Context-free Grammar (CFG)
    - Restrição contextual e Semântica: informal
    - Obs: já existe semântica formal de Java!



# Especificação de Linguagens

- Sintaxe é a parte **mais visível** da linguagem
- Paradigma da Linguagem é a **segunda parte mais visível**
  - A escolha do paradigma depende da forma como o programador modela o problema
  - Não existe o modelo "certo" de se programar; apenas modelos diferentes adequados a diferentes tipos de problemas
- A parte **menos visível** é a semântica da linguagem
  - Linguagem que possui uma semântica bem definida normalmente permite implementações claras e precisas



# Sintaxe

- Especificada através de Gramáticas Livre de Contexto (**Context Free Grammars – CFG**)
  - Conjunto finito de símbolos **terminais**
  - Conjunto finito de símbolos **não-terminais**
    - Entre eles, um **símbolo de início**
  - Conjunto finito de **regras de produção**



# Sintaxe

- Normalmente CFGs são escritas com a notação **BNF (Backus-Naur Form)**
- Regra de produção em BNF:

$N ::= \alpha$

onde  $N$  é um não-terminal;

$\alpha$  é sequência de terminais e não-terminals; e

$::=$  significa "consiste de"

$N ::= \alpha \mid \beta \mid \dots$

é o mesmo que definir várias regras com  $N$



# Sintaxe

- Cada CFG define uma linguagem, que nada mais é que um conjunto de strings de símbolos terminais.

- Exemplo:

Inicio ::= Letra

| Inicio Letra

| Inicio Digito

Letra ::= **a** | **b** | **c** | **d** | ... | **z**

Digito ::= **0** | **1** | **2** | ... | **9**





# Sintaxe

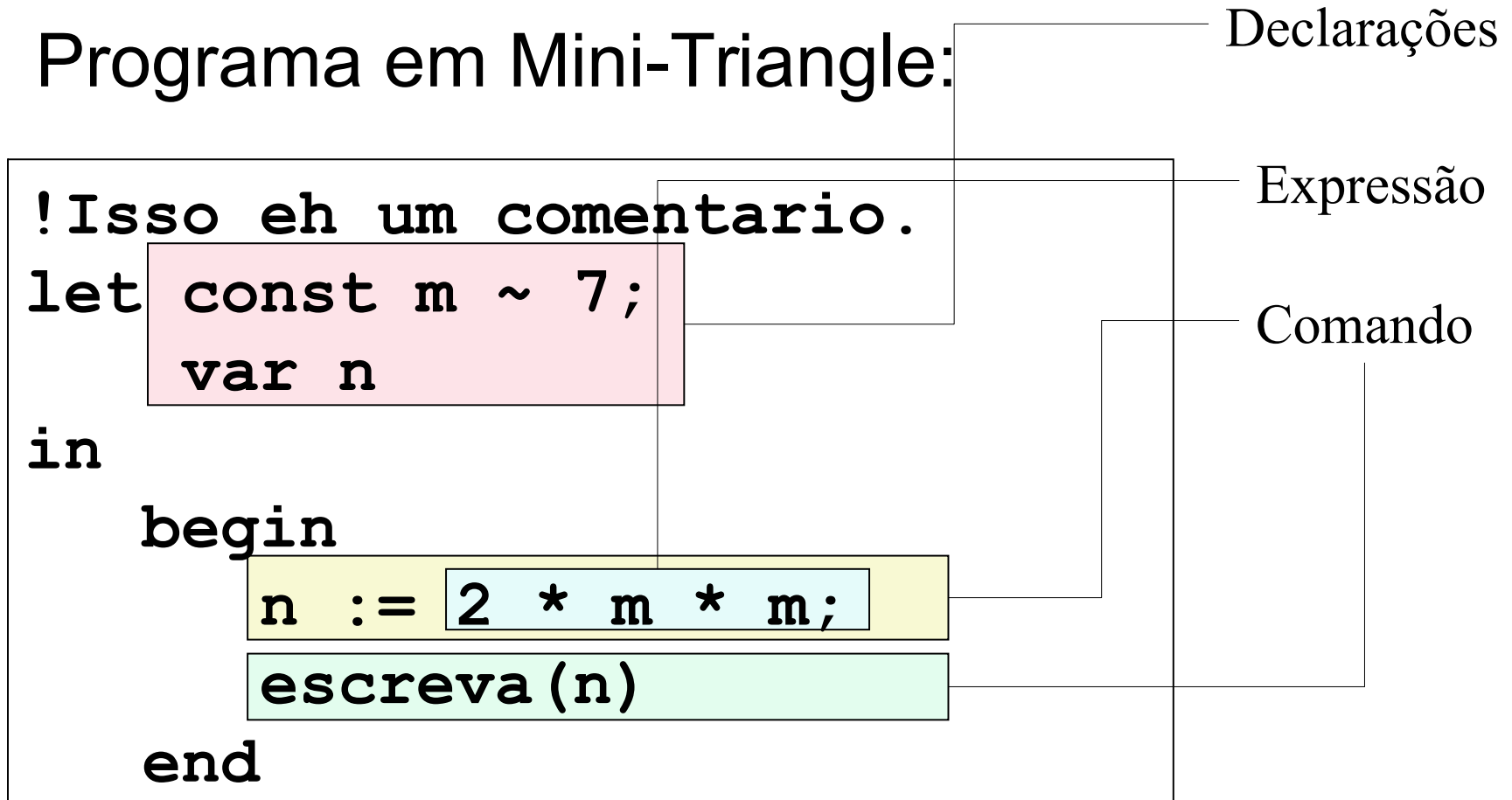
- Derivação: geração de uma sentença, ou aplicação de uma produção na linguagem

```
Inicio => Inicio Digito
      => Inicio Digito Digito
      => Inicio Digito 1
      => Inicio 1 1
      => Inicio Letra 1 1
      => Inicio Letra Letra 1 1
      => Letra Letra Letra 1 1
      => Letra Letra s 1 1
      => Letra f s 1 1
      => u f s 1 1
```



# Sintaxe::Linguagem “Mini-Triangle”

- Linguagem bem simples
- Programa em Mini-Triangle:



# Sintaxe::CFG de "Mini-Triangle"

```
Program ::= Single-Command
Single-Command ::= V-name := Expression
                | Identifier ( Expression )
                | if Expression then Single-Command
                  else Single-Command
                | while Expression do Single-Command
                | let Declaration in Single-Command
                | begin Command end
Command ::= Single-Command
          | Command ; Single-Command
...

```



# Sintaxe::CFG de "Mini-Triangle"

```
Expression ::= Primary-Expression  
            | Expression Operator Primary-Expression  
Primary-Expression ::= Integer-Literal  
                    | V-name  
                    | Operator Primary-Expression  
                    | ( Expression )  
V-name ::= Identifier  
Identifier ::= Letter  
            | Identifier Letter  
            | Identifier Digit  
Integer-Literal ::= Digit  
                | Integer-Literal Digit  
Operator ::= + | - | * | / | < | > | =
```



# Sintaxe::CFG de "Mini-Triangle"

```
Declaration ::= single-Declaration  
              | Declaration ; Single-Declaration  
Single-Declaration  
  ::= const Identifier ~ Expression  
      | var Identifier : Type-denoter  
Type-denoter ::= Identifier
```

```
Comment ::= ! CommentLine eol  
CommentLine ::= Graphic CommentLine  
Graphic ::= qualquer caracter que possa ser impresso
```



# Sintaxe::Árvore Sintática

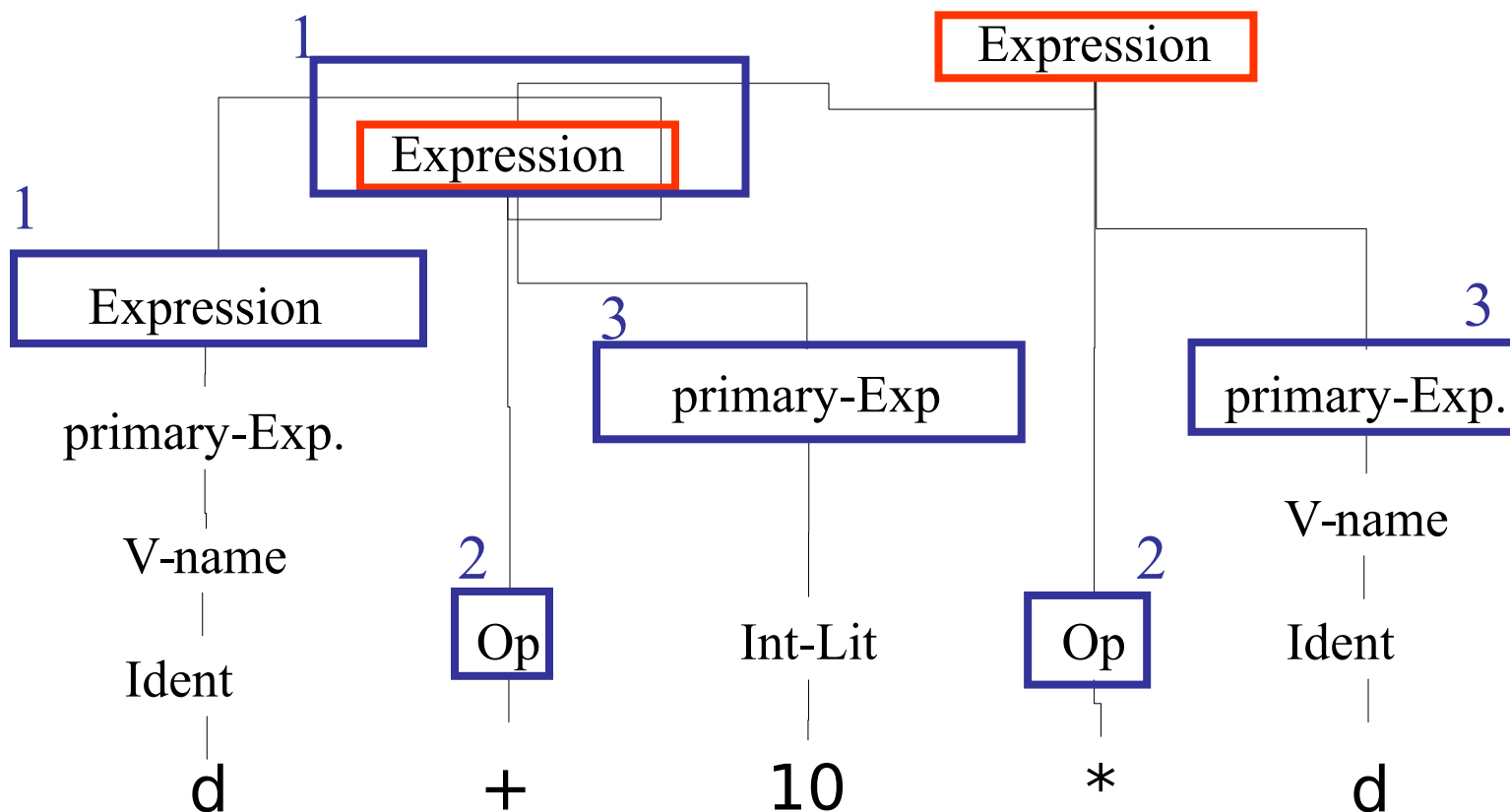
- Árvore ordenada de rótulos de uma CFG:
  - a) Nós terminais (folhas) são rotulados por símbolos terminais
  - b) Nós não-terminais (nós internos) são rotulados por símbolos não-terminais
  - c) Cada nó não-terminal rotulado por  $N$  tem filhos  $X_1, X_2, \dots, X_n$  (nessa ordem) tal que  $N ::= X_1 \dots X_n$  é uma regra de produção.



# Sintaxe::Árvore Sintática

Exemplo: Arv. Sint. de uma Expression de MT

**Expression** ::= <sup>1</sup>Expression <sup>2</sup>Op <sup>3</sup>primary-Exp



# Restrições de Contexto

- Regras Sintáticas sozinhas não são suficientes para especificar o formato de programas bem-formados

## Exemplo 1:

```
let const m~2  
in m + x
```

**Indefinido!**



Regras de Escopo

## Exemplo 2:

```
let const m~2 ;  
    var n:Boolean  
in begin  
    n := m<4;  
    n := n+1  
end
```

**Erro de Tipo!**



Regras de Tipo





# Restrições de Contexto

- **Regras de Escopo:**

- Regulam a visibilidade de identificadores.
- Relacionam toda **ocorrência de aplicação** de um identificador a uma **ocorrência de vínculo**

## Exemplo 1

let const **m** ~ 2;  
var **r** : Integer  
in  
**r** := 10 \* **m**

**Vínculo** (circles around **m** and **r**)

**Aplicação** (circle around **m** in the expression)

## Exemplo 2:

let const **m** ~ 2  
in **m** + **x**

**Vínculo** (circles around **m** and **x**)

**Aplicação** (circle around **m** in the expression)

**?** (question mark above the expression)

**Conceito associado:**

*Vínculo Estático vs. Vínculo Dinâmico*



# Restrições de Contexto

**Regras de Tipo:** Regem a definição do tipo esperado do operando e o tipo do resultado da operação

**Exemplo1:**  $E1 < E2$

Regra de Tipo do  $<$  é:

se  $E1$  e  $E2$  são do tipo int, então o resultado é do tipo boolean

**Exemplo2:** **while**  $E$  **do**  $C$

Regra de Tipo do **while** é:

$E$  deve ser do tipo boolean

**Conceito associado:**

*Tipagem Estática vs. Tipagem Dinâmica*



# Semântica

- Especificação da semântica está ligada com a especificação do "significado" de programas bem-formados
- **Conceitos relacionados:**
  - **Expressões** são **avaliadas** e **geram valores** (pode ter o efeito colateral de atualizar variável)
  - **Comandos** são **executados** para **atualizar variável** (pode ter o efeito colateral de executar operações de entrada/saída)
  - **Declarações** são **elaboradas** para **produzir vínculos** (pode ter o efeito colateral de alocar e inicializar variáveis)



# Semântica::Expressões

A expressão  $E1 \ O \ E2$  é avaliada da seguinte forma:

- 1)  $E1$  gera valor  $v$
- 2)  $E2$  gera valor  $w$
- 3) operação binária  $O$  é aplicada a  $v$  e  $w$



# Semântica::Declarações

A declaração de constante `const I~E` é elaborada da seguinte forma:

- 1) `E` gera um valor `v`
- 2) identificador `I` é vinculado a `v`

A declaração de variável `var I:T` é elaborada da seguinte maneira:

- 1) uma nova variável é alocada na memória
- 2) valor inicial é indefinido
- 3) `I` é vinculado a essa nova variável
- 4) A variável será desalocada na saída do comando `let` contendo a declaração

A declaração sequencial `D1 ; D2` é elaborada da seguinte forma:

- 1) `D1` é elaborada e gera vínculos `g`
- 2) `D2` é elaborada e gera vínculos `h` levando `g` em consideração



# Semântica::Comandos

O comando de atribuição  $V := E$  é executado da seguinte maneira:

- 1) a expressão  $E$  é avaliada e gera um valor  $v$
- 2) depois,  $v$  é atribuído à variável de nome  $V$

O comando sequencial  $C1; C2$  é executado da seguinte maneira:

- 1) o comando  $C1$  é executado
- 2) depois, o comando  $C2$  é executado.

O comando condicional  $\text{if } E \text{ then } C1 \text{ else } C2$  é executado da seguinte maneira:

- 1) a expressão  $E$  é avaliada e gera um valor-verdade  $t$
- 2) se  $t$  é verdadeiro,  $C1$  é executado
- 3) se  $t$  é falso,  $C2$  é executado.

O comando  $\text{let } D \text{ in } C$  é executado da seguinte maneira:

- 1) a declaração  $D$  é elaborada e produz vínculos  $b$
- 2)  $C$  é executado no contexto do comando  $\text{let}$
- 3) os vínculos  $b$  não possuem efeito fora do  $\text{let}$



# Semântica Operacional

- Descreve o significado de um programa executando-o em uma máquina (real ou não)
- Representada pelas alterações feitas pelo programa ao estado da máquina (registradores, memória, I/O).



# Semântica Operacional

- Exemplo: Semântica da construção **for** da linguagem C

**// Comando em C**

```
for (exp1; exp2; exp3)
{
    ...
}
```

**// Semântica Operacional**

```
exp1;
loop: if exp2 = 0 goto out
    ...
exp3;
goto loop
out: ...
```





# Semântica Denotacional

- **Método mais rigoroso** de descrição de significado de programas. Baseada na teoria de funções recursivas
- **Bastante complexa e muito pouco utilizada** nas linguagens conhecidas
- Funciona através da definição de um **objeto matemático e uma função para cada entidade da linguagem**
- Esta função **mapeia instâncias da entidade em instâncias do objeto matemático**
- Pode funcionar como um auxílio ao design de linguagens.
  - Ex: Quando a SD de uma construção da linguagem parecer muito complexa, pode significar a necessidade de rever a sintaxe desta construção



$$[\text{ass}_{\text{ns}}] \quad \langle x := a, s \rangle \rightarrow s[x \mapsto \mathcal{A}[a]s]$$

$$[\text{skip}_{\text{ns}}] \quad \langle \text{skip}, s \rangle \rightarrow s$$

$$[\text{comp}_{\text{ns}}] \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}$$

$$[\text{if}_{\text{ns}}^{\text{tt}}] \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \quad \text{if } \mathcal{B}[b]s = \text{tt}$$

$$[\text{if}_{\text{ns}}^{\text{ff}}] \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \quad \text{if } \mathcal{B}[b]s = \text{ff}$$

$$[\text{while}_{\text{ns}}^{\text{tt}}] \quad \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \quad \text{if } \mathcal{B}[b]s = \text{tt}$$

$$[\text{while}_{\text{ns}}^{\text{ff}}] \quad \langle \text{while } b \text{ do } S, s \rangle \rightarrow s \quad \text{if } \mathcal{B}[b]s = \text{ff}$$



# Sugestões de Leitura

- Concepts of Programming Languages  
(Robert Sebesta)
  - Capítulo 3

