Implementação de subprogramas

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Conteúdo

A semântica geral das chamadas e retornos

Implementação de subprogramas “simples”

Implementação de subprogramas com variáveis locais dinâmicas na pilha

Subprogramas aninhados

Blocos

Implementação escopo dinâmico

A semântica geral das chamadas e retornos

A semântica geral das chamadas e retornos

- As operações de chamada e retorno de subprogramas são denominadas conjuntamente de **ligação de subprograma**
- A implementação de subprogramas deve ser baseada na semântica da ligação de subprogramas

A semântica geral das chamadas e retornos

- Ações associadas com as chamadas de subprogramas
  - Passagem de parâmetros
  - Alocação e vinculação das variáveis locais dinâmicas na pilha
  - Salvar o estado de execução do subprograma chamador
  - Transferência do controle para o subprograma chamado
  - Mecanismos de acesso a variáveis não locais (no caso de subprogramas aninhados)

A semântica geral das chamadas e retornos

- Ações associadas com os retornos de subprogramas
  - Retorno dos parâmetros out e inout
  - Desalocação das variáveis locais
  - Restauração do estado da execução
  - Retorno do controle ao subprograma chamador

Implementação de subprogramas “simples”

Implementação de subprogramas “simples”

- Subprogramas simples
  - Subprogramas não podem ser aninhados
  - Todas as variáveis locais são estáticas

Implementação de subprogramas “simples”

- Semântica da chamada requer as ações
  1. Salvar o estado da unidade de programa corrente
  2. Computar e passar os parâmetros
  3. Passar o endereço de retorno ao subprograma chamado
  4. Transferir o controle ao subprograma chamado

Implementação de subprogramas “simples”

- Semântica do retorno requer as ações
  1. Copiar os valores dos parâmetros formais inout (passagem por cópia) e out para os parâmetros reais
  2. Copiar o valor de retorno (no caso de função)
  3. Restaurar o estado de execução do chamador
  4. Transferir o controle de volta ao chamador

Implementação de subprogramas “simples”

- Como estas ações são distribuídas entre o subprograma chamador e o subprograma chamado?

Implementação de subprogramas “simples”

- Memória requerida para a chamada e retorno
  - Estado do chamador
  - Parâmetros
  - Endereço de retorno
  - Valor de retorno para funções
  - Valores temporários utilizados no código do subprograma

Implementação de subprogramas “simples”

- As ações do subprograma chamado podem ocorrer
  - No início da execução (**prólogo**)
  - No final da execução (**epílogo**)

Implementação de subprogramas “simples”

- Um subprograma simples consiste de duas partes de tamanhos fixos
  - Código do subprograma
  - As variáveis locais e os dados (não código, listados anteriormente)

Implementação de subprogramas “simples”

- O formato (layout) da parte não código do subprograma é chamado de **registro de ativação**
- Uma **instância do registro de ativação** é um exemplo concreto do registro de ativação
- Como as linguagens com subprogramas simples não suportam recursão só pode haver uma instância do registro de ativação de cada subprograma
- Como o registro de ativação tem tamanho fixo, pode ser alocado estaticamente

Implementação de subprogramas “simples”

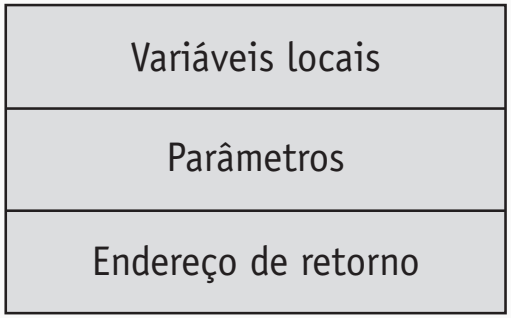


Figura 10.1 Um registro de ativação para subprogramas simples.

Implementação de subprogramas “simples”

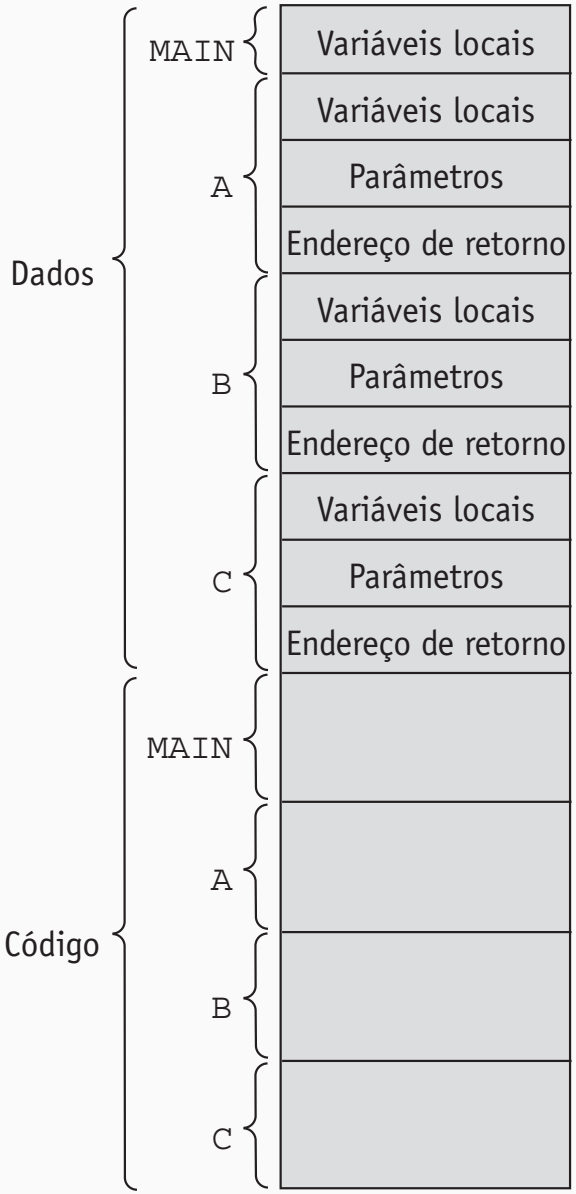


Figura 10.2 Um registro de ativação para subprogramas simples.

Implementação de subprogramas com variáveis locais dinâmicas na pilha

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- Registros de ativação mais complexos
  - O compilador precisa gerar código para alocação e desalocação das variáveis locais
  - Suporte a recursão (mais de uma instância de registro de ativação para um subprograma)

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- O formato de registro de ativação é conhecido em tempo de compilação (na maioria das linguagens)
- O tamanho do registro de ativação pode variar (se existirem arranjos dinâmicos na pilha, por exemplo)
- Os registros de ativação precisam ser criados dinamicamente

Implementação de subprogramas com variáveis locais dinâmicas na pilha

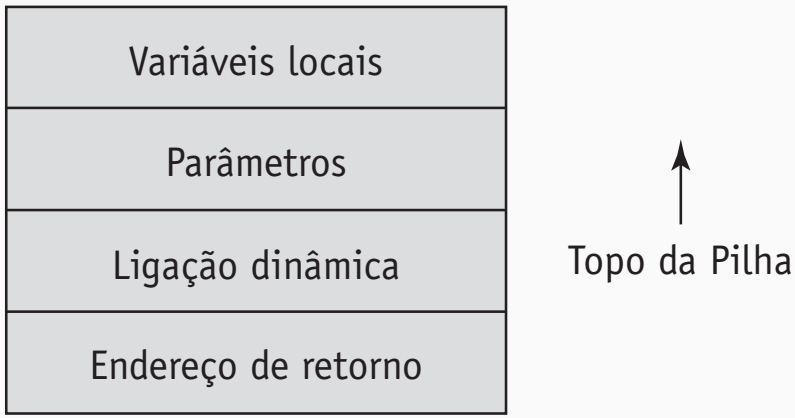


Figura 10.3 Um registro de ativação típico para uma linguagem com variáveis locais dinâmicas da pilha.

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- O endereço de retorno geralmente consiste em um ponteiro para a instrução que segue a chamada do subprograma
- O **vínculo dinâmico** aponta para a base da instância do registro de ativação do chamador
- Os parâmetros são valores ou endereços dados pelo chamador
- As variáveis locais são alocadas (e possivelmente inicializadas) no subprograma chamado

Implementação de subprogramas com variáveis locais dinâmicas na pilha

```
void sub(float total, int part){
  int list[5];
  float sum;
  ...
}
```

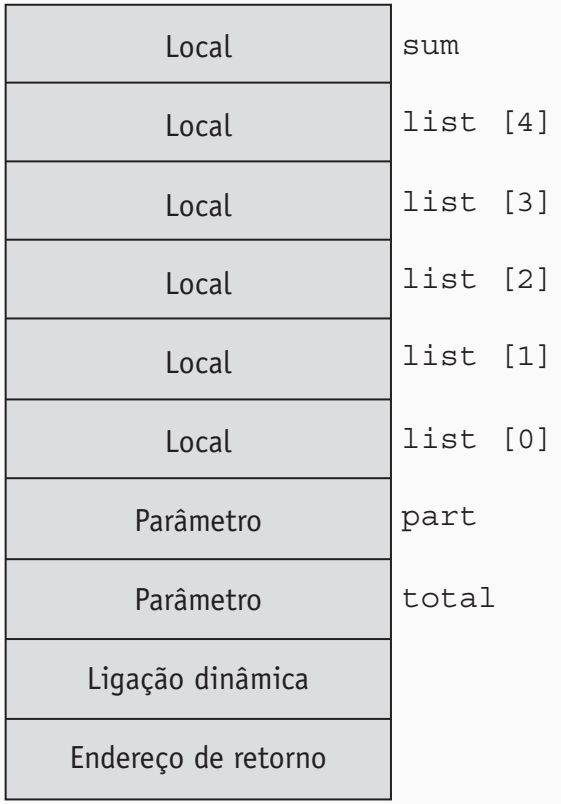


Figura 10.4 O registro de ativação para a função `sub`.

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- Ativar um subprograma requer a criação dinâmica de um registro de ativação
- O registro de ativação é criado na **pilha de execução** (run-time stack)
- O **ponteiro de ambiente** (EP - Enviromnet Pointer) aponta para a base do registro de ativação da unidade de programa que está executando

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- O EP é mantido pelo sistema
  - O EP é salvo como vínculo dinâmico do novo registro de ativação quando um subprograma é chamado
  - O EP é alterado para apontar para a base do novo registro de ativação
  - No retorno, o topo da pilha é alterado para EP - 1, e o EP é alterado para o vínculo dinâmico do registro de ativação do subprograma que está terminado
- O EP é usado como base do endereçamento por deslocamento dos dados na instância do registro de ativação

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- Ações do subprograma chamador
  1. Criar um registro de ativação
  2. Salvar o estado da unidade de programa corrente
  3. Computar e passar os parâmetros
  4. Passar o endereço de retorno ao subprograma chamado
  5. Transferir o controle ao subprograma chamado

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- Ações no prólogo do subprograma chamado
  1. Salvar o antigo EP com vínculo dinâmico, e atualizar o seu valor
  2. Alocar as variáveis locais

Implementação de subprogramas com variáveis locais dinâmicas na pilha

- Ações no epílogo do subprograma chamado
  1. Copiar os valores dos parâmetros formais inout (passagem por cópia) e out para os parâmetros reais
  2. Copiar o valor de retorno (no caso de função)
  3. Atualizar o topo da pilha e restaurar o EP para o valor do antigo vinculo dinâmico
  4. Restaurar o estado de execução do chamador
  5. Transferir o controle de volta ao chamador

Exemplo sem recursão

```
void fun1(float r) {
  int s, t;
  ...        // <- 1
  fun2(s);
}
void fun2(int x) {
  int y;
  ...        // <- 2
  fun3(y);
}
void fun3(int q) {
  ...        // <- 3
}
void main() {
  float p;
  ...
  fun1(p);
}
```

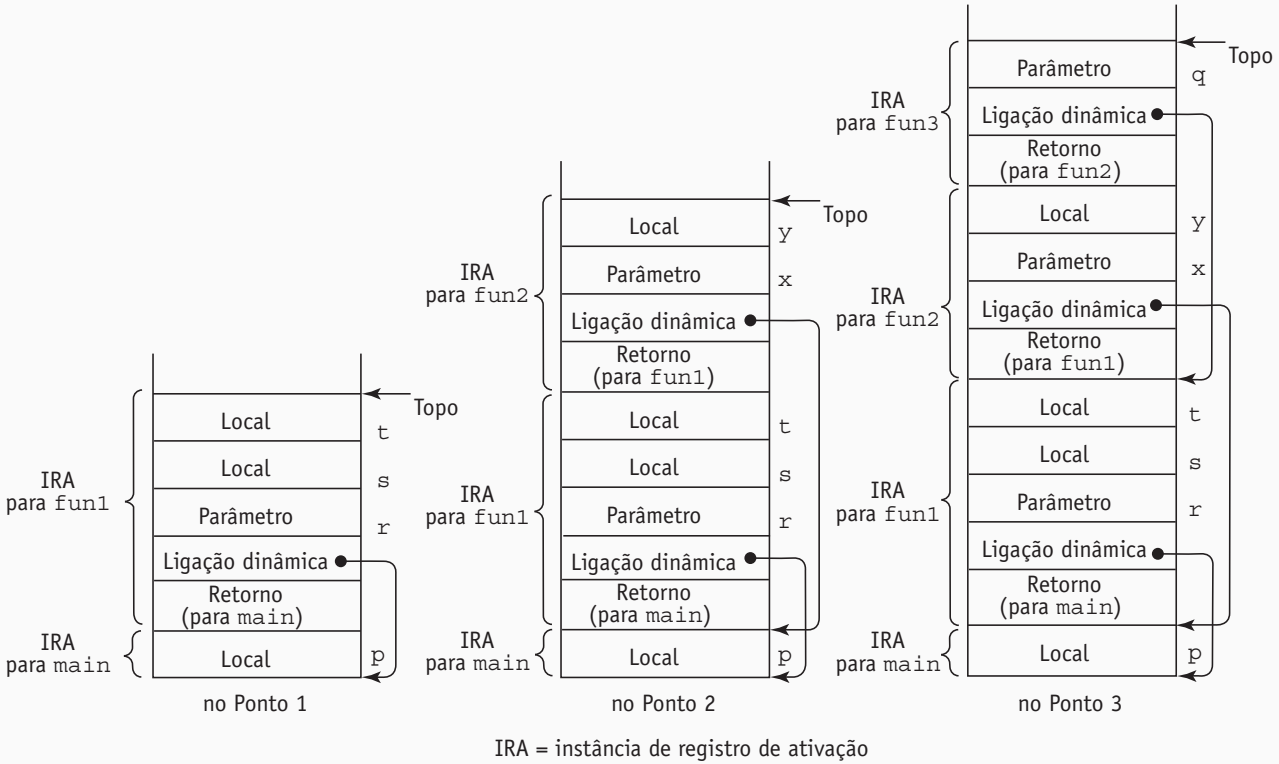


Figure 10.5 Conteúdo da pilha para três pontos em um programa.

Implementação subprogramas com variáveis locais dinâmicas na pilha

- A coleção de vínculos dinâmicos na pilha em um determinado momento é chamado de **cadeia dinâmica** ou **cadeia de chamadas**
- Referência as variáveis locais podem ser representadas por deslocamentos a partir do início do registro de ativação, cujo endereço é armazenado em EP
  - Este é o **deslocamento local** (`local_offset`)
- O deslocamento local pode ser determinado em tempo de compilação

Exemplo com recursão

```
int fat(int n) {
  // <- 1
  if (n <= 1) {
    return 1;
  } else {
    return n * fat(n - 1);
  }
  // <- 2
}
void main() {
  int value;
  value = fat(3);
  // <- 3
}
```

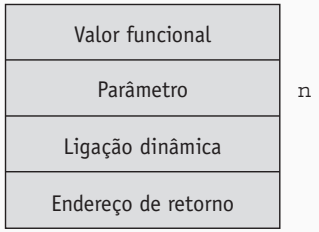


Figura 10.6 O registro de ativação para `factorial`.

Exemplo com recursão

```
int fat(int n) {
  // <- 1
  if (n <= 1) {
    return 1;
  } else {
    return n * fat(n - 1);
  }
  // <- 2
}
void main() {
  int value;
  value = fat(3);
  // <- 3
}
```

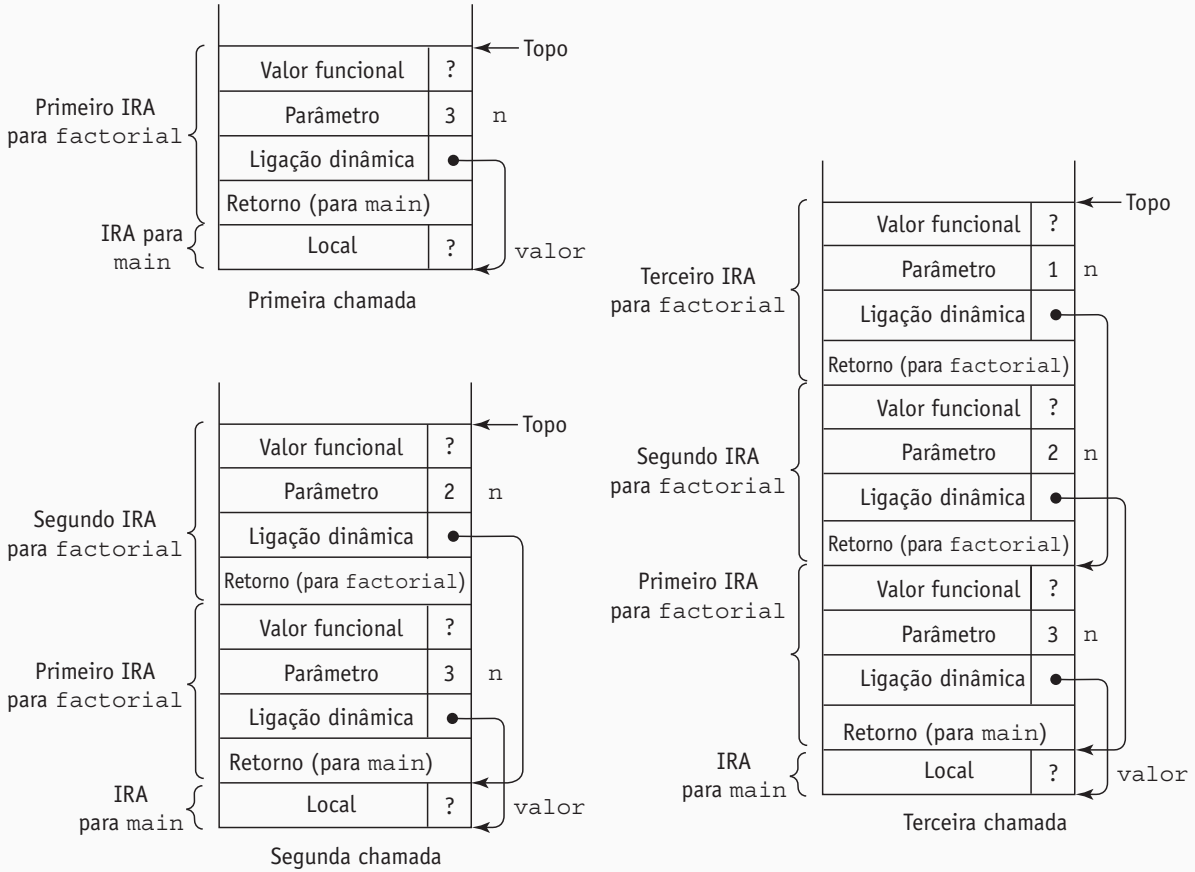


Figura 10.7 Conteúdo da pilha na posição `1` de `factorial`.

Exemplo com recursão

```
int fat(int n) {
  // <- 1
  if (n <= 1) {
    return 1;
  } else {
    return n * fat(n - 1);
  }
  // <- 2
}
void main() {
  int value;
  value = fat(3);
  // <- 3
}
```

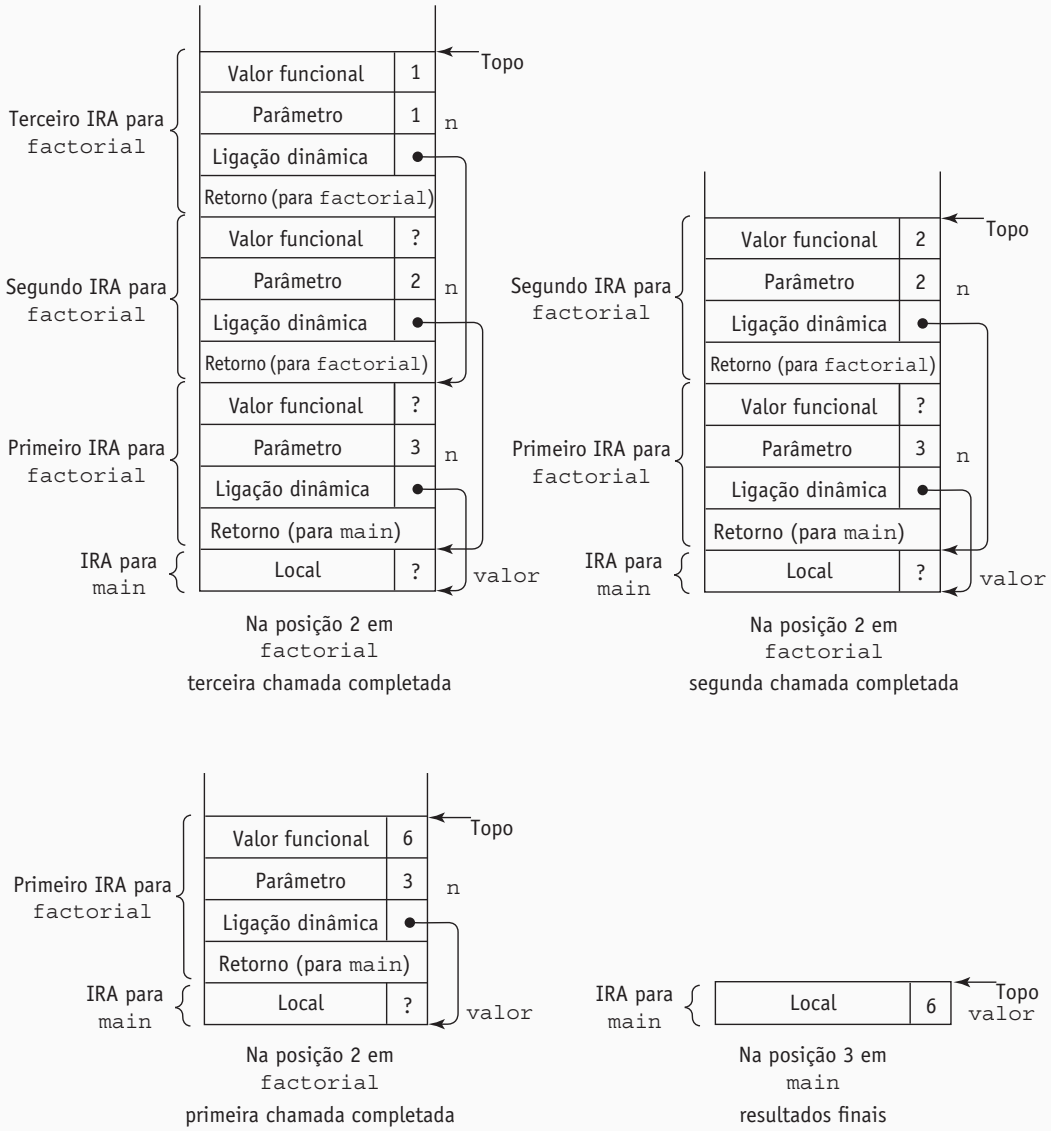


Figura 10.8 Conteúdo da pilha durante a execução de `main` e `factorial`.

Subprogramas aninhados

Subprogramas aninhados

- Algumas linguagens com escopo estático permitem subprogramas aninhados (Fortran 95, Ada, Python, Javascript, Ruby, Lua)
- Todas as variáveis não locais que podem ser acessadas estão em algum registro de ativação na pilha

Subprogramas aninhados

- Processo para encontrar um referência não local
  - Encontrar o registro de ativação correto
  - Determinar o deslocamento dentro do registro de ativação
- A forma mais comum de implementação é o encadeamento estático

Subprogramas aninhados

- Um **vínculo estático** aponta para a base de uma instância do registro de ativação de uma ativação do pai estático
- Uma **cadeia estática** é uma cadeia de vínculos estáticos que conecta certas instâncias de registros de ativação
- A cadeia estática conecta todos os ancestrais estáticos de um subprograma em execução
- A cadeia estática é usado para implementar acesso as variáveis não locais

Subprogramas aninhados

- Encontrar a instância do registro de ativação correta é direto, basta seguir os vínculos. Mas pode ser ainda mais simples
  - **`static_depth`** é um inteiro associado com o escopo estático cujo valor é a profundidade de aninhamento deste escopo
  - O **`chain_offset`** ou **`nesting_depth`** de uma referência não local é a diferença entre o `static_depth` do subprograma que faz a referência e do subprograma onde a variável referenciada foi declarada
  - Uma referência a uma variável pode ser representada por um par `(chain_offset, local_offset)`, sendo `local_offset` o deslocamento no registro de ativação da variável referenciada

Exemplo

```
procedure Main is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
        A := B + C;     -- <----- 1
      end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
          Sub1;
          E := B + A;   -- <----- 2
        end; -- of Sub3
      begin -- of Sub2
        Sub3;
        A := D + E;     -- <----- 3
      end; -- of Sub2
    begin -- of Bigsub
      Sub2(7);
    end; -- of Bigsub
  begin
    Bigsub;
  end; -- of Main
```

Qual é o valor (`chain_offset`, `local_offset`) da referência a variável `A` nos pontos 1, 2 e 3?

- Ponto 1
  - Variável local
  - `(0, 3)`
- Ponto 2
  - Variável não local (`bigsub`)
  - `(2, 3)`
- Ponto 3
  - Variável não local (`bigsub`)
  - `(1, 3)`

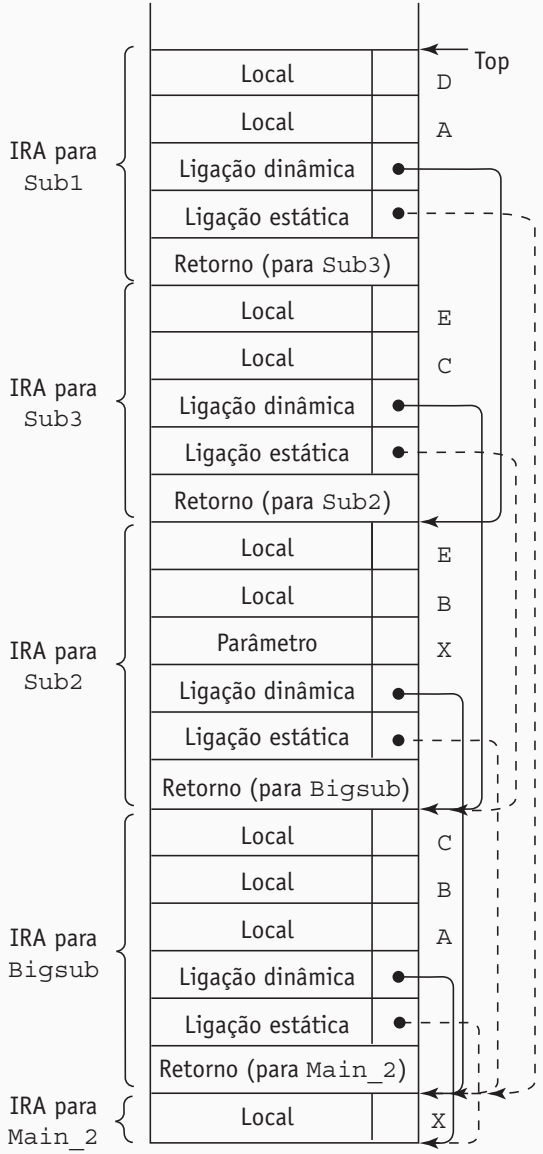


Figura 10.9 Conteúdo da pilha na posição 1 do programa.

Subprogramas aninhados

- Como a cadeia estática é mantida?
- A cadeia estática precisa ser modificada a cada chamada ou retorno de subprograma
- O retorno é trivial
- A chamada é mais complexa

Subprogramas aninhados

- Embora o escopo do pai seja facilmente determinado em tempo de compilação, a mais recente instância do registro de ativação do pai precisa ser encontrada a cada chamada
- Este processo pode ser feito com dois métodos
  - Buscar pela cadeia dinâmica
  - Tratar as declarações e referências de subprogramas como as de variáveis

Subprogramas aninhados

- Críticas ao método de encadeamento estático
  - Acesso a uma referência não local pode ser lento se a profundidade do aninhamento for grande
  - Em sistemas de tempo crítico, é difícil determinar o custo de acesso a variáveis não locais

Blocos

Blocos

- Algumas linguagens (incluindo C), permitem a criação de escopos definidos pelos usuários, chamados blocos

```
{
  int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

Blocos

- Blocos podem ser implementados de duas maneiras
  - Usando encadeamento estáticos (blocos são tratados como subprogramas sem parâmetros)
  - Como o valor máximo de memória requerido pelas variáveis de bloco podem ser determinados em tempo de compilação, esta memória poder ser alocada na pilha e as variáveis de bloco são tratadas como variáveis locais

Blocos

```
void main() {
  int x, y, z;
  while (...) {
    int a, b, c;
    ...
    while (...){
      int d, e;
      ...
    }
  }
  while (...) {
    int f, g;
    ...
  }
  ...
}
```

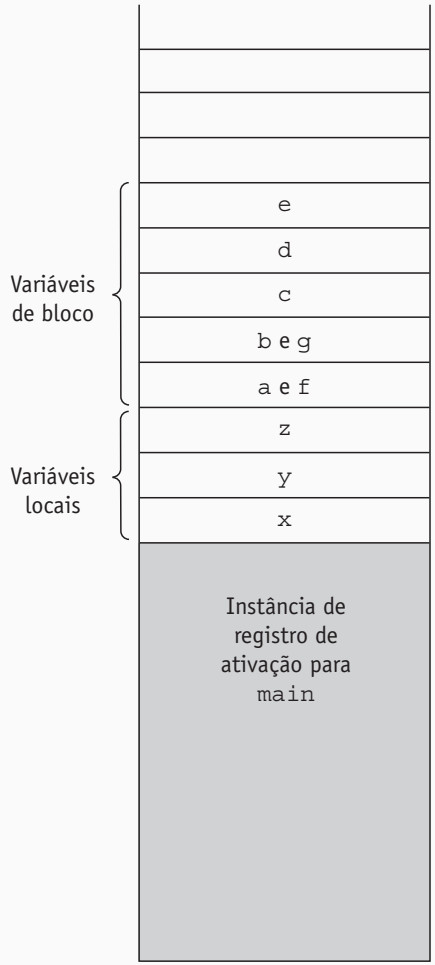


Figura 10.10 Armazenamento de variáveis de blocos quando os blocos não são tratados como procedimentos sem parâmetros.

Implementação escopo dinâmico

Implementação escopo dinâmico

- Existem duas maneiras para implementar acesso a variáveis não locais em linguagens com escopo dinâmico
  - Acesso profundo
  - Acesso raso

Implementação escopo dinâmico

- Acesso profundo
  - As referências não locais são encontradas seguindo os registros de ativação na cadeia dinâmica
  - O tamanho da cadeia não pode ser estaticamente determinada
  - Os nomes das variáveis devem ser armazenadas nos registros de ativação

Implementação escopo dinâmico

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}
void sub2() {
  int w, x;
  ...
}
void sub1() {
  int v, w;
  ...
}
void main() {
  int v, u;
  ...
}
```

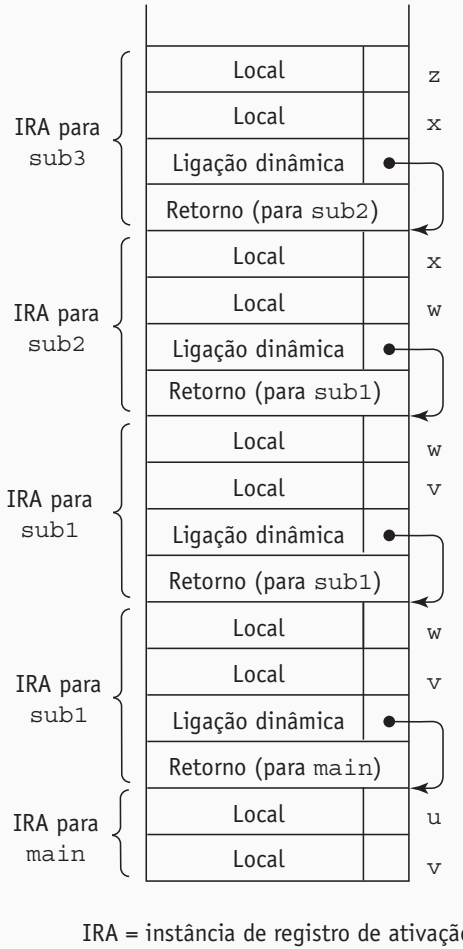


Figura 10.11 Conteúdo da pilha para um programa de escopo dinâmico.

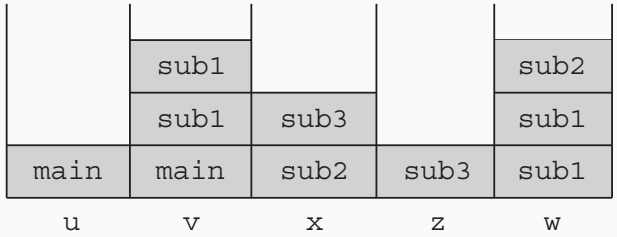
Implementação escopo dinâmico

- Acesso raso
  - As variáveis locais ficam em um lugar central (não ficam no registro de ativação)
  - Uma pilha para cada nome de variável

Implementação escopo dinâmico

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}
void sub2() {
  int w, x;
  ...
}
void sub1() {
  int v, w;
  ...
}
void main() {
  int v, u;
  ...
}
```

Chamadas: main, sub1, sub1, sub2, sub3.



(Os nomes nas células da pilha indicam as unidades de programa da declaração da variável.)

Figura 10.12 Um método de uso do acesso raso para implementar escopo dinâmico.

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 10.