Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Ambientes de Execução

- Um compilador deve implementar precisamente abstrações definidas na linguagem fonte
- Abstrações incluem nomes, escopo, bindings, tipos de dados, operadores, procedimentos…
- O compilador deve cooperar com o sistema operacional e outros sistemas de software para dar suporte a estas abstrações na máquina alvo

Ambientes de Execução

- Para realizar isto, o compilador cria um ambiente de execução no qual assume que os programas serão executados
- Este ambiente permite lidar com uma variedade de questões, como:
  - layout e alocação de posições de memória para elementos do programa
  - mecanismos usados pelo programa para acessar variáveis
  - mecanismos de passagem de parâmetros
  - interfaces com o SO, dispositivos I/O…

Organização da Memória

- O programa tem seu próprio espaço lógico de memória, onde cada valor tem seu local
- Gerenciamento e organização deste espaço é compartilhado entre compilador, SO e máquina
- O SO mapeia endereços lógicos em físicos, espalhados pela memória
- A representação de um programa neste espaço lógico consiste de áreas de dados e programa

Organização da Memória

- Em geral, assume-se que o espaço vem em blocos de *bytes* contíguos, com o *byte* sendo a menor unidade de memória endereçável
- Armazenamos objetos *multibyte* em posições consecutivas, com endereço no 1º *byte*
- A quantidade de espaço de um nome pode ser determinada a partir do seu tipo
    - objetos complexos e agregados devem ser capazes de armazenar todos os seus componentes

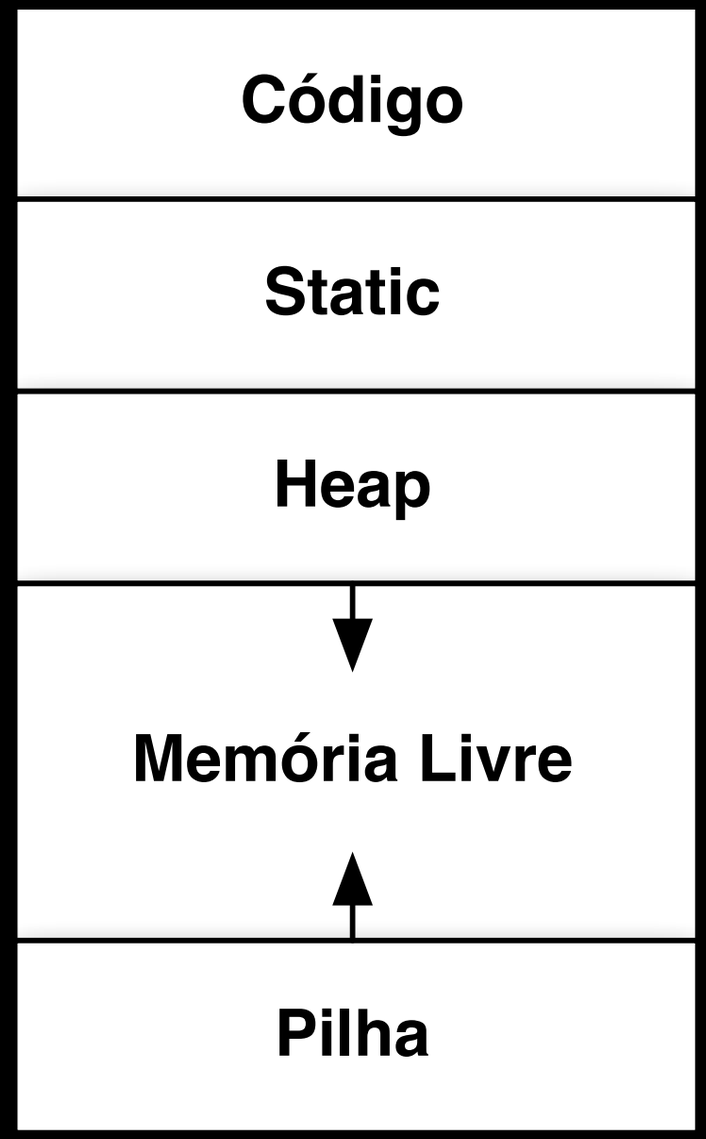
Organização da Memória

- O layout pode ser influenciado por restrições da máquina alvo
- Pode ser necessário *alinhamento*.
  - Ex.: armazenar em endereços divisíveis por 4. Embora alguns elementos precisem de 10 bytes, armazenamos em 12 (*padding*)
  - compiladores podem empacotar dados de forma a eliminar *padding*
  - isto pode gerar instruções adicionais em tempo de execução para alinhamento dos dados

Representação do programa em tempo de execução na memória

- Tamanho do código gerado é fixo em tempo de compilação, podemos colocar em uma área estaticamente determinada, geralmente em um extremo da memória

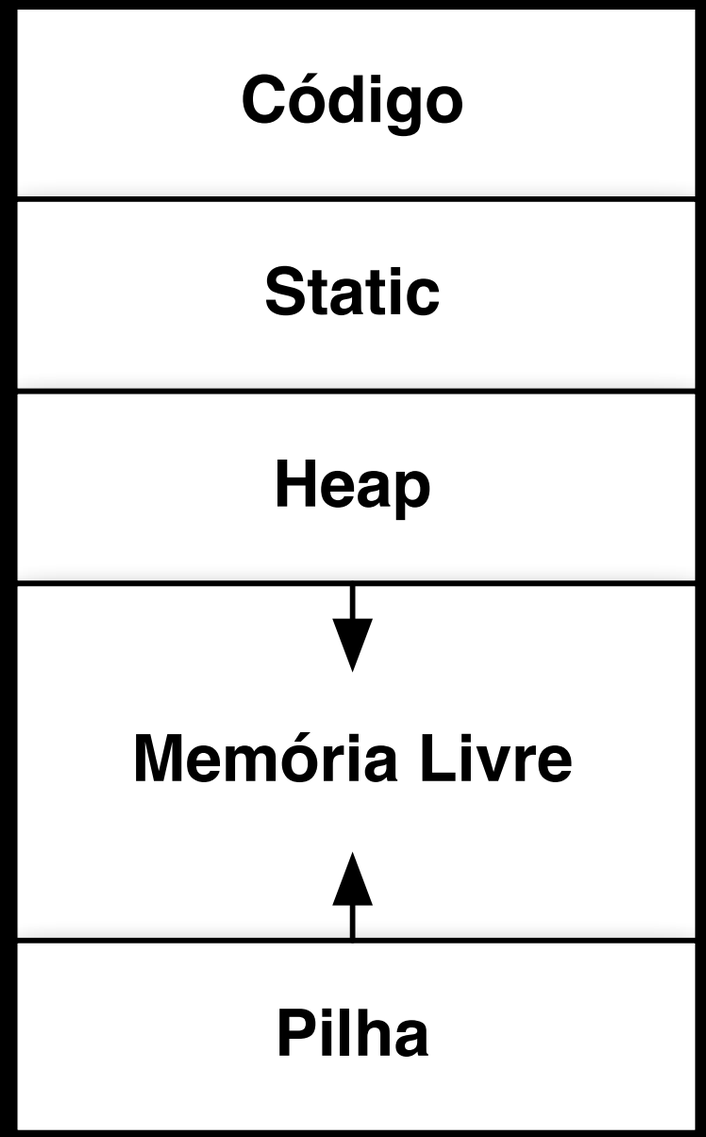
- Similarmente, o tamanho de alguns objetos de dados do programa, como constantes globais e dados gerados pelo compilador (ex.: para GC), podem também ser colocados em áreas estáticas



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Representação do programa em tempo de execução na memória

- Para maximizar uso do espaço durante execução, outras duas áreas estão em dois extremos do restante do espaço
- Mudam de tamanho dinamicamente
- A pilha é usada para guardar registros de ativação, gerados ao chamar procedimentos
- A heap é usada para dados que vivem indefinidamente, ou até que o programa explicitamente os remova



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Alocação
Estática vs. Dinâmica

- Decisões de alocação estáticas são feitas apenas com base no texto do programa fonte
- Decisões dinâmicas só podem ser tomadas durante a execução
- Muitos compiladores combinam estratégias para alocação dinâmica
  - Pilha: nomes locais a um procedimento são alocados na pilha
  - Heap: dados que podem existir após uma chamada de procedimento são alocados na heap

Alocação Estática

- Facilita operações em tempo de execução
- Nomes sempre ligados às mesmas posições
- Valores “persistem” entre chamadas diferentes a procedimentos
- Espaço de memória para variáveis pode ser alocado próximo ao procedimento ou em uma área específica

Alocação Estática - Limitações

- Tamanho dos objetos tem que ser conhecido em tempo de compilação
- Procedimentos recursivos em geral não são permitidos
- Estruturas de dados não podem ser criadas dinamicamente

Stack Allocation

- Compiladores de linguagens que usam procedimentos, funções ou métodos como unidades de modularização gerenciam ao menos parte da memória runtime em uma pilha
- Ao chamar um procedimento, espaço para as variáveis locais é alocado na pilha e ao término da execução, o espaço é liberado

Stack Allocation

- Este arranjo permite compartilhar espaço entre chamadas a procedimentos que não são sobrepostas
- permite compilar código para procedimentos de tal forma que o endereço relativo seja sempre o mesmo

Procedimentos

- Alocação na pilha não seria possível se as chamadas a procedimentos (ativações) não fossem aninhadas apropriadamente

- Fluxo de controle é sequencial (em linguagens imperativas/OO).

- A execução de um procedimento começa no início do corpo do procedimento e ao final retorna o controle ao ponto imediatamente depois do ponto em que o procedimento foi chamado.

```
int a[11];
void readArray() {…}
int partition (int m, int n) {…}

void qs (int m, int n) {
 int i;
 if (n > m) {
   i := partition(m,n);
   qs(m,i-1);
   qs(i+1,n);
  }
}

main() {readArray(); … qs(1,9);}
```

```
entrou main()
  entrou readArray()
  saiu readArray()
  entrou qs(1,9)
    entrou partition(1,9)
    saiu partition(1,9)
    entrou qs(1,3)
    …
    saiu qs(1,3)
    entrou qs(5,9)
    …
    saiu qs(5,9)
  saiu qs(1,9)
saiu main()
```

Ativação

- Ativação de um procedimento = execução de um procedimento
- Tempo de vida de uma ativação – sequência de passos do início ao fim do corpo de um procedimento *p*, incluindo a execução de procedimentos chamados por *p*
- **se *a* e *b* são ativações de procedimentos, seus tempos de vida ou não se sobrepõem ou são aninhados**

Ativação

- Se a ativação de um procedimento ***p*** chama procedimento ***q***, a ativação de ***q*** deve terminar antes que a ativação de ***p***
- Podem ocorrer três situações comuns, quais seriam?

Ativação

- Ativação de ***q*** termina normalmente
  - controle volta para o ponto de ***p*** onde ***q*** foi chamado

Ativação

- Ativação de ***q*** termina normalmente
  - controle volta para o ponto de ***p*** onde ***q*** foi chamado

- Ativação de ***q***, ou de algum procedimento chamado por ***q***, aborta, direta ou indiretamente.
  - ***p*** encerra simultaneamente com ***q***

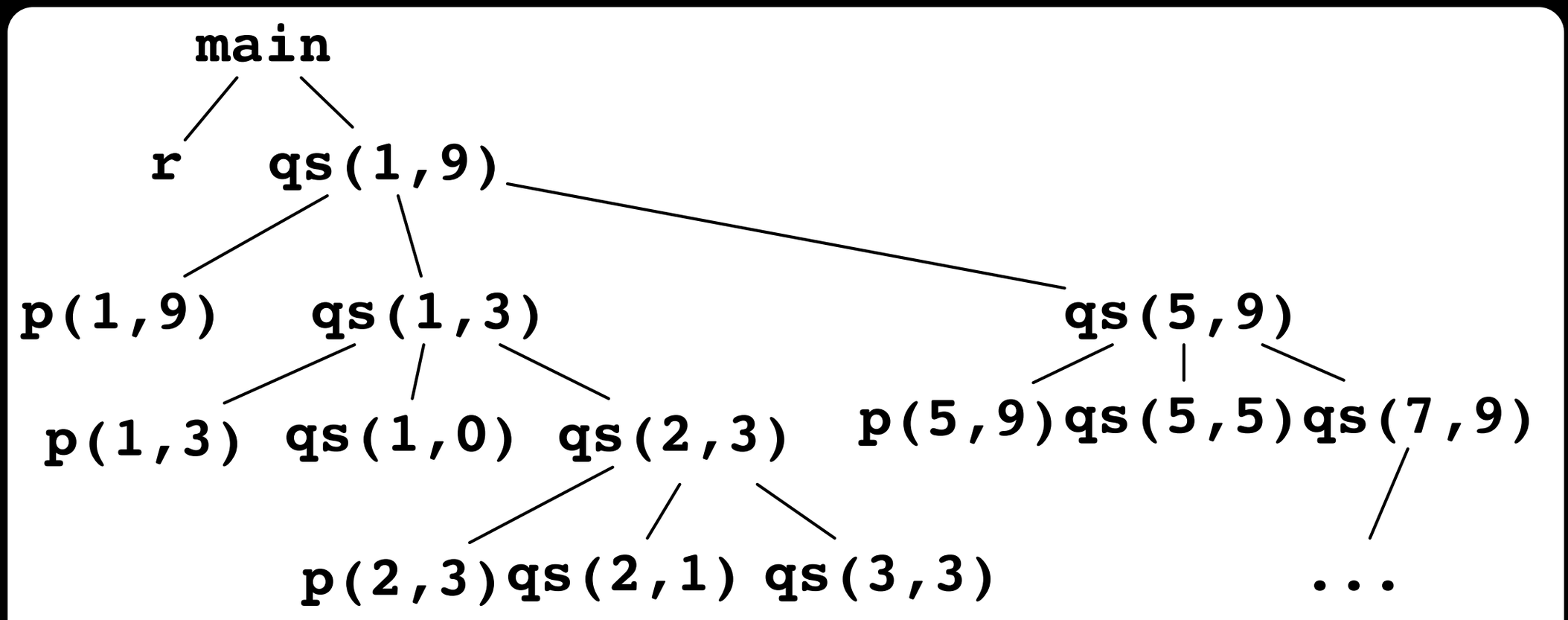
Ativação

- Ativação de ***q*** termina por conta de uma exceção que ***q*** não consegue tratar
  - procedimento ***p*** pode tratar a exceção, neste caso a ativação de ***q*** termina, enquanto a ativação de ***p*** continua, *não necessariamente do ponto onde* ***q*** *foi chamada*
  - se ***p*** não consegue tratar a exceção, a ativação de ***p*** termina ao mesmo tempo que a de ***q***, e presumidamente, a exceção será tratada por outro procedimento

Árvores de Ativação

- Podemos representar as ativações de procedimento feitas durante a execução de um programa com uma árvore
- Cada nó corresponde a uma ativação
- Os filhos de um nó ***p*** são ativações de procedimento feitas durante ativação de ***p***
- Ativações são ordenadas da esquerda pra direita, na ordem que foram chamadas

Árvore de Ativação



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

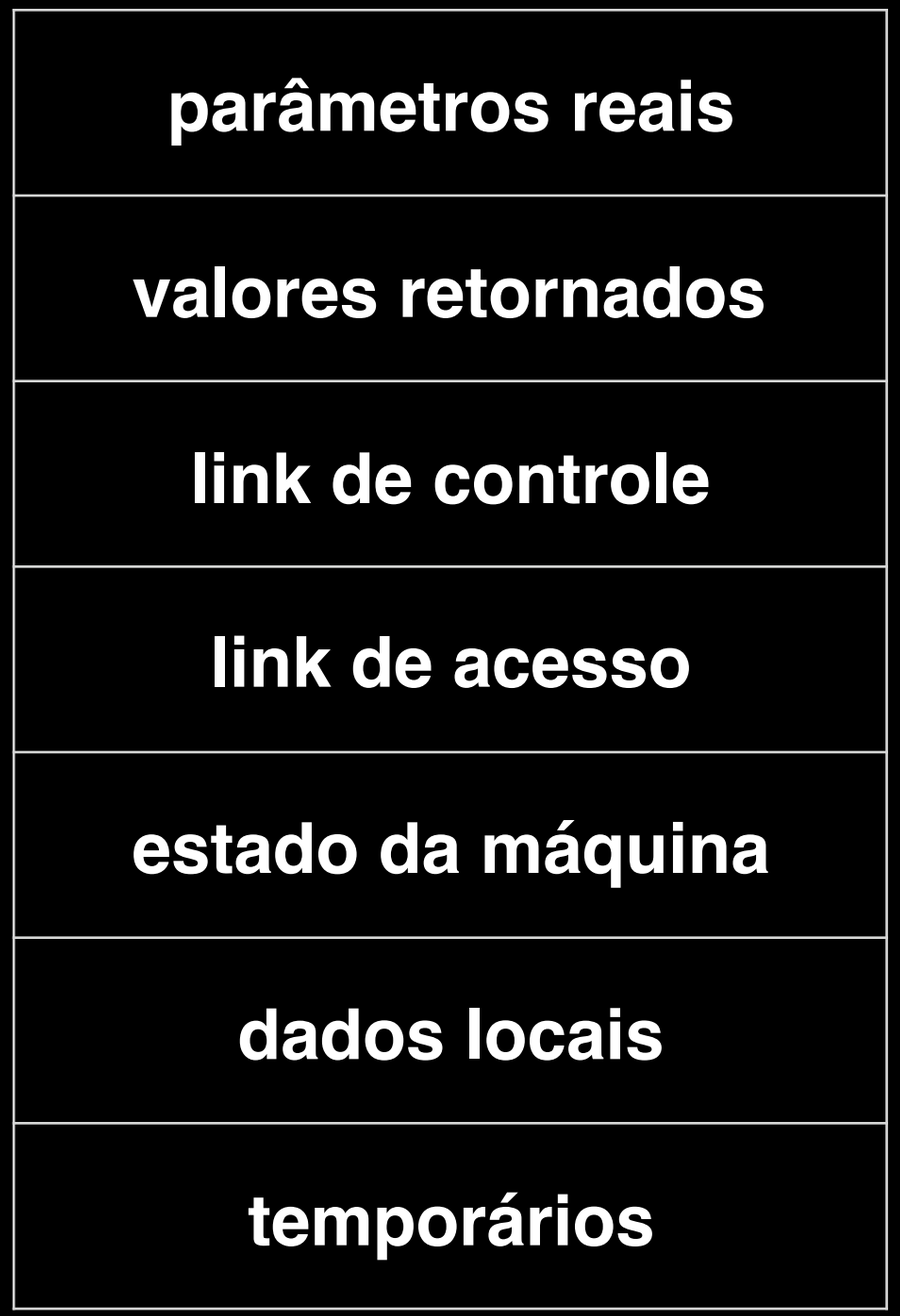
Pilha de Controle

- Fluxo de controle do programa consiste na travessia em profundidade da árvore de ativação.
- Podemos usar uma pilha de controle para acompanhar as ativações“vivas”.
- Empilha um nó no inicio da ativação e desempilha quando ela termina.
- A pilha indica o caminho até a raiz na árvore de ativação.

Registros de Ativação

- Cada ativação viva tem um registro de ativação (*frame*) na pilha de controle
  - com raiz da árvore de ativação no fundo
- A sequência de registros de ativação corresponde ao caminho percorrido na árvore de ativação, onde o controle se encontra
- Última ativação reside no topo da pilha
  - geralmente pilha é desenhada '*ao contrário*'

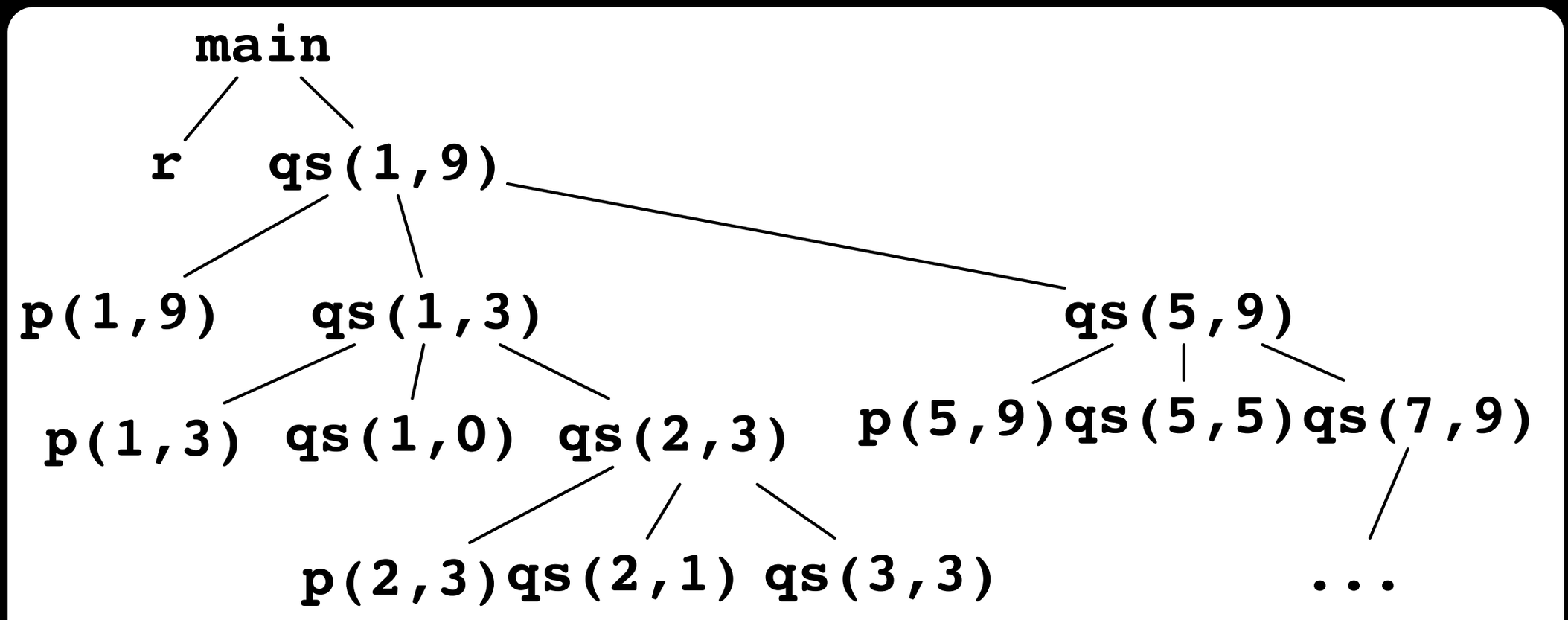
Registros de Ativação



Elementos

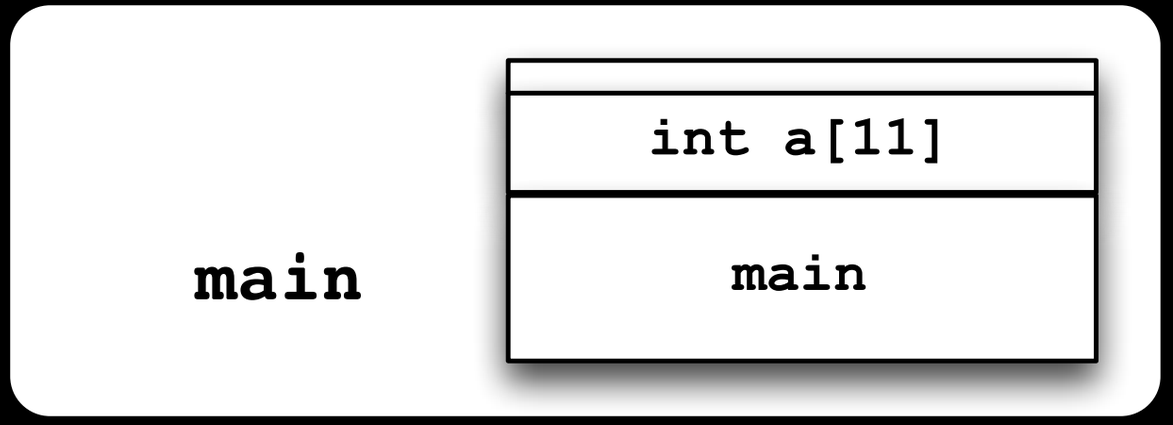
- valores temporários, resultantes de avaliação de expressões, etc.
- dados locais pertencentes ao procedimento ativo
- estado da máquina logo antes da chamada ao procedimento, endereço de retorno do contador de programas, por ex. conteúdo de registradores que será restaurado
- link de acesso para dados localizados em outros registros de ativação
- link de controle, registro de ativação de quem chamou procedimento
- valor de retorno, se houver, se possível usar registradores
- parâmetros reais, se possível, usar registradores

Exemplo

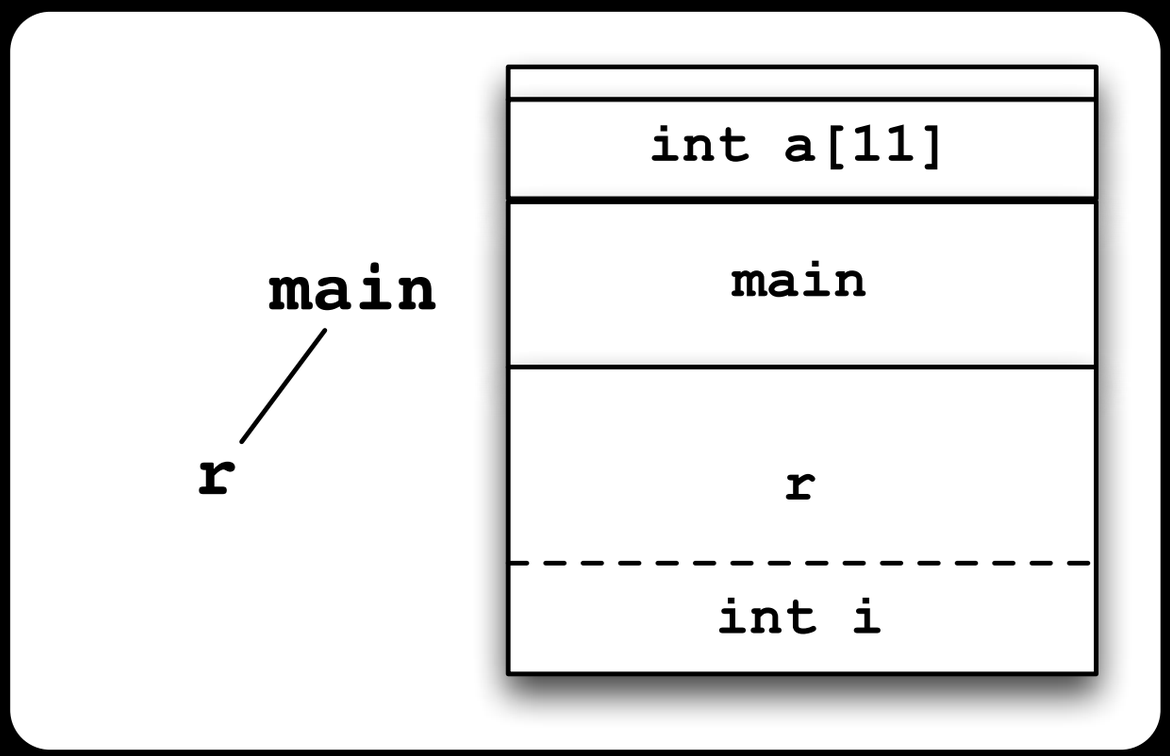


Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Exemplo

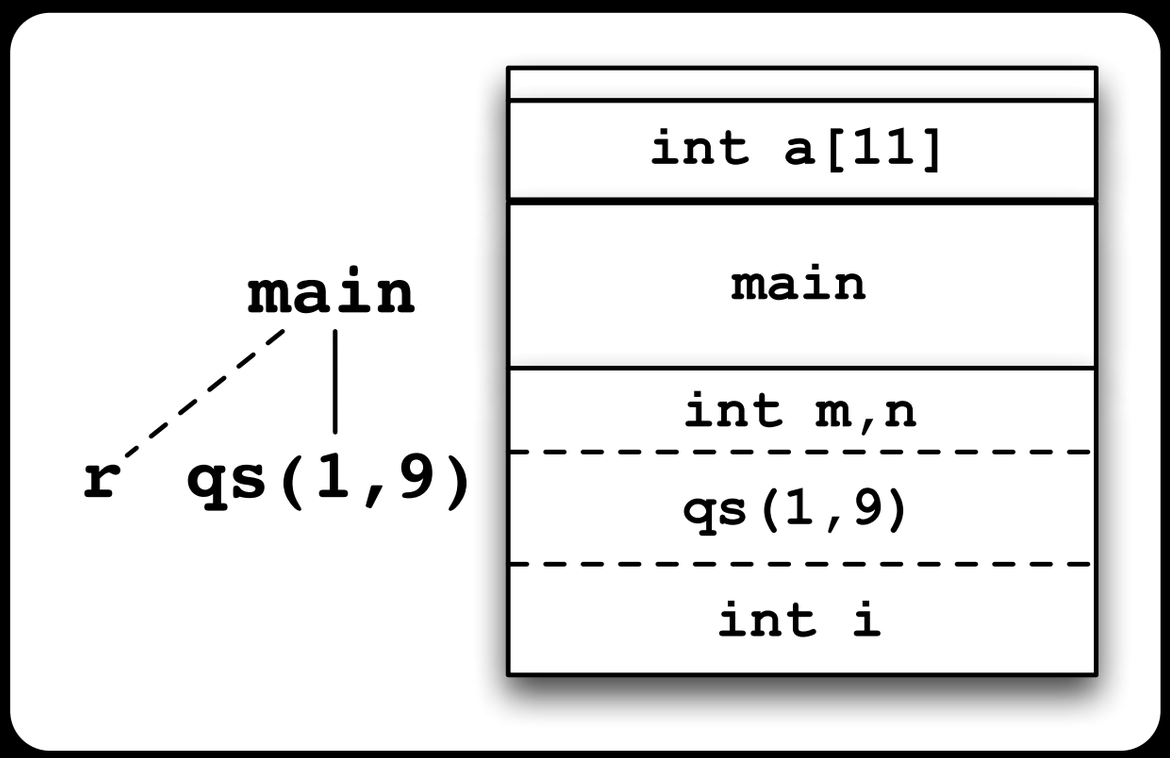


Exemplo



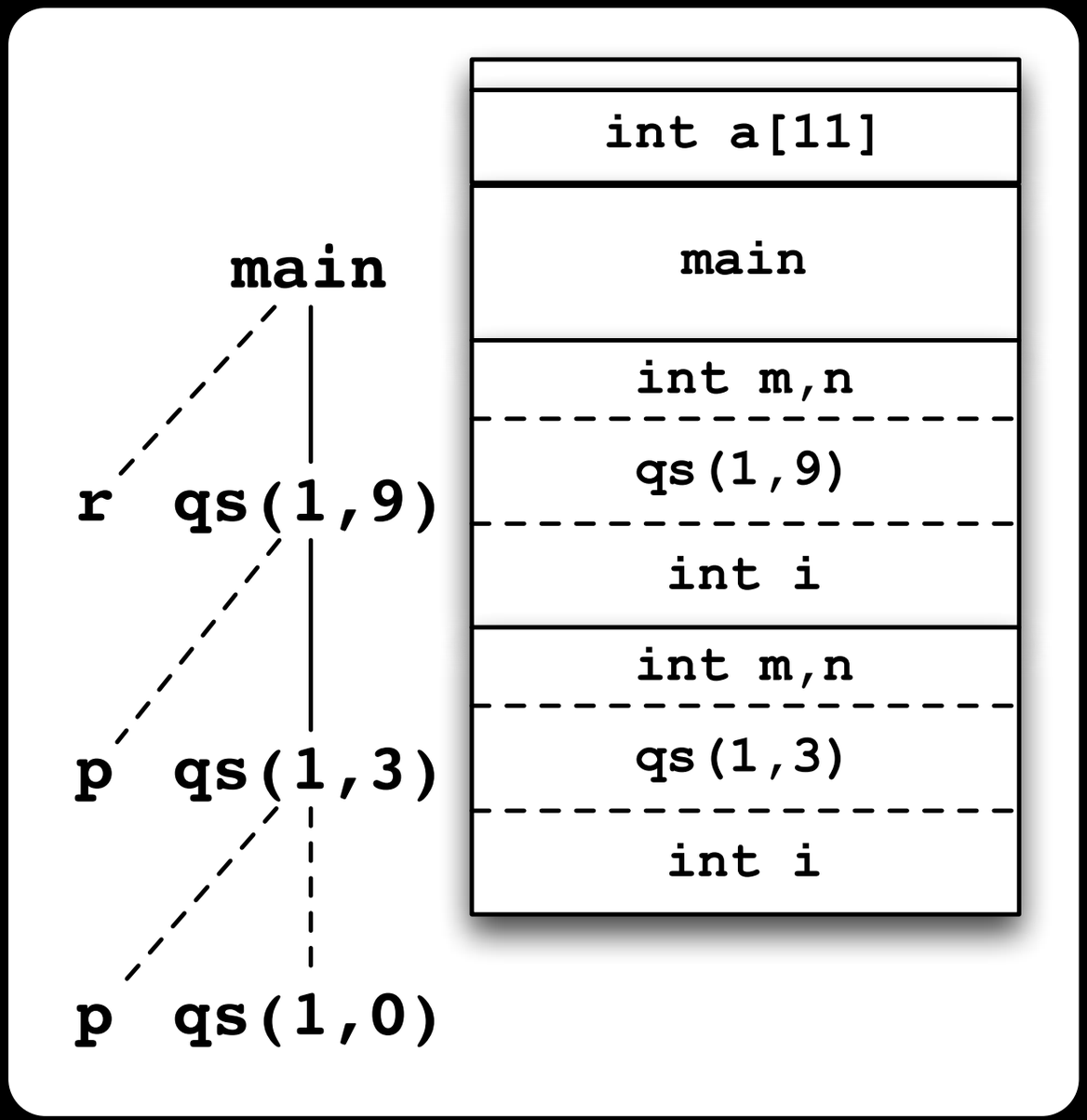
Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Exemplo



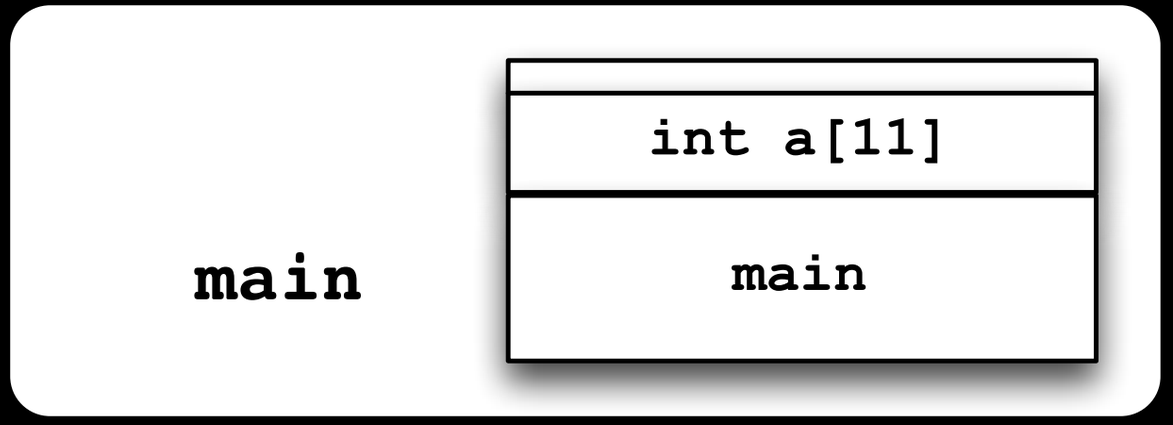
Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Exemplo



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Exemplo



Sequências

- Chamadas de procedimentos são implementadas por sequências de chamadas
  - código que aloca registro de ativação na pilha e preenche informações nos campos
- Sequências de Retorno funcionam de maneira similar
  - restauram estado da máquina, de forma que procedimento chamado continue sua execução após a chamada

Variações

- Sequências de chamada e o layout de registros de ativação podem variar, mesmo entre implementações da mesma linguagem

- Código da sequência é geralmente dividido entre *caller* e *callee*

- Não há uma divisão exata de quais tarefas são realizadas pelo *caller* vs. *callee*

Variações

- Em geral, se um procedimento é chamado de *n* pontos diferentes, a sequência de chamada assinalada ao *caller* é gerada *n* vezes
- No entanto, a porção assinalada ao *callee* é gerada apenas uma vez
- Portanto, qual seria uma boa estratégia?

Princípios

- Valores comunicados entre *caller* e *callee* são geralmente colocados no início do registro de ativação do callee
  - localizados para estarem o mais próximo possível do registro de ativação de *caller*
- Motivação é que o *caller* pode computar os valores dos parâmetros atuais e colocá-los no topo do seu registro de ativação, sem precisar criar ou mesmo saber layout do registro de *callee*

Princípios

- Itens de tamanho fixo são geralmente colocados até o meio do registro, como link de controle, link de acesso e estado da máquina

- Se sempre os mesmos componentes do estado da máquina são salvos, o mesmo código pode ser utilizado para salvar e restaurar

- Além disso, pode facilitar o trabalho de programas depuradores de erro

Princípios

- Itens de tamanho desconhecido são colocados no fim dos registros de ativação
- Maioria das variáveis locais tem tamanho fixo, mas algumas só tem tamanho conhecido durante execução
    - array de tamanho dinâmico, de acordo com um dos parâmetros da chamada

Princípios

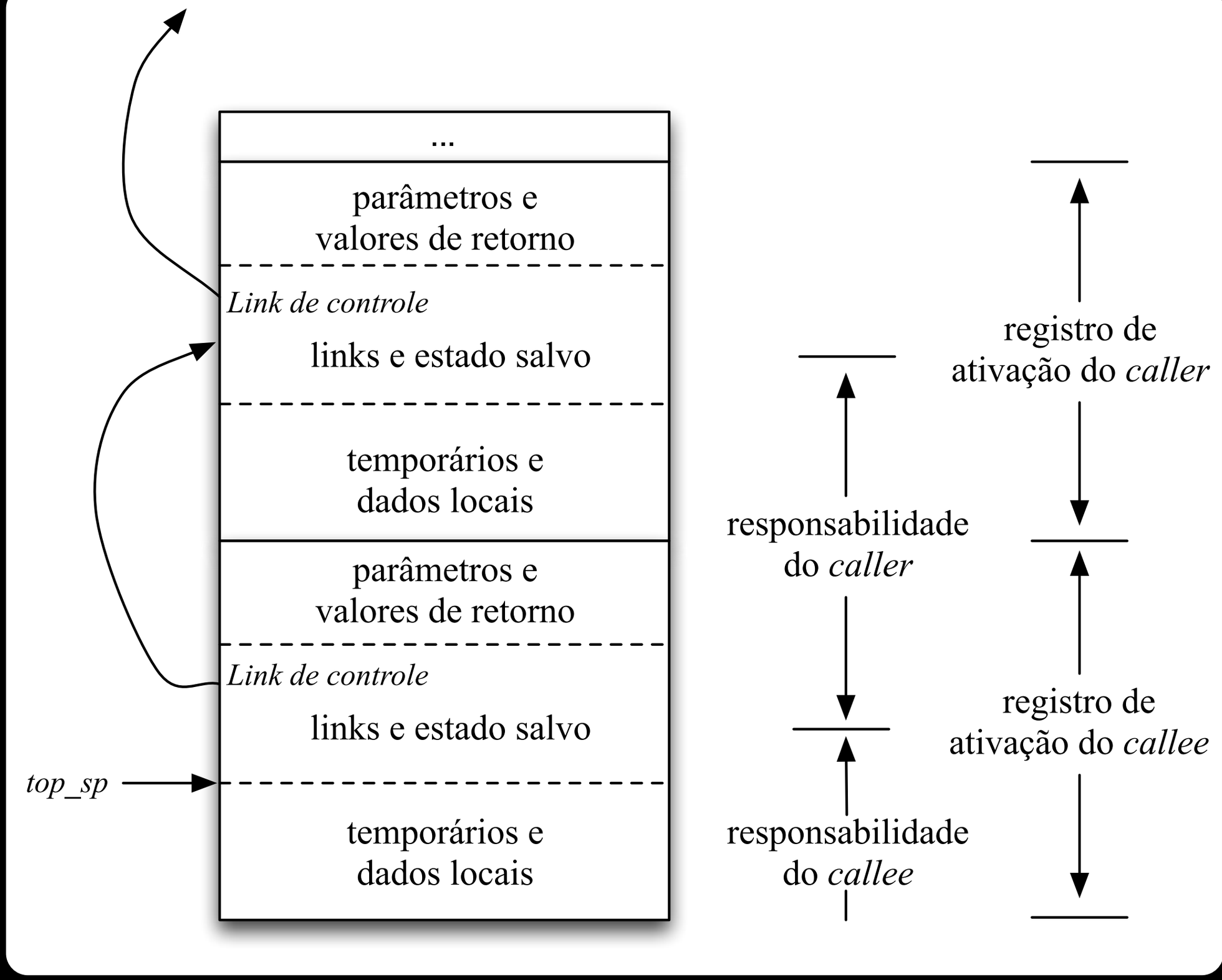
- Onde colocar o ponteiro de topo da pilha?
- Uma opção comum é colocar no fim dos campos de tamanho fixo no registro de ativação
- Desta forma, podemos acessar dados de tamanho fixo usando *offsets* fixos
- Para acessar campos de tamanho variável, precisamos calcular o *offset* em tempo de execução, pois estarão ‘acima’ do topo da pilha

Divisão de tarefas - sequência de chamada

- *caller* avalia parâmetros
- *caller* guarda endereço de retorno e valor de *top_sp*, e atualiza *top_sp*
- *callee* guarda registradores e outras informações de status
- *callee* inicializa dados locais e começa execução

Divisão de tarefas - sequência de retorno

- *callee* guarda valor de retorno próximo ao registro de ativação do *caller*
- usando informações do campo de status o *callee* restaura *top_sp* e outros registradores e vai para o endereço de retorno do *caller*.
- *caller* copia valor retornado, se necessário.



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Dados de tamanho variável na pilha

- Frequentemente o ambiente precisa alocar espaço para objetos de tamanho desconhecido
  - que são locais e portanto seriam armazenados na pilha
- Em linguagens mais modernas, estes objetos são armazenados na heap, mas também é possível armazenar na pilha
- Por qual razão desejaríamos armazenar na pilha?

Dados de tamanho variável na pilha

- Assumindo que em um procedimento há três arrays locais, cujo tamanho não pode ser determinado em tempo de compilação

- O registro de ativação portanto contém ponteiros para estes três arrays na área de campos de tamanho fixo

- Desta forma, o código pode acessar os elementos por meio destes ponteiros, localizados a um *offset* fixo do topo da pilha

Acessando dados externos

- Como encontrar e acessar dados usados em um procedimento p que não pertencem ao seu registro de ativação?
- O acesso torna-se complicado em linguagens onde procedimentos podem ser declarados dentro de outros procedimentos

Acesso em linguagens sem procedimentos aninhados

- Em linguagens como as da família C, todas as variáveis são definidas em uma função ou são variáveis globais

- Não é permitido declarar um procedimento dentro de outro

- Variáveis declaradas em uma função, tem o escopo delimitado por esta função

Alocação em linguagens sem procedimentos aninhados

- Variáveis globais são alocadas estaticamente
  - localização permanece fixa e é conhecida em tempo de compilação
    - para acessar qualquer variável que não é local ao procedimento ativo, usamos o endereço conhecido
- Qualquer outro nome deve ser local à ativação no topo da pilha
  - para acessá-los, usamos o ponteiro de topo da pilha

Questões com procedimentos aninhados

- Acesso pode se tornar mais complicado ao permitirmos declarações aninhadas
- Embora em tempo de compilação saibamos que p está aninhado dentro de q, isto não permite saber as posições relativas de seus registros de ativação em tempo de execução
  - se p ou q forem recursivos, podem ter vários

Questões com procedimentos aninhados

- Podemos saber estaticamente que um procedimento p acessa um valor não-local x
- x pode estar declarado no escopo de q, o procedimento que contém p
- No entanto, só podemos saber em tempo de execução a ativação de q correspondente à ativação de p
- Utilizamos links de acesso para estas informações