

Compiladores



JEAN LUCA BEZ

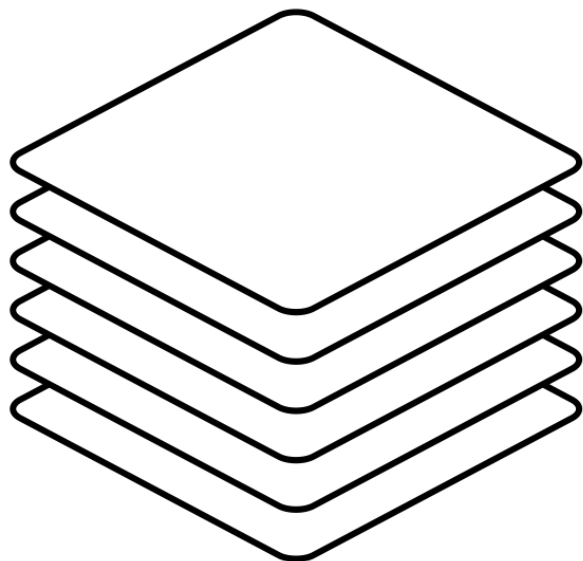
LUCAS MELLO SCHNORR

Ambiente de Execução



INF01147

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

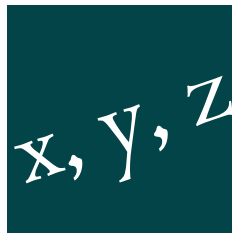


- Fases do **front-end**
 - Análise léxica
 - *Parsing*
 - Análise semântica
- Fases do **back-end**
 - Otimização
 - Geração de código

Garantir a definição da linguagem

Sistema alvo (execução)

- Linguagens possuem uma série de abstrações:

A dark blue square containing the text 'x, y, z' in a white, serif font, representing variable names.

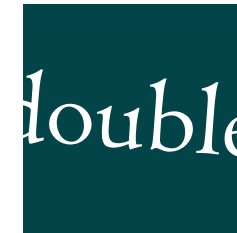
NOMES

A dark blue square containing white curly braces '{' and '}' with dotted lines connecting them, representing code scopes.

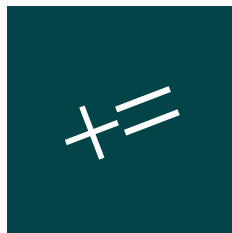
ESCOPOS

A dark blue square containing the text 'x9FFF' in a white, serif font, representing a memory address or association.

ASSOCIAÇÕES

A dark blue square containing the text 'double' in a white, serif font, representing a data type.

TIPOS DE DADOS

A dark blue square containing the text '+=' in a white, serif font, representing an operator.

OPERADORES

A dark blue square containing the text 'go()' and a closing curly brace '}', representing a procedure or function call.

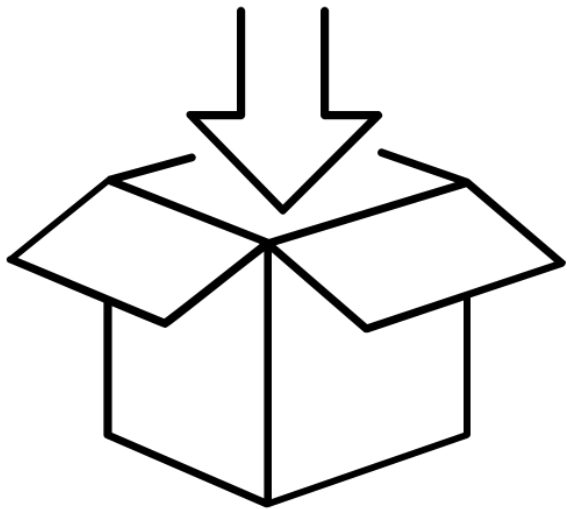
PROCEDIMENTOS

A dark blue square containing the text '(int x)' in a white, serif font, representing a parameter declaration.

PARÂMETROS

A dark blue square containing the text 'if (x)' and 'else' with curly braces, representing control flow statements.

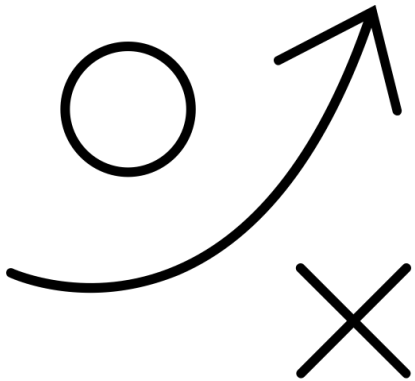
FLUXO DE CONTROLE



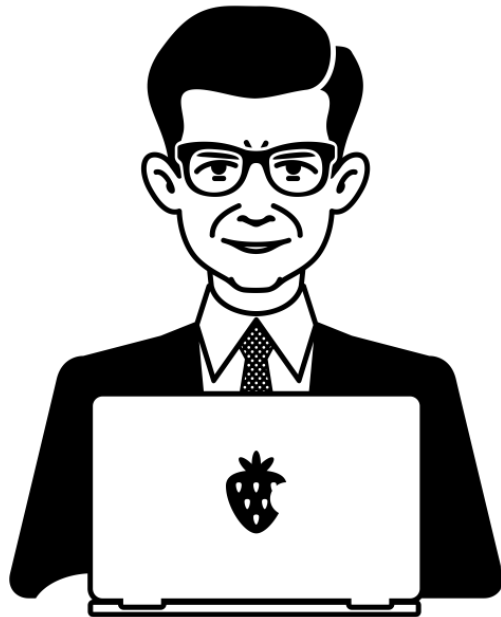
- Implementar com precisão as abstrações existentes na linguagem fonte
- Relacionar o código-fonte com as ações em tempo de execução
- Nomes (variáveis) podem denotar objetos diferentes na máquina alvo
- Alocação e liberação de dados são gerenciados:

Pacote de suporte em tempo de execução

- Rotinas carregadas com o código-alvo gerado

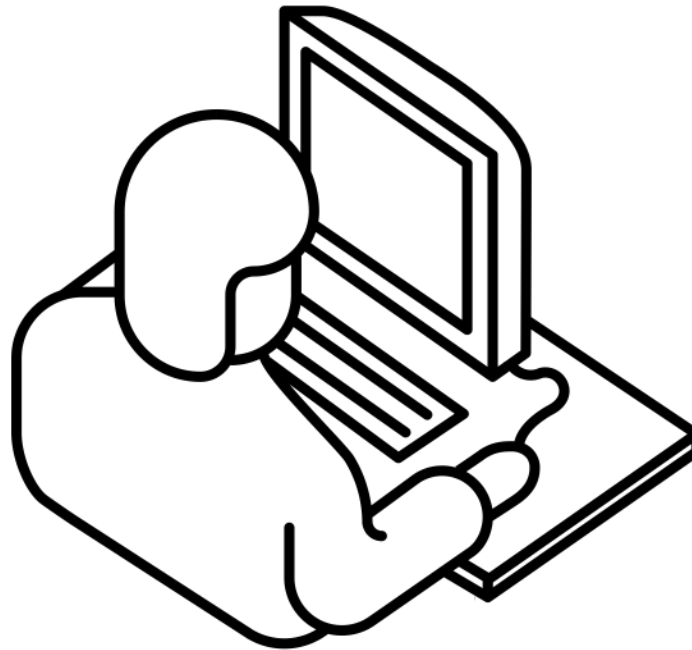


- Compilador cria e gerencia um **ambiente em tempo de execução**
- Assume que os seus programas executam neste ambiente
- Layout e alocação de endereços de memória
- Mecanismos para o acesso de variáveis
- Ligações entre procedimentos
- Mecanismos de passagem de parâmetro
- Interfaces para o SO
- Dispositivos I/O

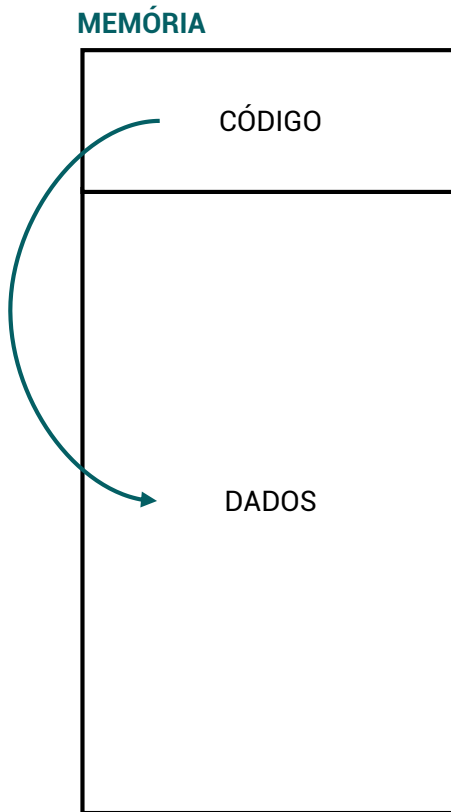


- Execução de um programa está inicialmente sobre controle do SO
- Quando um programa é **executado**:
 - O sistema operacional aloca espaço para o programa
 - O código é carregado para parte deste espaço
 - O sistema operacional pula para o ponto de entrada (e.g. `main`)

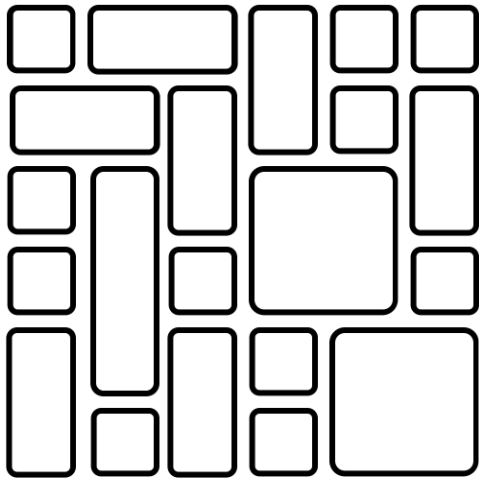
“Cada programa executa em seu próprio espaço de endereçamento lógico, no qual cada valor do programa possui um endereço.”



- Gerenciamento e organização
 - Compilador
 - Sistema Operacional
 - Máquina alvo



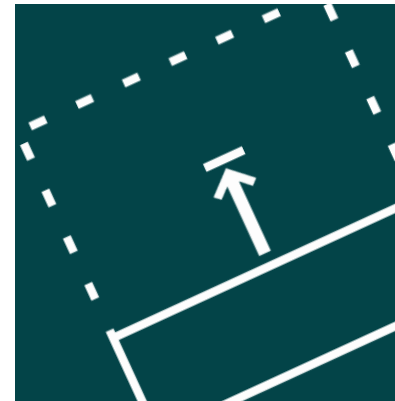
- SO mapeia endereços lógicos em endereços físicos
- Representação em tempo de execução de um programa objeto
- Espaço de endereçamento lógico
 - Área de **código**
 - Área de **dados**
- Compilador é responsável por:
 - Gerar código
 - Organizar o uso da área de dados



- Layout para os objetos de dados é **fortemente influenciado** pelas restrições de endereçamento da máquina alvo
- Uso de *padding* para alinhar áreas de memória
 - **Exemplo:** em operações com inteiros endereços deve ser divisível por 4
 - **Exemplo:** *array* com 10 caracteres, alocar 12 bytes (2 bytes sem uso)
- Compactação para salvar espaço (removendo espaços vazios)
- Instruções adicionais (durante a execução) para descompactar e operar normalmente



ESTÁTICA
TEMPO DE COMPILAÇÃO

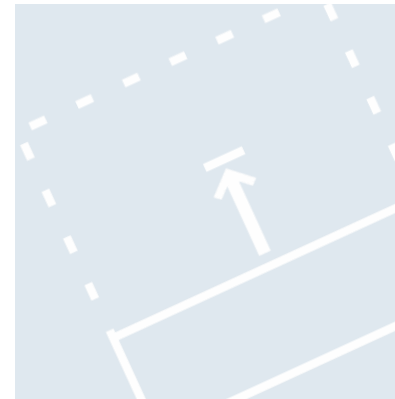


DINÂMICA
TEMPO DE EXECUÇÃO

Porque é melhor
alocar estaticamente
o máximo de objetos
de dados possível?



ESTÁTICA
TEMPO DE COMPILAÇÃO



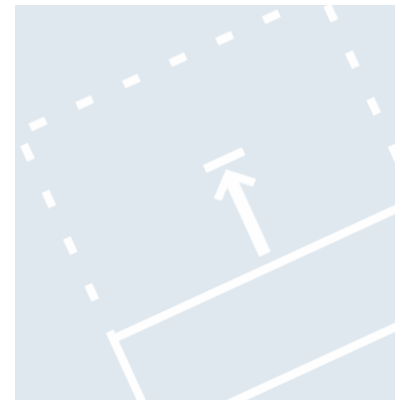
DINÂMICA
TEMPO DE EXECUÇÃO

Porque é melhor
alocar estaticamente
o máximo de objetos
de dados possível?

R. Endereços podem
ser compilados para
o código objeto.

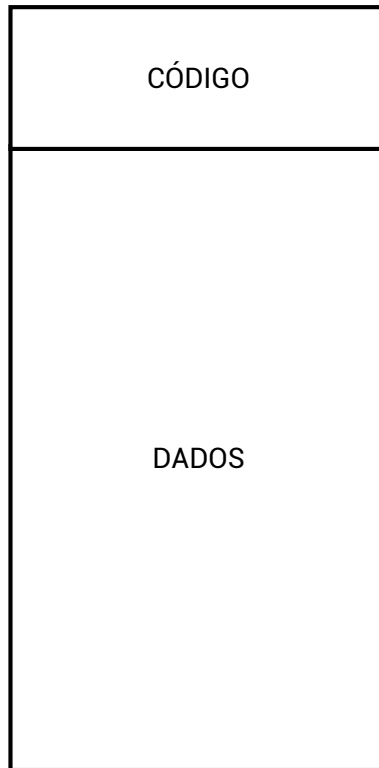


ESTÁTICA
TEMPO DE COMPILAÇÃO

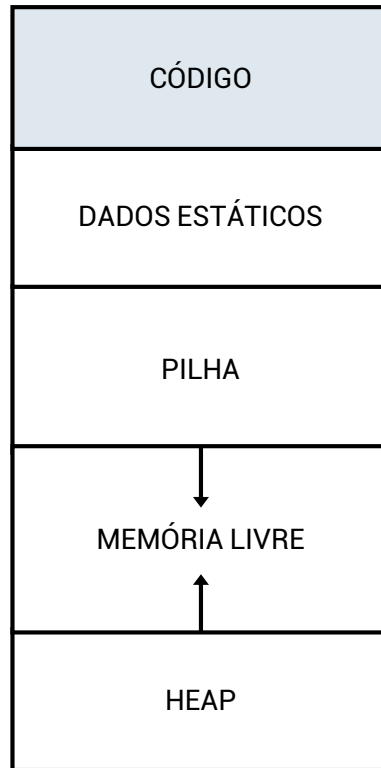


DINÂMICA
TEMPO DE EXECUÇÃO

MEMÓRIA

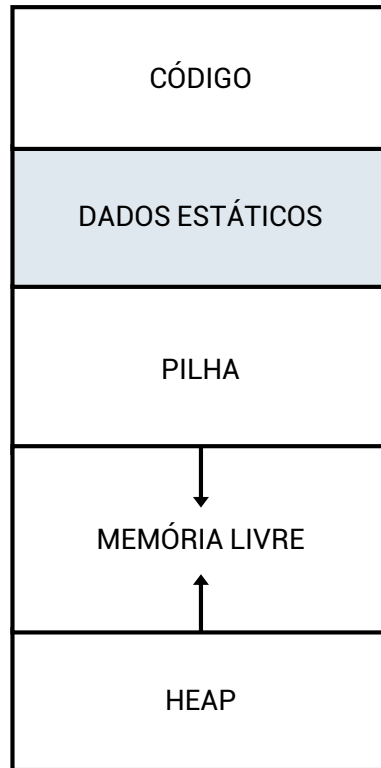


MEMÓRIA



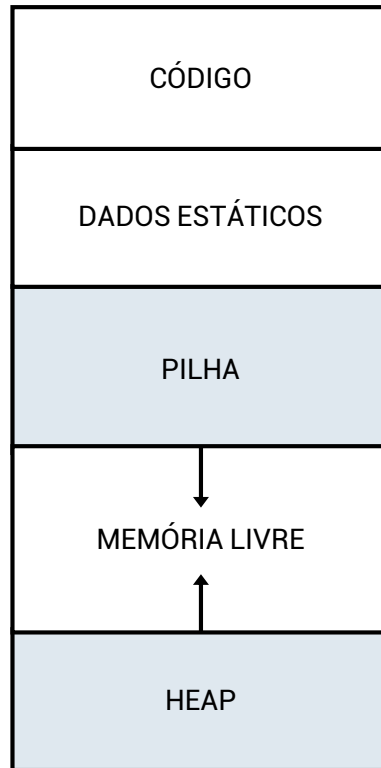
- Tamanho do código objeto gerado é determinado durante a compilação
- Código fica em uma área **código** determinada estaticamente

MEMÓRIA



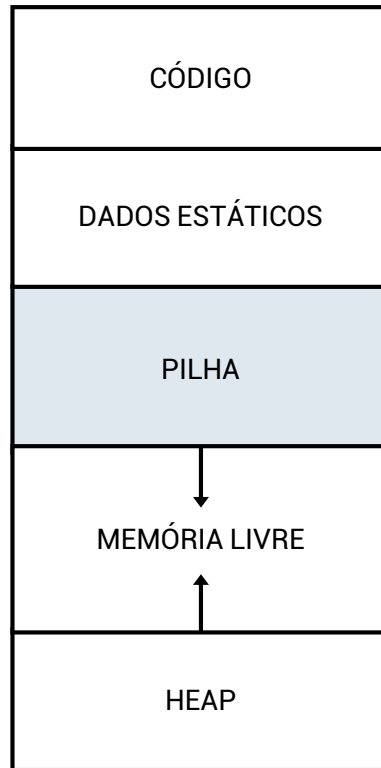
- Tamanhos conhecidos pelo compilador em tempo de compilação
 - Constantes globais
- Dados gerados pelo compilador (informações para coleta de lixo)
- Colocados em uma área determinada **estaticamente**

MEMÓRIA



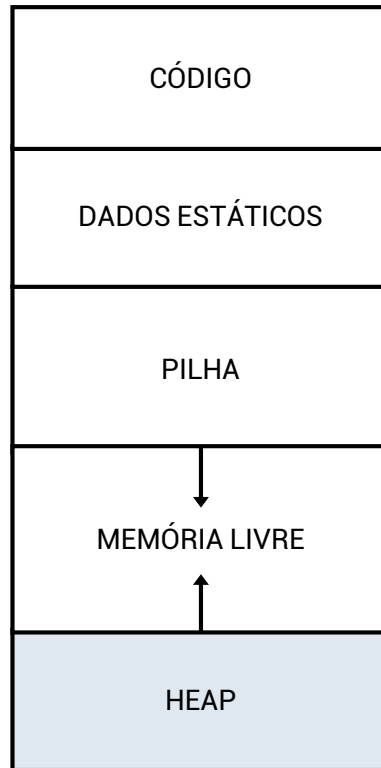
- Área de **pilha** e **heap**
- Tamanhos podem mudar
- Localizadas em extremidades **opostas**
- Crescem, conforme a necessidade, uma área em direção à outra

MEMÓRIA

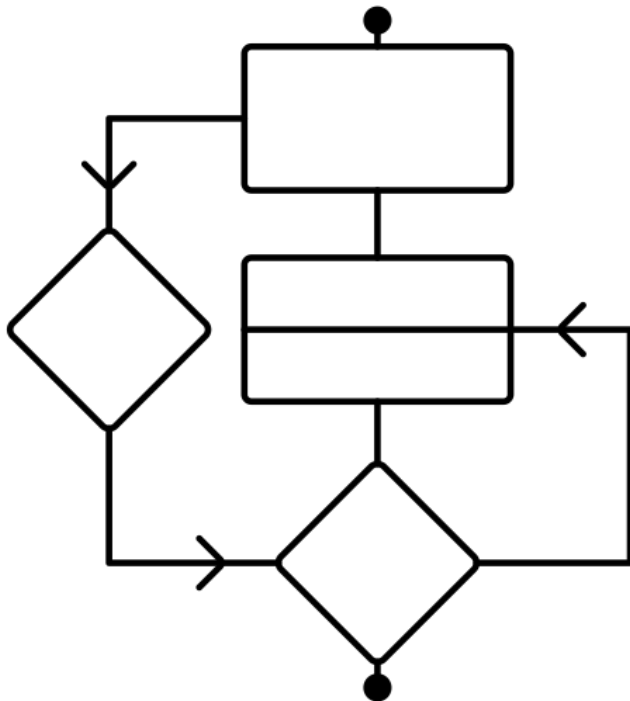


- Pilha é utilizada para armazenar **registros de ativação**
- Registros de ativação
 - Utilizado para armazenar estado da máquina
 - PC, registradores, ...
 - Chamada de procedimento
- Objetos de dados cujo tempo de vida estão contidos no de uma ativação

MEMÓRIA



- Alocação de memória sob controle do programa
- Dados de longa duração
 - `malloc`
 - `free`
- Coleta de lixo para gerenciar o **heap**



- Suposições sobre o fluxo de controle durante a execução de um programa:
 1. A **execução é sequencial**, i.e. controle se move de um ponto do programa para outro específico
 2. Quando um procedimento é chamado, o controle sempre **retorna** para o ponto imediatamente seguinte a chamada
- Invocação de um procedimento **p** é uma ativação de **p**
- O tempo de vida de uma ativação de **p** é equivalente a todos os passos para executar **p**, incluindo os procedimentos chamados por **p**

- Ativações de **a** e **b** ou são não sobrepostos ou são aninhados
- Podemos representar as ativações representar como uma árvore (**árvore de ativação**)

ESBOÇO DO PROGRAMA QUICKSORT

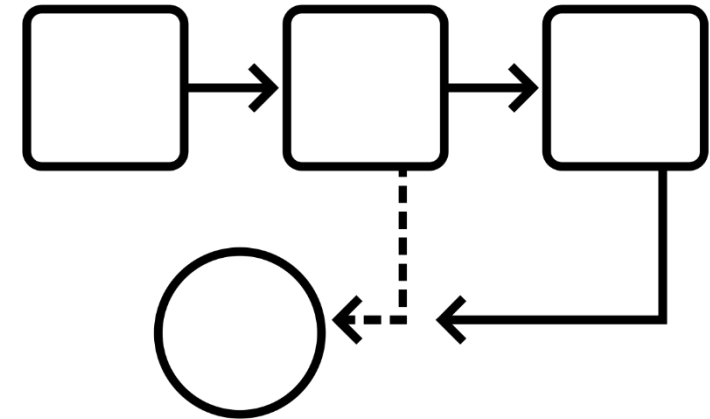
```
void quicksort (int m, int n) {
    int i;
    if (n > m) {
        i = partition (m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1, 9);
}
```

ATIVAÇÕES POSSÍVEIS

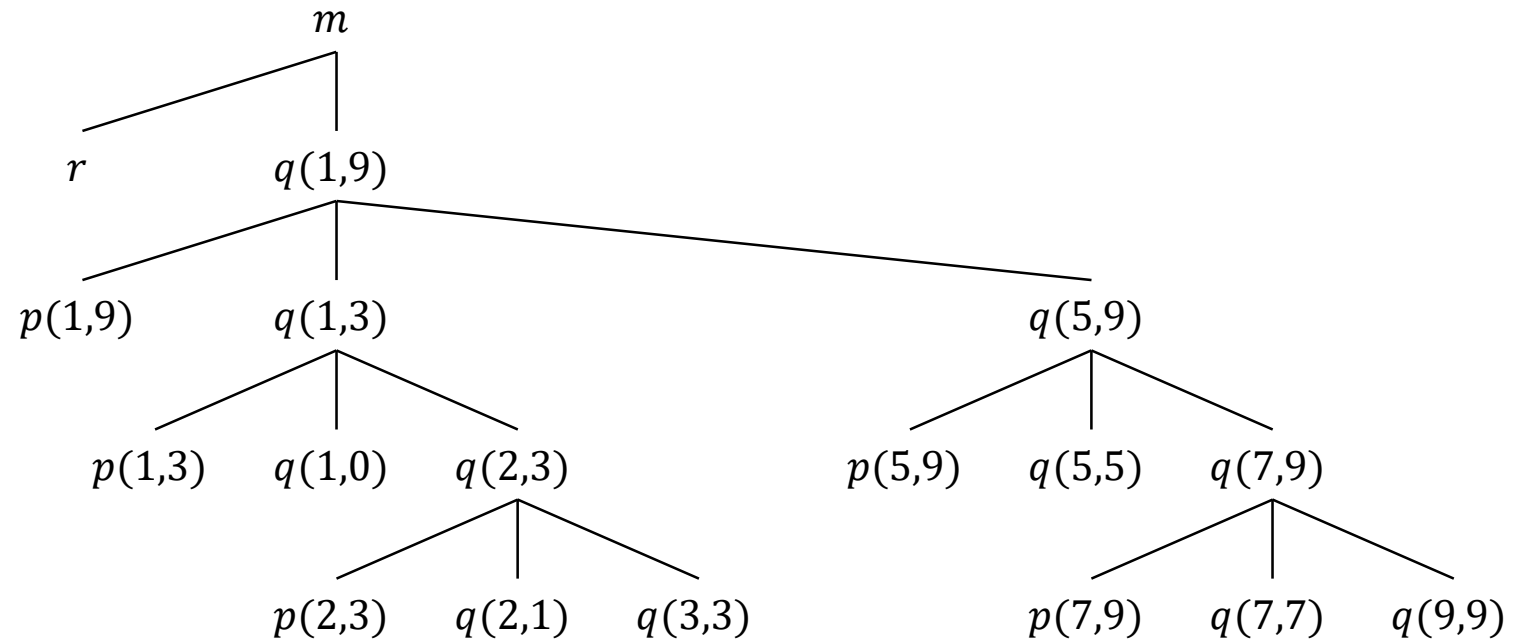
```
entra em main()
    entra em readArray()
    sai de readArray()
    entra em quicksort(1,9)
        entra em partition(1,9)
        sai de partition(1,9)
        entra em quicksort(1,3)
            ...
            sai de quicksort(1,3)
        entra em quicksort (5,9)
            ...
            sai de quicksort(5,9)
        sai de quicksort(1,9)
    sai de main()
```

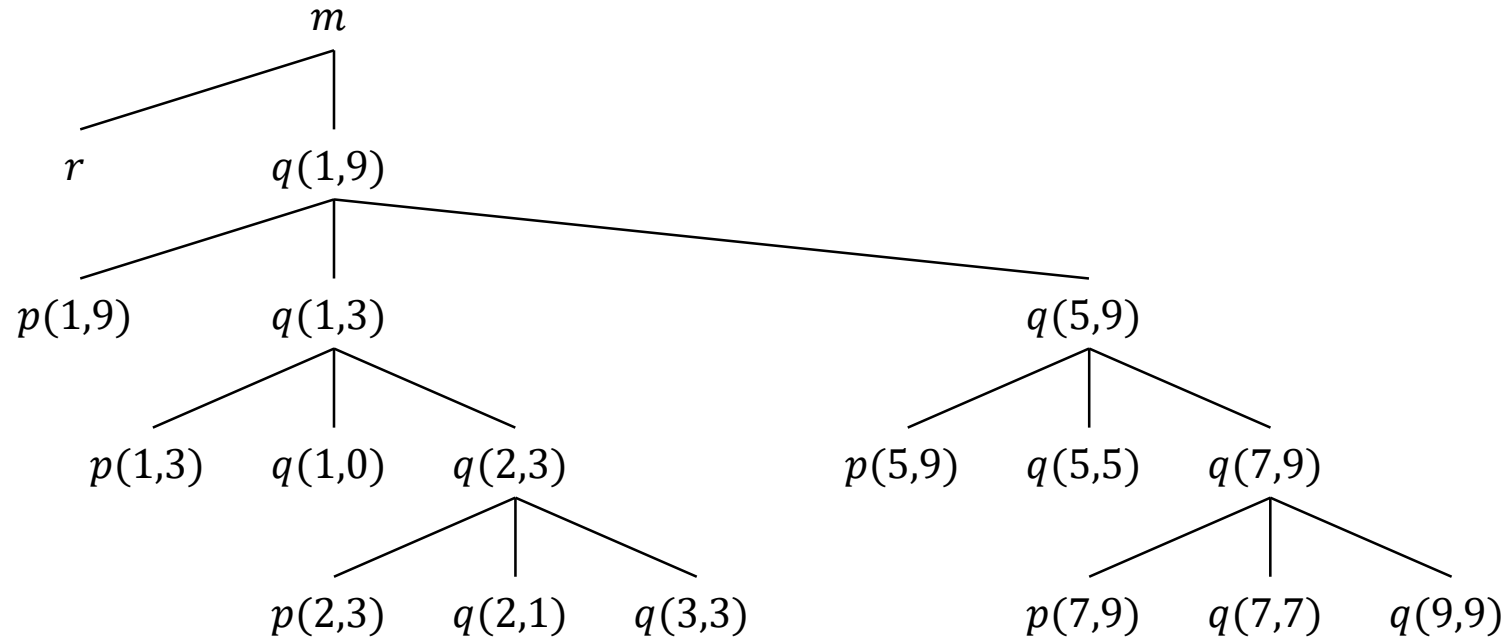
- Ativações são aninhadas no tempo (e.g. quicksort)
 - Ativação do procedimento **p** chama **q**
 - Para **p** terminar, **q** precisa ter terminado
- Casos comuns:
 1. Ativação de **q** termina normalmente, controle retoma
 2. Ativação de **q** ou algum procedimento chamado em **q** aborta, **p** e **q** terminam simultaneamente
 3. Ativação de **q** termina por exceção que **p** pode tratar ou **p** termina e algum procedimento aberto pode tratar



ATIVAÇÕES POSSÍVEIS

```
entra em main()
  entra em readArray()
  sai de readArray()
  entra em quicksort(1,9)
    entra em partition(1,9)
    sai de partition(1,9)
    entra em quicksort(1,3)
      ...
    sai de quicksort(1,3)
    entra em quicksort(5,9)
      ...
    sai de quicksort(5,9)
  sai de quicksort(1,9)
sai de main()
```





- Chamada de procedimentos (ativação) é caminho em **pré-ordem**
- Retorno de procedimentos um caminhamento **pós-ordem**
- Ativações abertas (vivas) do nó **N** são todos os seus ancestrais

EXERCÍCIO 1

```
Class Main {  
    g(): Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```

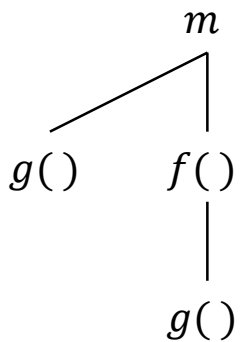
EXERCÍCIO 2

```
Class Main {  
    g() : Int { 1 };  
    f(x:Int): Int {  
        if x = 0 then g()  
        else f(x - 1) fi  
    };  
    main(): Int {{ f(3); }};  
}
```


EXERCÍCIO 1

```
Class Main {  
    g(): Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```

RESPOSTA



Pode existir múltiplas ativações
de um procedimento em uma
árvore de ativação

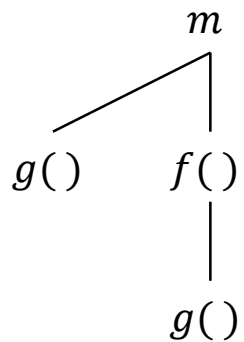
EXERCÍCIO 2

```
Class Main {  
    g() : Int { 1 };  
    f(x:Int): Int {  
        if x = 0 then g()  
        else f(x - 1) fi  
    };  
    main(): Int {{ f(3); }};  
}
```

EXERCÍCIO 1

```
Class Main {  
    g(): Int { 1 };  
    f(): Int { g() };  
    main(): Int {{ g(); f(); }};  
}
```

RESPOSTA



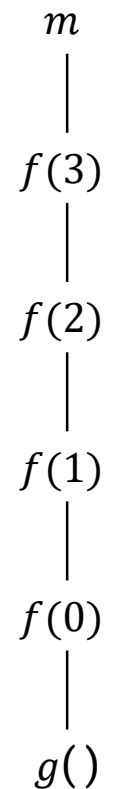
Pode existir múltiplas ativações de um procedimento em uma árvore de ativação

EXERCÍCIO 2

```
Class Main {  
    g() : Int { 1 };  
    f(x:Int): Int {  
        if x = 0 then g()  
        else f(x - 1) fi  
    };  
    main(): Int {{ f(3); }};  
}
```

Pode existir múltiplas ativações de um procedimento em uma árvore de ativação de forma recursiva

RESPOSTA



EXERCÍCIO 3

```
isEven(x:Int) : Bool { x % 2 == 0 };  
isOne(x:Int)  : Bool { x == 1 };  
powerOfTwo(x:Int) : Bool {  
    if isEven(x) then powerOfTwo(x / 2)  
    else isOne(x)  
};
```

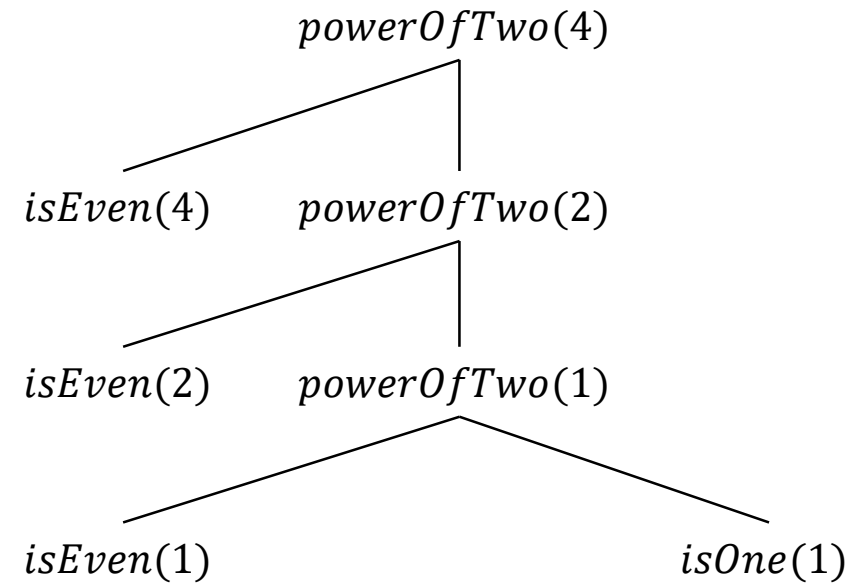
Como fica a árvore de ativação para **powerOfTwo(4)**?

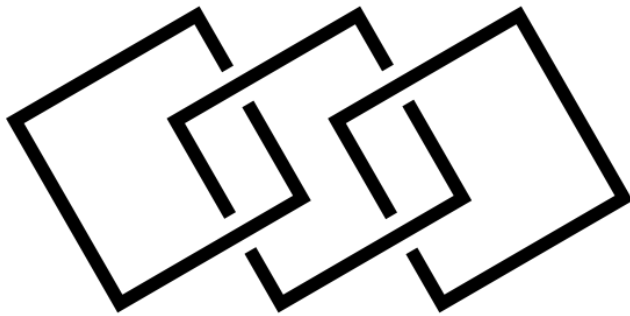
EXERCÍCIO 3

```
isEven(x:Int) : Bool { x % 2 == 0 };  
isOne(x:Int) : Bool { x == 1 };  
powerOfTwo(x:Int) : Bool {  
    if isEven(x) then powerOfTwo(x / 2)  
    else isOne(x)  
};
```

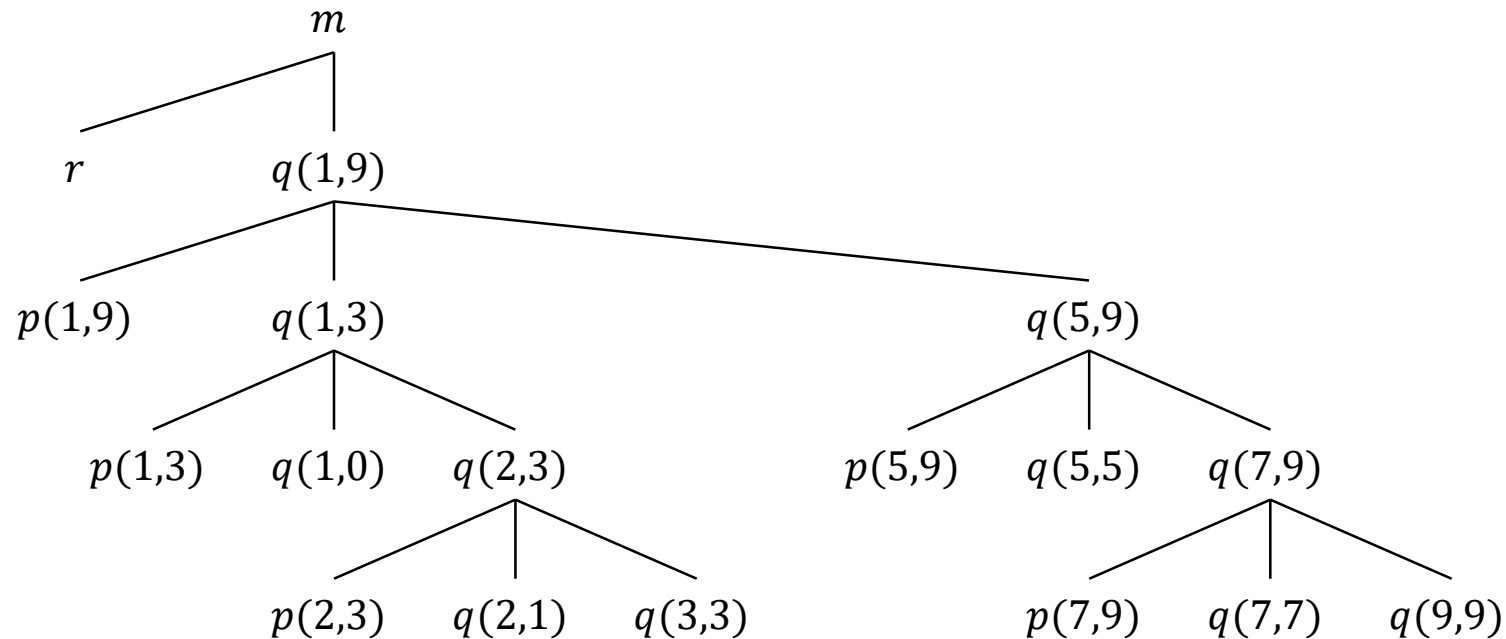
Como fica a árvore de ativação para **powerOfTwo(4)**?

RESPOSTA





- Árvore de ativação depende do comportamento da aplicação
- Pode ser diferente para cada entrada do programa
 - Exemplo do **powerOfTwo(3)** e **powerOfTwo(4)**
- Pode conter várias ativações de um mesmo procedimento
- Pode conter ativações recursivas de um procedimento (direta ou indiretamente)



Quem faz o gerenciamento das chamadas e retornos?

R. Pilha de execução camada de **pilha de controle**

- Compiladores gerenciam parte de sua memória em tempo de execução como uma **pilha**

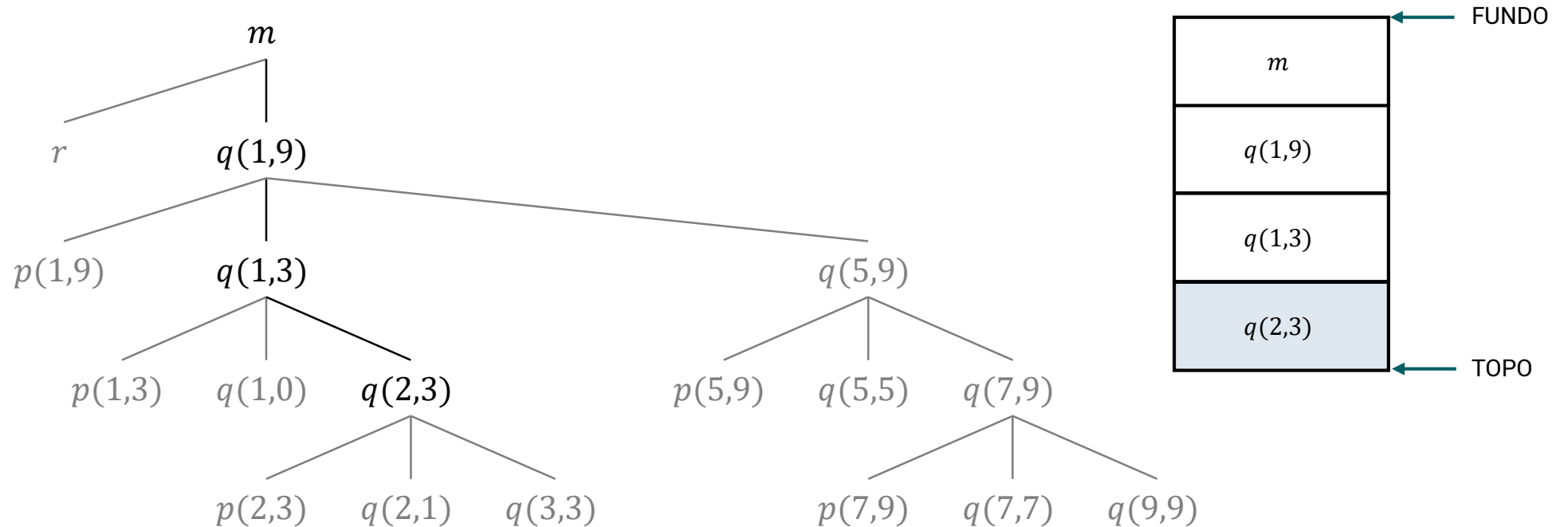


CHAMADA DE PROCEDIMENTO
O ESPAÇO PARA SUAS VARIÁVEIS
LOCAIS É **COLOCADO** NA PILHA



RETORNO DE PROCEDIMENTO
O ESPAÇO PARA SUAS VARIÁVEIS
LOCAIS É **RETIRADO** DA PILHA

- Permite que o espaço seja compartilhado pelas chamadas
- Desde que as durações dos procedimentos não se sobrepõem no tempo

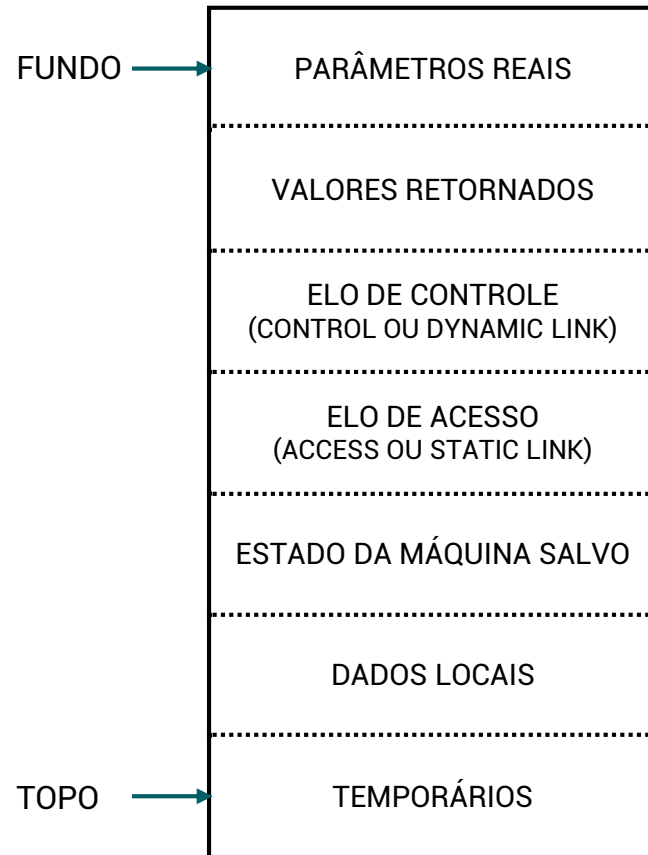


- Toda ativação viva tem um **registro de ativação** ou **frame** nesta pilha
- Se o controle estiver em **q(2,3)**, então este registro estará no topo da pilha
- Todos os outros registros estarão abaixo, até chegar ao **main (m)**

Conteúdo dos Registros de Ativação

INF01147 - COMPILADORES

35

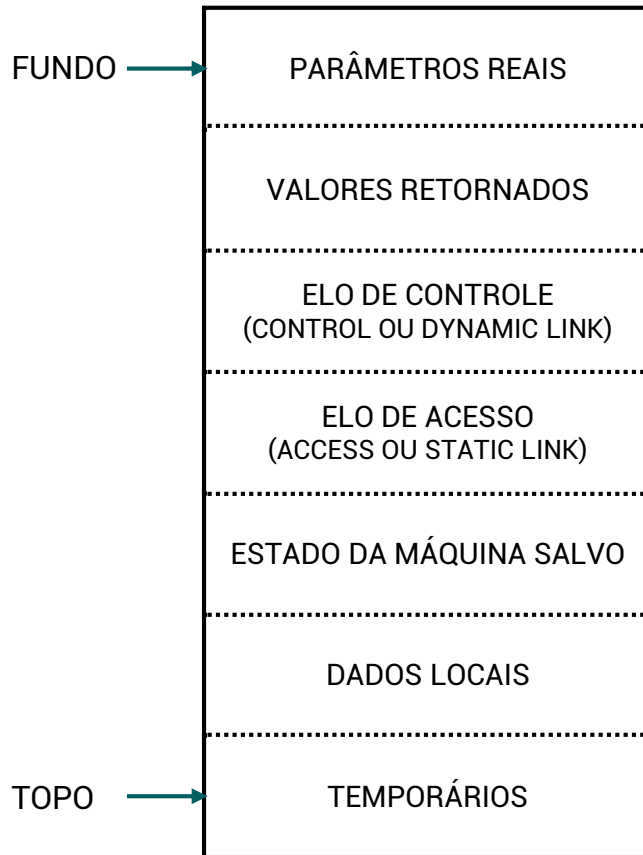


- **Temporários**
 - Valores usados na avaliação de expressões
 - Quando não podem ser mantidos nos registradores
- **Dados locais**
 - Dados que pertencem ao procedimento
- **Estado da máquina salvo**
 - Informações antes da chamada
 - Contador de programa (PC)
 - Endereço de retorno
 - Conteúdo dos registradores (restauração)

Conteúdo dos Registros de Ativação

INF01147 - COMPILADORES

36



- **Elo de acesso**
 - Encontrar informações em outros registros
- **Elo de controle**
 - Aponta para o registro de ativação que o chamou
- **Valores retornados**
 - Se houver algum
 - Podemos por em um registrador (eficiência)
- **Parâmetros reais**
 - Parâmetros usados pelo procedimento
 - Geralmente colocados em registradores (mais eficiente)

ESBOÇO DO PROGRAMA QUICKSORT

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

```
main() {  
    readArray();  
    a[0] = -9999;  
    a[10] = 9999;  
    quicksort(1, 9);  
}
```

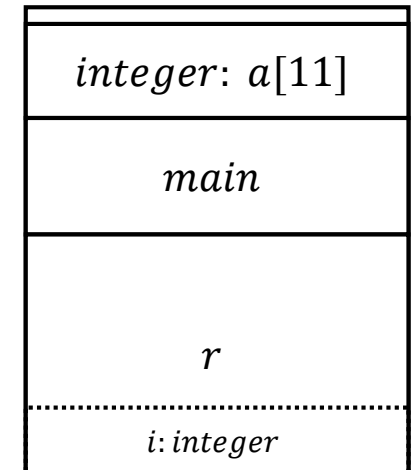
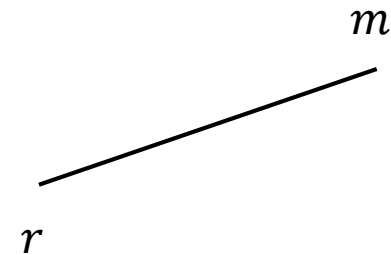
 m

<i>integer: a[11]</i>
<i>main</i>

ESBOÇO DO PROGRAMA QUICKSORT

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

```
main() {  
    readArray();  
    a[0] = -9999;  
    a[10] = 9999;  
    quicksort(1, 9);  
}
```



ESBOÇO DO PROGRAMA QUICKSORT

```

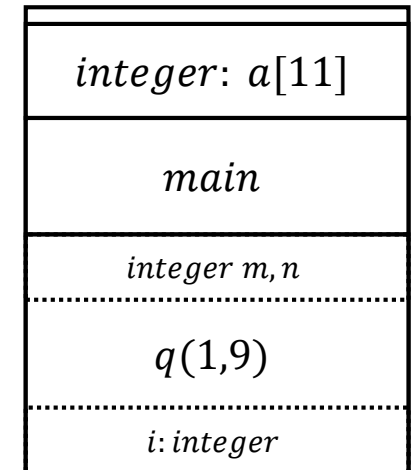
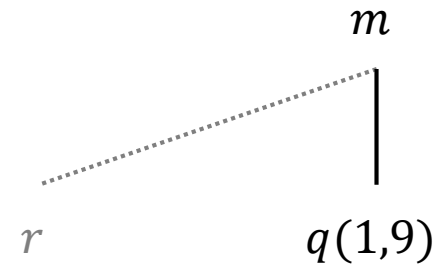
void quicksort (int m, int n) {
    int i;
    if (n > m) {
        i = partition (m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}

```

```

main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1, 9);
}

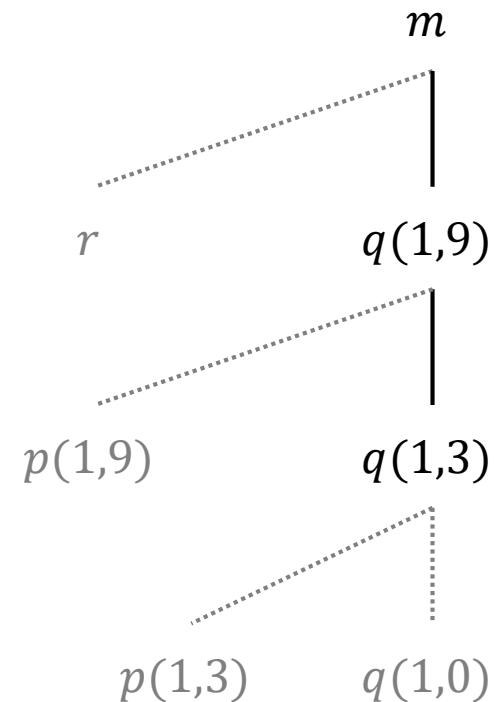
```



ESBOÇO DO PROGRAMA QUICKSORT

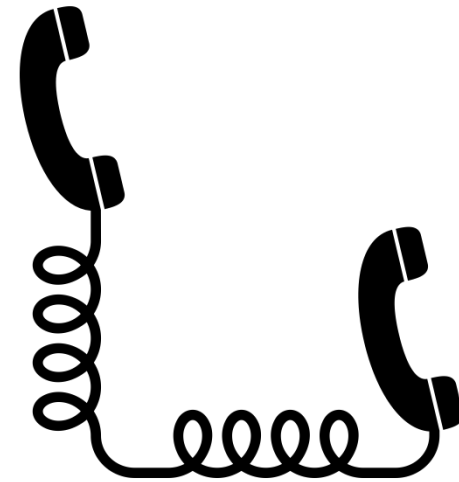
```
void quicksort (int m, int n) {
    int i;
    if (n > m) {
        i = partition (m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

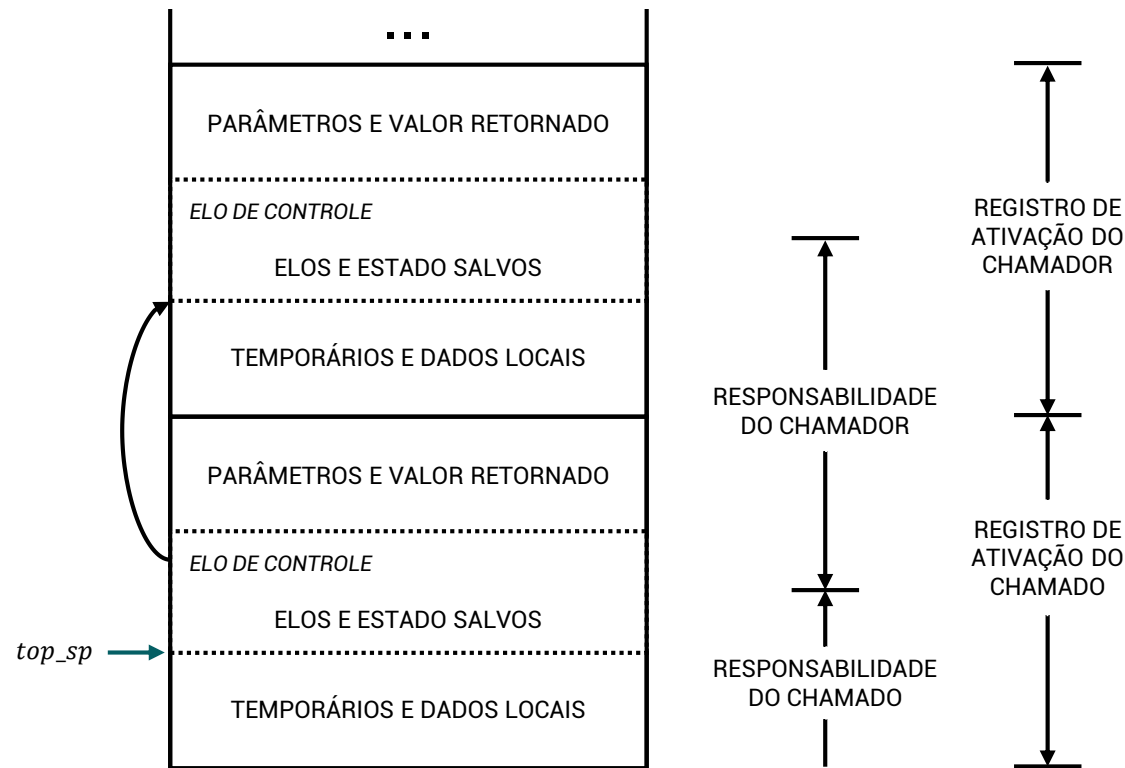
```
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1, 9);
}
```



<i>integer: a[11]</i>
<i>main</i>
<i>integer m, n</i>
<i>q(1,9)</i>
<i>i: integer</i>
<i>integer m, n</i>
<i>q(1,3)</i>
<i>i: integer</i>

- Como as chamadas de procedimentos são implementadas?
- **Sequências de chamadas**
 - Consistem em código que aloca um registro de ativação na pilha
 - Entra com as informações em seus campos
- **Sequências de retorno**
 - Código semelhante
 - Restaura o estado da máquina
 - Procedimento que chama pode continuar a sua execução
- Implementação pode diferir
 - Até mesmo entre implementações de uma mesma linguagem



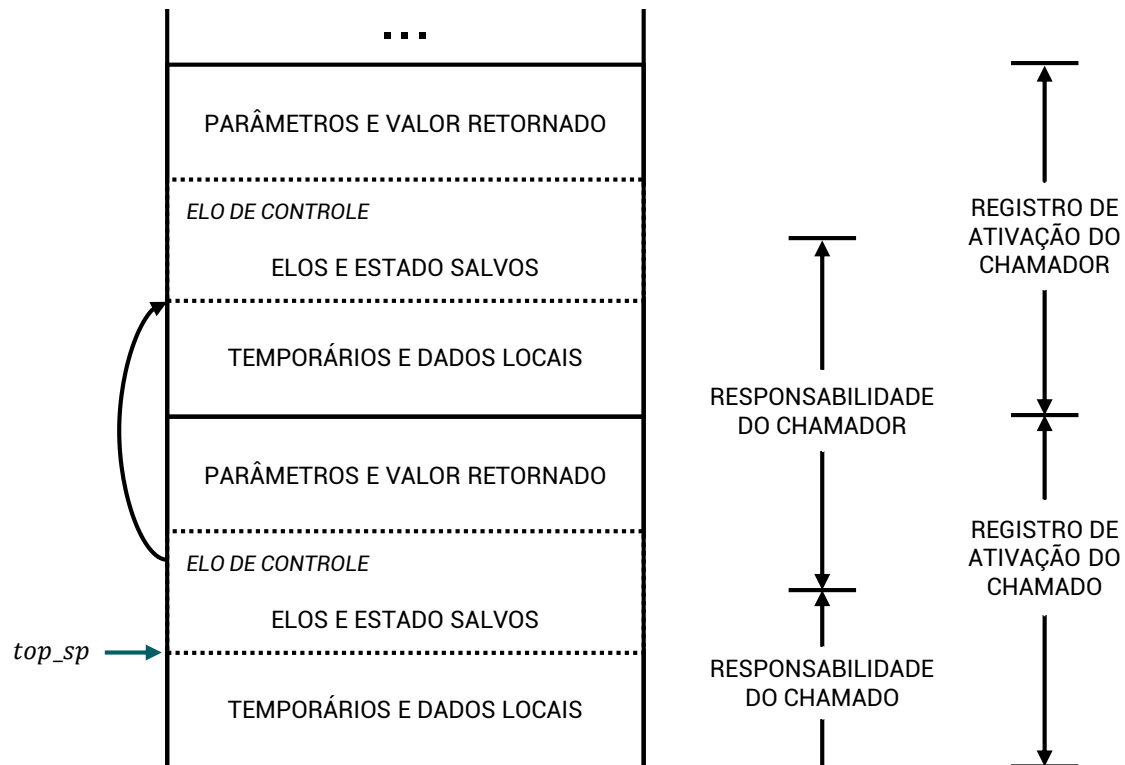


- Código de uma **sequência de chamadas** é dividido em:
 - Chamador (*caller*)
 - Chamado (*callee*)
- Valores comunicados são geralmente colocados no **início** do registro de ativação do chamado
- i.e. Próximos do registro de ativação do chamador.
- Porque?

Princípios ao Projetar

INF01147 - COMPILADORES

43

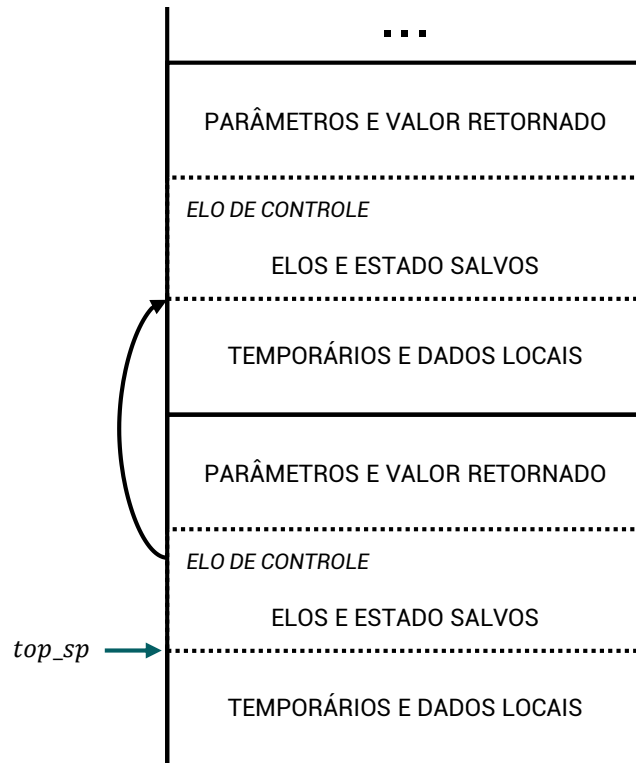


- Chamador pode calcular valores dos parâmetros reais da chamada
 - Pode colocar no topo do seu próprio registro
 - Não é necessário criar o registro do chamado
 - Não é necessário saber o layout desse registro
- Permite o uso de número variável de parâmetros
 - Exemplo `printf` do C
 - Conhece o primeiro argumento
 - Este pode localizar os seguintes

Princípios ao Projetar

INF01147 - COMPILADORES

44

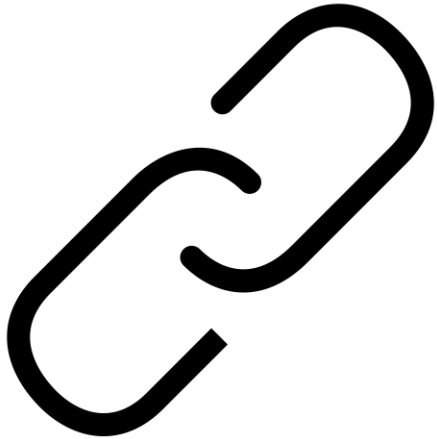


Sequência de chamada

- **Chamador** avalia seus parâmetros atuais
- **Chamador** armazena endereço de retorno no registro de ativação do **chamado**
- **Chamador** armazena o valor antigo de *top_sp* no registro do **chamado**
- **Chamado** salva os valores do registrador e outras informações de estado
- **Chamado** inicializa os seus dados locais
- **Chamado** começa a sua execução

Sequência de retorno

- **Chamado** coloca valor de retorno próximo dos parâmetros
- **Chamado** restaura *top_sp* e registradores
- **Chamado** desvia para endereço de retorno (informado pelo **chamador**)

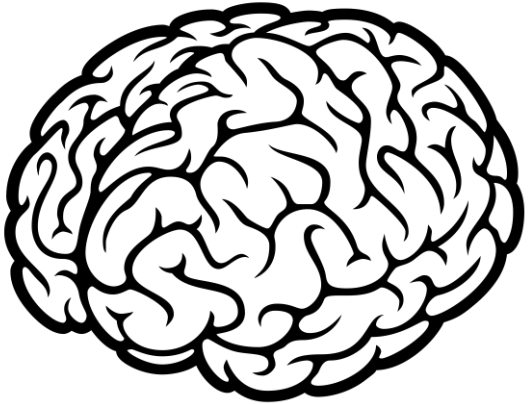


- Itens de tamanho **fixo** são geralmente colocados no **meio**
 - Elo de controle
 - Elo de acesso
 - Estado da máquina
- Se a mesma informação de estado da máquina for salvo para cada ativação
 - Mesmo código pode fazer salvamento e restauração
 - Auxilia depuradores a decifrar conteúdo da pilha (erro)
- Itens cujo tamanho **não** são **conhecidos** com antecedência são colocados no **fim** do registro de ativação
 - Exemplo de um *array* tamanho dinâmico
 - Parâmetro determina o tamanho do *array*

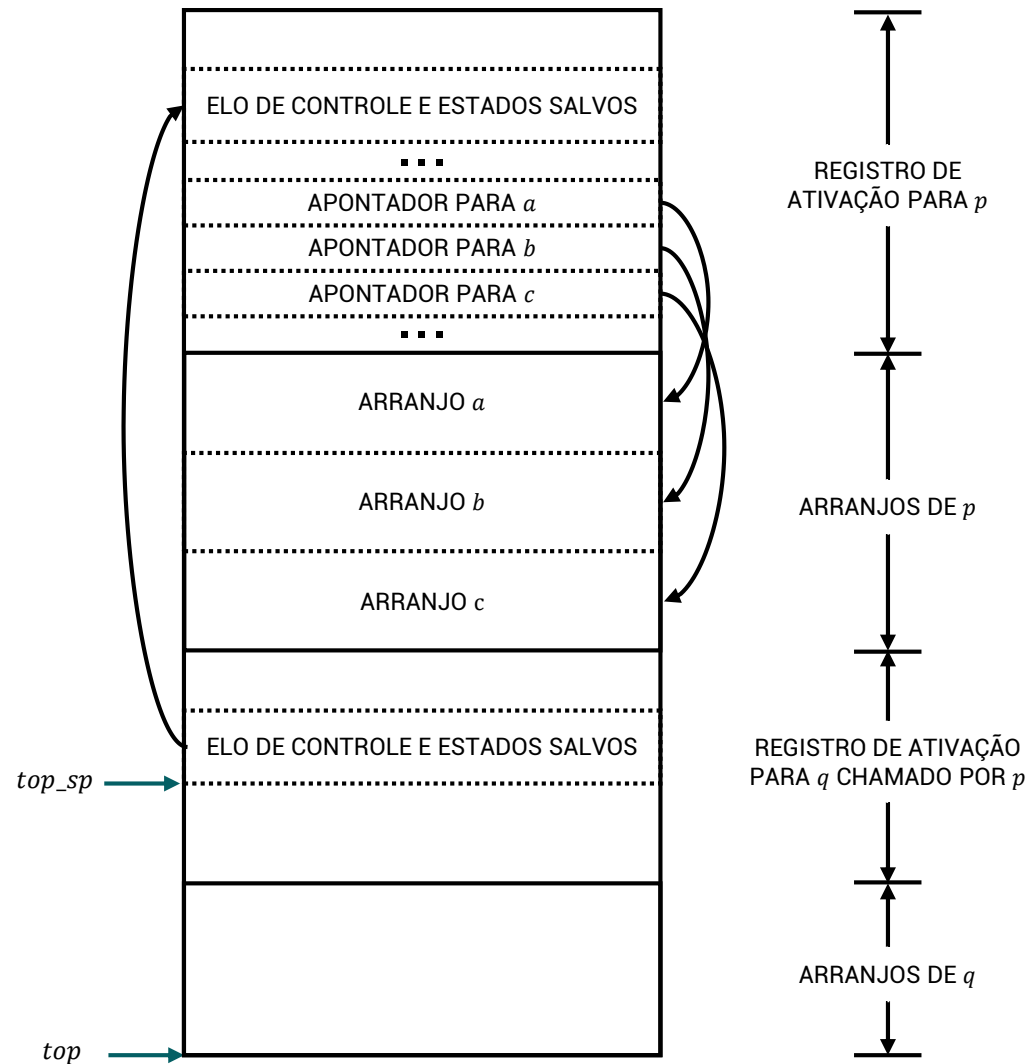
- Apontador do **topo** da pilha aponta para o **fim** dos campos de tamanho fixo
- Acesso ao dados de tamanho **fixo**:
 - Através de deslocamentos fixos
 - Conhecidos pelo gerador de código intermediário
 - Relativo ao topo da pilha
- Acesso ao dados de tamanho **variável**:
 - Deslocamento calculado em tempo de execução
 - Utiliza-se deslocamento contrário
 - Relativo ao topo da pilha



Dados de Tamanho Variável na Pilha



- Sistema de gerenciamento de memória em tempo de execução precisa alocar espaço para objetos com tamanhos desconhecidos (durante a compilação)
- Como podem ser variáveis locais a um procedimento, podem ser alocadas na pilha
- *Arrays*, objetos e estruturas
 - É melhor colocar objetos na pilha
 - Evita coleta de lixo
 - Objetos deve ser local a um procedimento
 - Objeto fica inacessível quando procedimento retorna
- Linguagens modernas alocam este espaço no *heap*

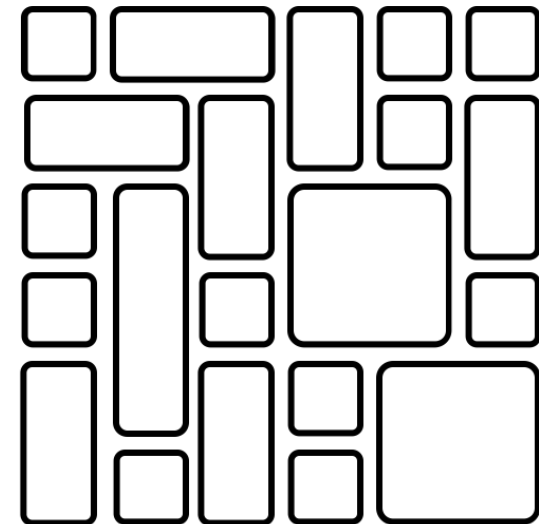


- Área de memória do arranjo **não** faz parte do registro
- Registro contém apenas um **apontador**
- Deslocamento é conhecido em tempo de execução
- **top**: topo da pilha (início do próximo registro)
- **top_sp**: campos locais de tamanho fixo do registro
- Elo de ativação permite encontrar **p**



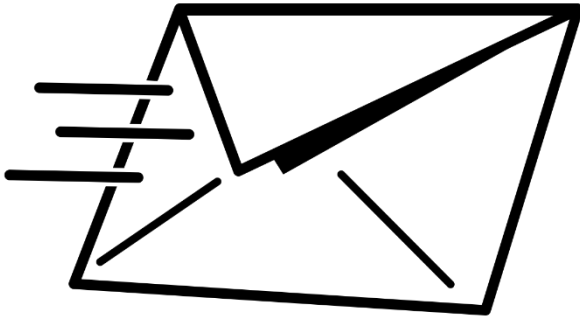
- Como procedimentos acessam seus dados?
- Como localizar dados usados dentro de um procedimento **p** mas que não pertencem a **p**?
- Regras de escopo de uma linguagem
- Determinam o tratamento das referências aos nomes não locais
 - Escopo **estático**
 - Sem procedimentos aninhados
 - Com procedimentos aninhados
 - Escopo **dinâmico**

- Nestes casos a alocação de memória é simples
 1. Variáveis globais são alocadas em uma área de memória estática
 - Endereços fixos
 - Endereços conhecidos em tempo de compilação
 - Acesso a variáveis não local utiliza-se endereço determinado estaticamente
 2. Qualquer outro nome deve ser local
 - Local à ativação no topo da pilha
 - Acesso através do apontador **top_sp**



Com Procedimento Aninhados

- Acesso se torna mais complicado em linguagens que permite declarações de procedimentos aninhados
- Usa a regra do escopo estático
- Saber (em tempo de compilação) se **p** está aninhada imediatamente dentro de **q** não nos informa a posição relativas de seus registros em tempo de execução
- Tanto **p** quanto **q**, ou ambos, podem ser recursivos, existindo vários registros de ativação de **p** / **q** na pilha
- Podemos utilizar elos de acesso para tentar resolver o problema
 - Armazenando a profundidade na declaração de **a**
 - Determinar quantos elos percorrer e calculando o deslocamento dentro do registro de ativação para encontrar **a**



- Passagem por **valor**:
 - Método simples e trivial
 - Cria-se uma cópia do parâmetro real \rightarrow parâmetro formal
 - Usa-se o parâmetro formal como uma variável local
- Passagem por **referência**:
 - Chamador passa o endereço de cada parâmetro real
 - Caso for um identificador:
 - Endereço é fornecido
 - Endereço pode ser na pilha, *heap*...
 - Caso for uma expressão:
 - Avalia-se a expressão
 - Coloca-se seu resultado em um temporário
 - Endereço do temporário é fornecido