Introdução a Construção de Compiladores

Parte 4 – Geração de Código

F.A. Vanini

IC – Unicamp

Klais Soluções

Organização em Tempo de Execução II

Esses exemplo apresentam os principais elementos a se considerar ao se definir a estrutura de um programa em tempo de execução:

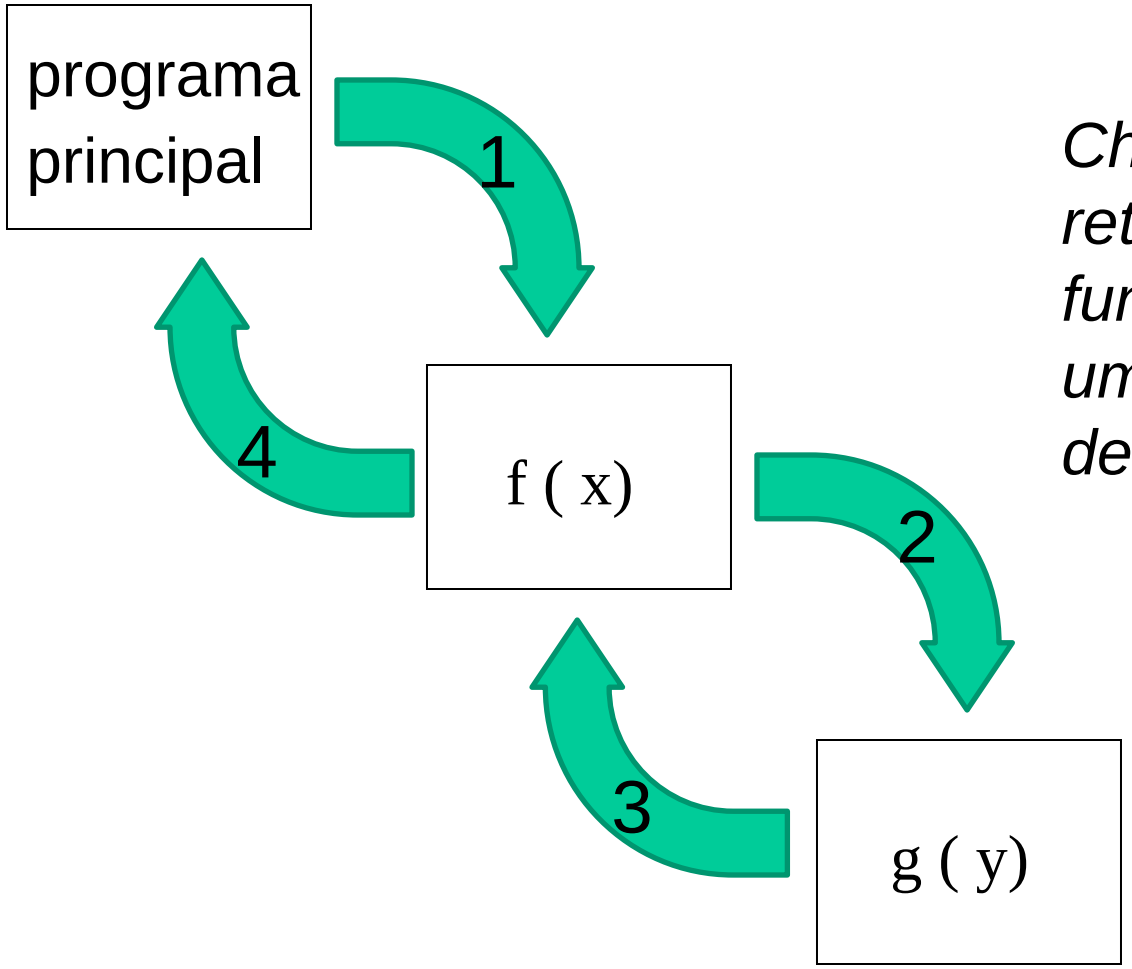
- Variáveis estáticas (globais)
- Variáveis locais
- Parâmetros
- Código de procedimento ou função
- Código do programa principal (que define o endereço de início de execução)

Registro de Ativação I

Na maioria das linguagens de programação, um procedimento ou função podem ser recursivos, o que significa que várias ativações de um procedimento ou função podem coexistir durante a execução do programa.

Cada ativação tem o seu conjunto de parâmetros e variáveis locais. Além disso, ao se encerrar a execução de uma dada ativação, o programa deve continuar a partir do ponto onde foi feita a chamada correspondente.

Registro de Ativação II



Chamada e retorno de função seguem uma disciplina de pilha.

Registro de Ativação III

- O registro de ativação deve conter:
  - Endereço de retorno
  - Parâmetros
  - Variáveis locais
- A criação e destruição dos registros de ativação segue uma disciplina de pilha.
- A posição de cada um desses elementos do registro de ativação em geral é aquela mais conveniente para o hardware utilizado.

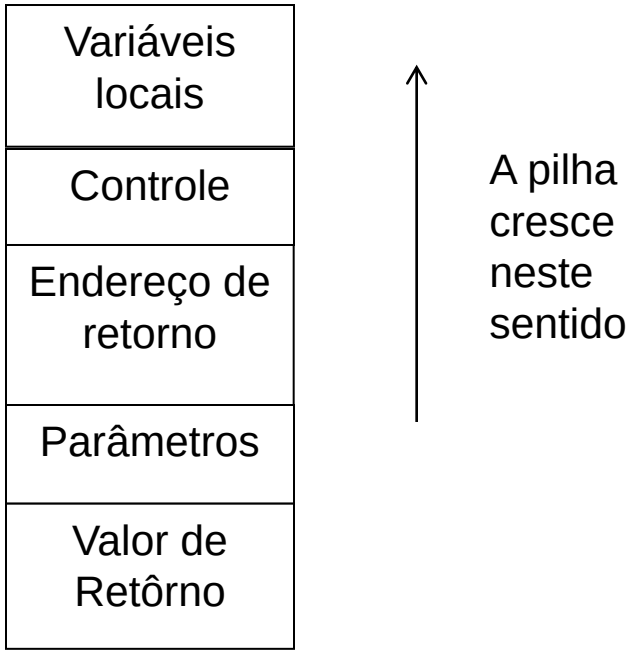
Registro de Ativação IV

Na grande maioria dos casos

- os registros de ativação são alocados na mesma pilha usada para manter resultados intermediários durante a execução do programa.
- a instrução de *chamada de funçao* empilha o endereço de retorno antes de desviar para o procedimento chamado.
- Os parâmetros são calculados pelo *chamador* e empilhados *antes da chamada*.

Registro de Ativação V

Nessas condições, o formato natural para o registro de ativação seria algo como



Chamada de Função

O código para a chamada de uma função deve:

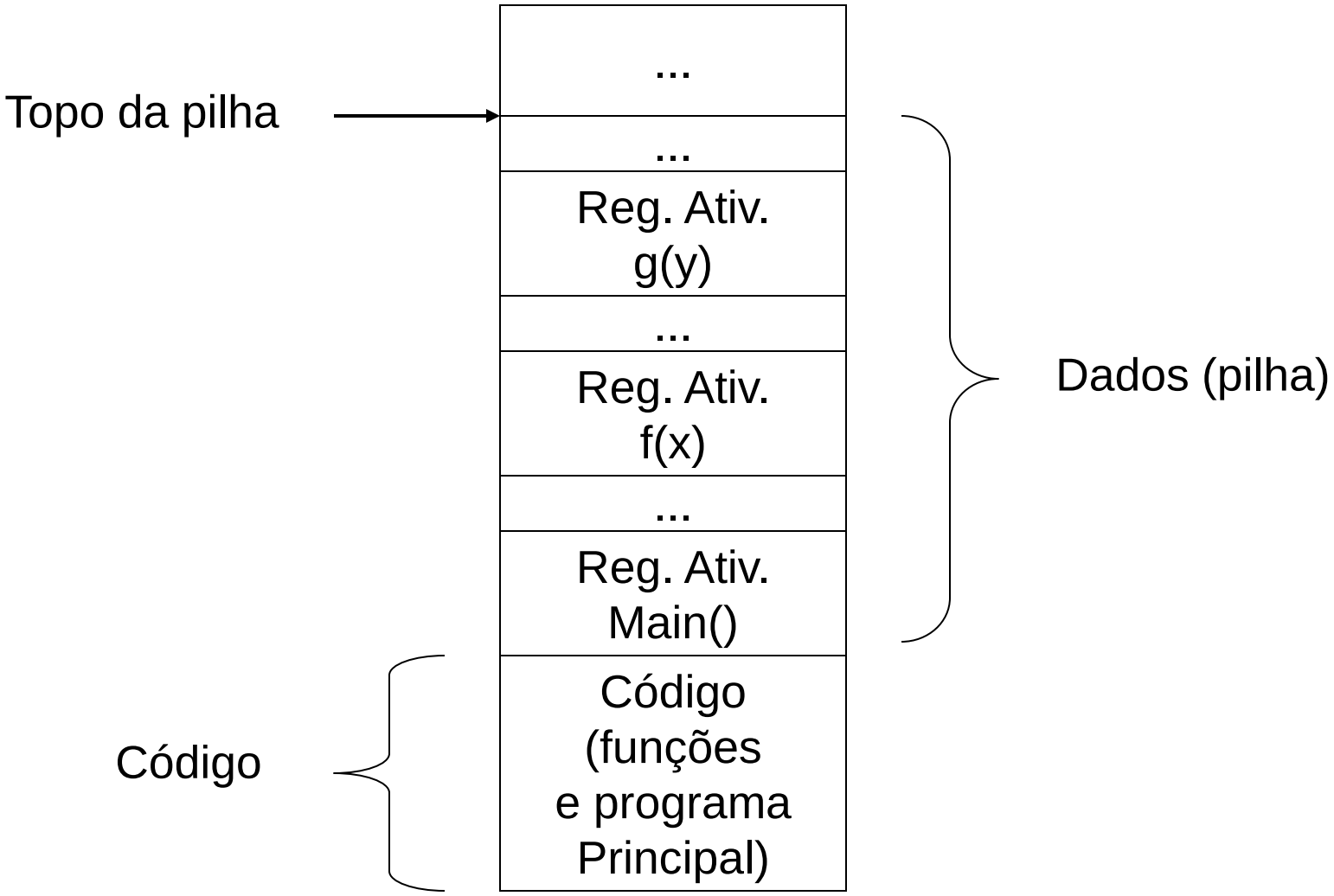
- Calcular e empilhar cada parâmetro
- Empilhar o endereço de retorno e desviar para a primeira instrução da função

O Código de uma função

Estrutura geral:

```
Início:  alocar espaço para
         variáveis locais;
          ...
          (código da função
          ...
          liberar o registro de
          ativação;
          retornar ao chamador;
```

Organização da Memória



Acesso a variáveis locais I

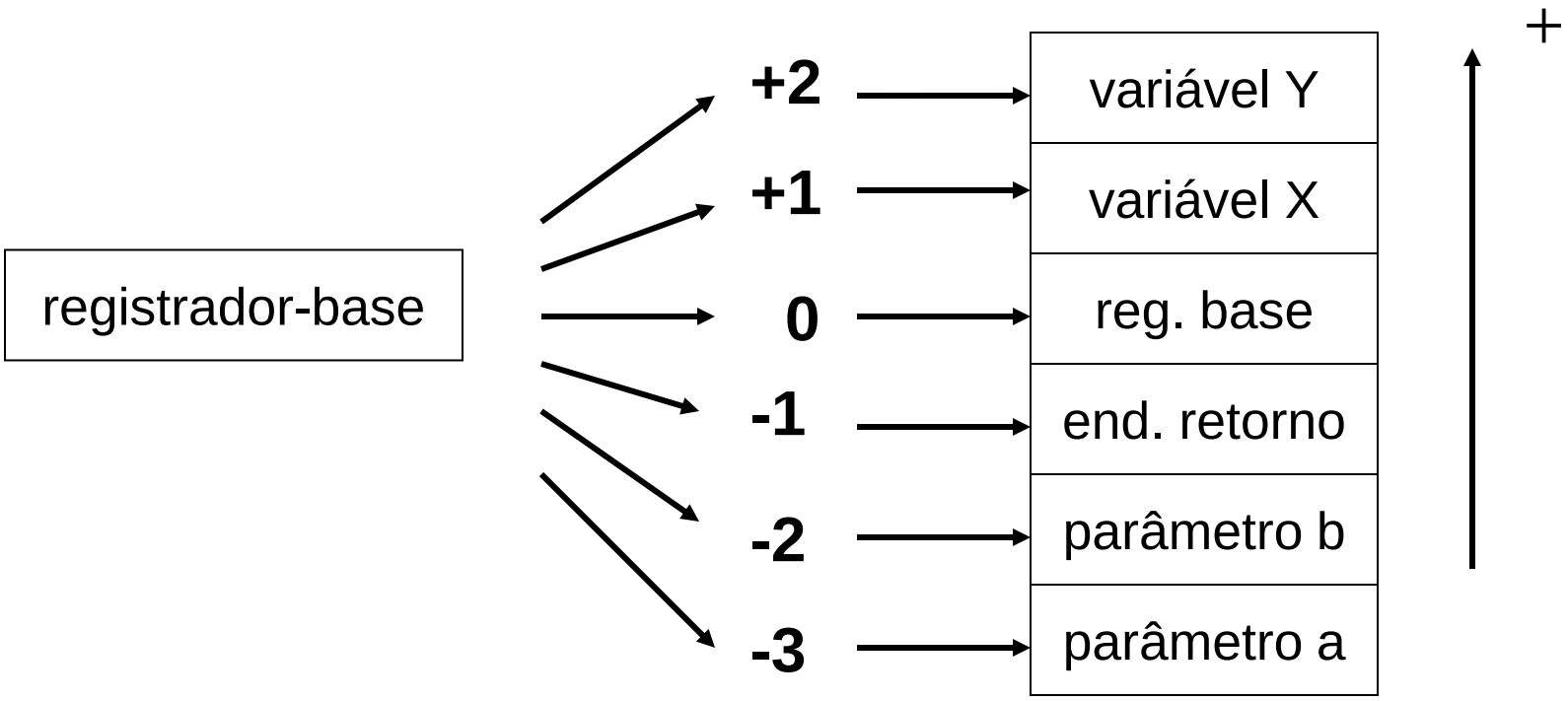
O código para as várias ativações de uma mesma função é único e cada registro de ativação tem um endereço inicial (ou endereço base) diferente.

Sendo assim, o acesso a uma variável deve independer do endereço inicial do registro de ativação.

A forma de se fazer isso é através de endereçamento relativo, disponível em qualquer máquina.

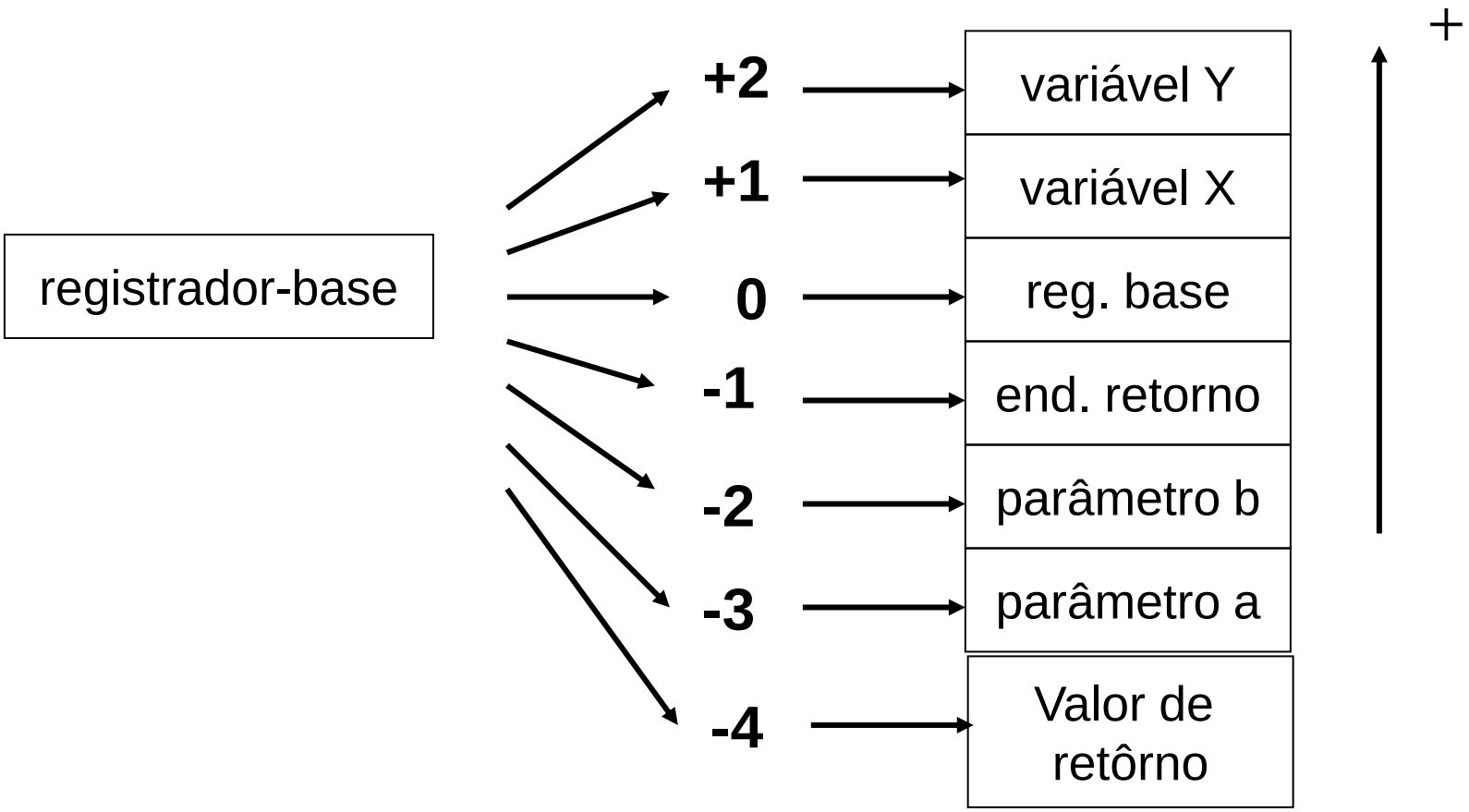
Acesso a variáveis locais II

```
void p(a,b) { int X,Y; ...  }
```



Acesso a variáveis locais III

```
int f(a,b) { int X,Y; ...  }
```



Acesso a variáveis locais III

Através do uso de um registrador-base para apontar para o início do registro da ativação, todas as ativações da função acessam as variáveis da mesma forma (através das mesmas instruções).

O que muda é o conteúdo do registrador-base, definido na entrada da função.

Acesso a variáveis locais IV

Nesse modelo, as variáveis locais são acessadas através de endereços relativos ao registrador base, com deslocamentos positivos (+1, +2, etc.) e os parâmetros com deslocamentos negativos (-2 para o último parâmetro,-3 para o penúltimo, etc.).

No caso de funções que retornam um valor, a chamada à mesma deve ser precedida da reserva de espaço para o valor de retorno. Essa reserva deve ser feita antes de empilhar os parâmetros (em muitos casos, o valor de retorno é usado numa expressão).

Variáveis Globais

As variáveis globais permanecem alocadas durante todo tempo de execução do programa e podem ser acessadas por um procedimento ou função (em Pascal ou C) .

Sendo assim, elas podem ser acessadas diretamente através do seu endereço absoluto ou através de um registrador base dedicado.

Entrada numa função

- Antes de iniciar a execução propriamente dita da função, o código gerado deve:
  - Salvar o registrador base usado para acesso às variáveis locais
  - Alocar memória para as variáveis locais

Retorno de função

- Antes de retornar ao *chamador*, o código gerado deve:
  - Liberar a memória usada para as variáveis locais
  - Restaurar o valor do registrador base alterado no início da função

A liberação da memória usada para os parâmetros pode ser feita pelo chamador.

Máquina virtual I

Para descrever o *modelo de execução* de uma linguagem podemos usar uma *máquina virtual* na qual as instruções são orientadas aos comandos e operações da linguagem.

A máquina virtual mostrada a seguir é aplicável a um subconjunto de C (ou de Pascal).

Máquina virtual II

Registradores:

- *pc (program counter): contém o endereço da próxima instrução a ser executada.*
- *top: aponta para o topo da pilha.*
- *base: registrador base, aponta para o registro de ativação da função ativa.*

Memória:

Duas áreas separadas:

- code[ ] : área de código
- stack[] : área de dados

Máquina virtual III

Instruções:

Alocação e liberação de memória

INCT n : // aloca n posições na pilha

top += n;

A mesma instrução pode ser usada para liberar n posições na pilha:

INCT –n:

top += (-n);

Máquina virtual IV

Chamada e retorno de função:

```
call f:   push(pc); // empilha endereço de retorno
          pc = f; // desvia
```

Entrada numa função:

```
enter n: push(base);
         base = top;
         top += n;
```

Retorno de uma função:

```
return n: t -= n;
          base = pop();
          pc = pop();
```

Máquina virtual V

Instruções Aritméticas:

As instruções aritméticas retiram os seus operandos da pilha e deixam na pilha o resultado da operação. Exemplos:

```
add:    push(pop() + pop());

sub:   temp = pop();
       push(pop() – temp);
```

Mult e *Div* operam de forma análoga.

Máquina virtual VI

Instruções Relacionais:

As instruções relacionais retiram os seus operandos da pilha e empilham 1 ou zero se o resultado da comparação for *verdadeiro* ou *falso.*

Gt – primeiro operando *maior que* o segundo

```
temp = pop();
push( pop()  >  temp  );
```

As instruções Lt, Le, Ge, Ne, Eq são análogas.

Máquina virtual VII

Desvio incondicional:

```
jump e: // desvio incondicional para e
        pc = e;
```

Desvio condicional:

```
jumpf  e:  // “desvie se falso para e”
           pop(temp);
           if(!pop()) pc = e;
```

Máquina virtual VIII

Acesso a variáveis:

ldvar d: // empilha o valor da variável com
// deslocamento d
push(stack[base + d]);

stvar d: //desempilha valor e armazena na
//variável com deslocamento d
stack[base+d] = pop();

Máquina virtual IX

Retorno de valor em função:

Em linguagens como C ou Java, funções são usadas nas mesmas situações em que se usam expressões.

Sendo assim, é natural, no nosso modelo, que o valor de uma função fique no topo da pilha após a chamada da mesma.

Antes da chamada de uma função, o chamador deve reservar espaço na pilha para o valor de retorno.

Máquina virtual X

Retorno de valor em função:

O espaço para o valor de retorno deve ser reservado antes do cálculo dos parâmetros.

Dentro da função, o valor de retorno é acessado com deslocamento –(k+2) onde k é o número de parâmetros.

Um exemplo I

```
int fat (int n) {
    if (n == 1) return 1;
      else return n*fat(n-1);
  }
```

```
 1  ENTER 0
 2  LDVAR -2
 3  INTCONST 1
 4  EQ
 5  JUMPF 9
 6  INTCONST 1
 7  STVAR -3
 8  RETURN 0
 9  LDVAR -2
10  INCT 1
11  LDVAR -2
12  INTCONST 1
13  SUB
14  CALL 0
15  INCT -1
16  MULT
17  STVAR -3
18  RETURN 0
```

Um exemplo II

```
void main () {
  int j,k;
  k = 5;
  j=fat(k);
}
```

```
20     ENTER 2
21     INTCONST 4
22     STVAR 2
23     INCT 1
24     LDVAR 2
25     CALL 0
26     INCT -1
27     STVAR 1
28     HALT
```

Outro exemplo I

```
int mdc(int a, int b){
if(a == b) return a;
    if(a > b)
       return mdc(a-b, b);
     return mdc(b,a);
  }
```

```
 1  ENTER 0
 2  LDVAR -3
 3  LDVAR -2
 4  EQ
 5  JUMPF 9
 6  LDVAR -3
 7  STVAR -4
 8  RETURN 0
 9  LDVAR -3
10  LDVAR -2
11  GT
12  JUMPF 22
13  INCT 1
14  LDVAR -3
15  LDVAR -2
16  SUB
17  LDVAR -2
18  CALL 0
19  INCT -2
20  STVAR -4
21  RETURN 0
22  INCT 1
23  LDVAR -2
24  LDVAR -3
25  CALL 0
26  INCT -2
27  STVAR -4
28  RETURN 0
```

Outro exemplo II

```
int main(){
  println("mdc=" + mdc(18,35);
}
```

```
30  ENTER 0
31  STRCONST "mdc ="
32  INCT 1
33  INTCONST 18
34  INTCONST 35
35  CALL 0
36  INCT -2
37  INTTOSTR
38  CONCAT
39  PRINT
40  PRINTLN
41  HALT
```

Parâmetros por referência II

A passagem de parâmetros por referência exige que uma variável seja acessada através do seu endereço absoluto.

A máquina virtual deve oferecer mecanismos para permitir esse acesso.

Durante a execução do procedimento, os parâmetros por referência devem ser tratados como contendo os endereços das variáveis que representam. Isso é explícito em C.

Parâmetros por referência I

```
void troca(ref int a, ref int b){
  int t = a;
  a = b;
  b = t;
}
```

```
 1 ENTER 1
 2 LDVAR -3
 3 LDI
 4 STVAR 1
 5 LDVAR -2
 6 LDI
 7 LDVAR -3
 8 STI
 9 LDVAR 1
10 LDVAR -2
11 STI
12 RETURN 1
```

Parâmetros por referência II

```
void main(){
    int x, int y;
    x = 99;
    y = 33;
    troca(x,y);
}
```

```
14  ENTER 2
15  INTCONST 99
16  STVAR 1
17  INTCONST 33
18  STVAR 2
31  LDADDR 1
32  LDADDR 2
33  CALL 0
34  INCT -2
35  HALT
```

Parâmetros por referência III

As instruções:

ldi: // carrega o valor da variável cujo endereço
// está no topo da pilha
push(stack[pop()]);

sti: // armazena o valor no topo da pilha na
//variável cujo endereço está no topo da pilha
stack[pop()] = pop();

ldaddr d: // empilha endereço absoluto de variável
push(base+d);

Vetores I

```
void main(){
  int v[5] = { 5, 7, 9, 3, 1 };
  vprint(v,5);
}
```

```
80 ENTER 5
81 INTCONST 5
82 INTCONST 7
83 INTCONST 9
84 INTCONST 3
85 INTCONST 1
86 LDADDR 1
87 STBLOCK 5
88 LDADDR 1
89 INTCONST 5
90 CALL 0
91 INCT -2
92 HALT
```

Vetores II

```
void vprint(ref int v[], int n){
  int i = 0;
  while(i < n){
    print(v[i]+" ");
    inc(i);
  }
  println();
}
```

```
 1  ENTER 1
 2  INTCONST 0
 3  STVAR 1
 4  LDVAR 1
 5  LDVAR -2
 6  LT
 7  JUMPF 20
 8  LDVAR -3
 9  LDVAR 1
10  ADD
11  LDI
12  INTTOSTR
13  STRCONST " "
14  CONCAT
15  PRINT
16  LDVAR 1
17  INC
18  STVAR 1
19  JUMP 4
20  PRINTLN
21  RETURN 1
```

Implementação de Vetores II

O acesso a um elemento de um vetor exige o cálculo de endereços absolutos em tempo de excução (como por exemplo em “v[i+j]:=10”).

Uma vez calculado o endereço (deixando um resultado na pilha), é necessário acessar a posição de memória correspondente.

Para isso são necessárias duas novas instruções na máquina virtual.

Outras instruções

- INTCONST, BOOLCONST, STRCONST, CHARCONST: empilha constante.
- CONCAT: desempilha dois strings e empilha a concatenação dos mesmos.
- LDBLOCK: empilha um ‘bloco’ (tupla).
- STBLOCK: retira um bloco da pilha e armazena numa área da memória.
- INTTOSTR, CHARTOSTR, BOOLTOSTR: converte o valor no topo da pilha para string.
- CHARTOINT: converte o valor no topo da pilha, do tipo Char para int.
- PRINT: desempilha string e o escreve em stdout.
- PRINTLN: nova linha em sdtout.