

Linguagens de Programação

Lista de exercícios 2

Luís Felipe Ramos Ferreira

2019022553

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

lframoss_ferreira@outlook.com

1. Semântica formal

1.1. $z := 1; \text{while } \neg(y = 0) \text{ do } (z := z * x, y := y - 1)$

1.2.

$$\text{2. } B[\Sigma \neg(x = 1)]_s = \begin{cases} \text{tt if } B[x = 1]_s = \text{ff} \\ \text{ff if } B[x = 1]_s = \text{tt} \end{cases}$$

$$B[\Sigma x = 1]_s = \begin{cases} \text{tt if } A[\Sigma x]_s = A[\Sigma 1]_s \\ \text{ff if } A[\Sigma x]_s \neq A[\Sigma 1]_s \end{cases}$$

Pela interpretação de expressões lógicas, tem-se: $A[\Sigma x]_s = s x = 3$ e $A[\Sigma 1]_s$

Logo, $A[\Sigma x]_s \neq A[\Sigma 1]_s \rightarrow B[\Sigma x = 1]_s = \text{ff} \rightarrow B[\neg(x = 1)]_s = \text{tt}$

Portanto, $B[\Sigma \neg(x = 1)]_s = \text{tt}$

1.3.

$$3. \text{true} [y \rightarrow a_0] = \text{tt}$$

$$\text{false} [y \rightarrow a_0] = \text{ff}$$

$$(a_1 = a_2) [y \rightarrow a_0] = (a_1 [y \rightarrow a_0]) = (a_2 [y \rightarrow a_0])$$

$$(a_1 \leq a_2) [y \rightarrow a_0] = (a_1 [y \rightarrow a_0]) \leq (a_2 [y \rightarrow a_0])$$

$$(\neg a_1) [y \rightarrow a_0] = \neg(b(a_1 [y \rightarrow a_0]))$$

$$(a_1 \wedge a_2) [y \rightarrow a_0] = (b(a_1 [y \rightarrow a_0])) \wedge (b(a_2 [y \rightarrow a_0]))$$

1.4.

4. Para simplificação, as seguintes convenções serão adotadas:

$s_0 \rightarrow$ Estado Inicial em que $x = 17, y = 5$ e não foi definido $S(a, b, c) \Rightarrow S[x \rightarrow a][y \rightarrow b][z \rightarrow c]$

$\text{Expr} \Rightarrow (z := e + f; x := x - y)$

$\langle z := 0, s_0 \rangle \rightarrow S(17, 5, 0)$

To

$\langle z := 0, \text{while } y \leq x \text{ do Expr}, s_0 \rangle \rightarrow S(2, 5, 3)$

compr ns

onde To é:

while ff
ns

$\lceil \text{while } y \leq x \text{ do Expr}, S(2, 5, 3) \rceil \rightarrow S(2, 5, 3)$

$\langle \text{Expr}, S(7, 5, 2) \rangle \rightarrow S(2, 5, 3) \dots$

$\langle \text{Expr}, S(12, 5, 1) \rangle \rightarrow S(7, 5, 2) \quad \langle \text{while } y \leq x \text{ do Expr}, S(7, 5, 2) \rangle \rightarrow S(2, 5, 3)$

$\langle \text{Expr}, S(17, 5, 0) \rangle \rightarrow S(12, 5, 1)$

$\langle \text{while } y \leq x \text{ do Expr}, S(12, 5, 1) \rangle \rightarrow S(2, 5, 3)$

while ff
ns

$\langle \text{while } y \leq x \text{ do Expr}, S(17, 5, 0) \rangle \rightarrow S(2, 5, 3)$

1.5.

1.5.1. while $\neg(x=1)$ do ($y := y * x$; $x := x - 1$)

É impossível determinar se o programa sempre termina ou sempre se prende em loop. Isso acontece pois a execução do programa depende do valor de x no estado inicial. O comando $x := 1$ irá decrementar x .

Logo, se x for menor que 1, nunca haverá um estado s' em que $x = 1$ e o programa entra em loop infinito. Se x for maior ou igual a 1, eventualmente x será igual a 1 e o programa irá terminar.

1.5.2. while $1 \leq x$ do ($y := y * x$; $x := x - 1$)

O programa sempre termina. O comando $x := x - 1$ sempre decrementa o valor de x . Portanto, se no estado inicial s' o valor de x for menor que 1, o programa já irá terminar imediatamente. Caso contrário, eventualmente x será decrementado até ficar menor que 1 e a regra 'whileff' será aplicada.

1.5.3. while true do skip

O programa sempre entra em um laço infinito. A condição do comando while é ‘true’ sempre,, e consequentemente o comando ‘skip’ é executado. Como ‘skip’ não altera o estado de execução, essa situação irá sempre ocorrer, logo, há um laço infinito.

1.6. Imagens anexasadas

6.a) $(\text{While } (b, \text{stmt})) \Rightarrow$
 if ($\text{evalB } b$) then
 let val newS = $\text{evalStmt } \text{stmt}$ in
 $\text{evalStmt } \text{stmt} \text{ newS}$
 end
 else s

$$b) \frac{\langle s, s \rangle \rightarrow s' \quad \langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s''}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s''} \text{ se } \beta[[b]]_{s'} = \text{ff} \quad \left\{ \begin{array}{l} \text{Enquanto } b \text{ for} \\ \text{falso permanece} \\ \text{"no loop"} \end{array} \right.$$

$$\frac{\langle s, s \rangle \rightarrow s'}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s'} \text{ se } \beta[[b]]_{s'} = \text{tt} \quad \left\{ \begin{array}{l} \text{enquanto } b \text{ for verdadeiro} \\ \text{"a loop finaliza."} \end{array} \right.$$

c) $(\text{Repeat } (\text{stmt}), b)) \Rightarrow$

let val new-state = $\text{evalStmt } \text{stmt}$; val b-v = $\text{evalB } b$ new-state

IN

if b_v then new-state else $\text{evalStmt } \text{stmt}$ new-state.

end

d) Caso: $b = \text{true}$

Em ambos os casos o programa
para imediatamente se que
 b é verdadeiro

Caso ii) $b = \text{false}$ e b se torna true após k iterações
Em ambos os casos, quando b for false , o comando
 S é executado e o estado se altera. Quando b vira true
os códigos nunca servem alterar o estado.

2. Binding, escopo

3.

- 3.1.1. Escopo estático : 1
- 3.1.2. Escopo dinâmico: 2
- 3.2.
 - 3.2.1. Escopo de g: bloco 1
Escopo do primeiro let: bloco 2
Escopo de f: bloco 3
Escopo de h: bloco 4
Escopo do segundo let: bloco 5
 - 3.2.2. Os nomes definidos neste programa são g, x, inc, f, y, h, z
 - 3.2.3. g, x -> bloco 1
inc (da terceira linha), f, h -> bloco 2
y -> bloco 3
z -> bloco 4
inc (da sétima linha) -> bloco 5
 - 3.2.4. O valor de g 5 é 6. Se SML possuísse escopo dinâmico, g 5 teria valor 7. Isso ocorre pois no cenário em que SML possui escopo dinâmico o valor da variável ‘inc’, definido na linha 3 (no bloco 2) é sobreescrito pelo valor de ‘inc’ definido na linha 7 (bloco 5). Neste caso, ‘inc’ passa a valer 2 e não 1 e consequentemente tem-se $5 + 2 = 7$ como resultado da chamada g 5. Considerando o escopo estático de SML, mesmo que a variável ‘inc’ seja declarada novamente na linha 7, sua definição inicial é a que compõe o escopo da função f, portanto na execução de g 5 temos $5 + 1 = 6$.
- 3.3.
 - 3.3.1. A função bad_max apresentada é ruim pois se trata de uma implementação extremamente ineficiente desse tipo de operação. Da forma descrita, a função é chamada recursivamente de modo inapropriado, gerando um código que roda em tempo quadrático no pior caso, já que cada valor na lista é comparado com todos os outros a partir dele até o fim da lista na busca pelo maior elemento. Ou seja, muitas comparações desnecessárias são feitas. Uma implementação eficiente iria fazer apenas uma quantidade linear de comparações.
 - 3.3.2.


```
fun good_max [] = 0
  | good_max (x::xs) = let val currMax = if xs = [] then x else (good_max xs
    in
    | if x > currMax then x else currMax
    end;
```
- 3.4.

```
fun expr () =
  let
    | val x = 1
    in
      ( let val x = 2 in x + 1 end ) + ( let val y = x + 2 in y + 1 end )
  end;
```

O valor de saída é 7. No primeiro escopo ($x = 2$, $x + 1$) a saída é 3. No segundo escopo ($y = x + 2$, $y + 1$) a saída é 4, pois tem-se o valor de $x = 1$ definido no escopo anterior. Assim, $4 + 3 = 7$.