Ambientes de Execução em Compiladores

Disciplina SCC0217 (2025) — Grupo 1, Turma 2

Nome	NUSP
Guilherme de Abreu Barreto	12543033
Hélio Nogueira Cardoso	10310227
Laura Fernandes Camargos	13692334
Sandy da Costa Dutra	12544570
Theo da Mota dos Santos	10691331

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo, ICMC - USP

21 de maio de 2025

Introdução

O que é um ambiente de execução?

- Estrutura de registros de memória do computador-alvo

Introdução

O que é um ambiente de execução?

- Estrutura de registros de memória do computador-alvo
- Responsável por:
  - Gerenciamento de memória (alocação/liberamento)

Introdução

O que é um ambiente de execução?

- Estrutura de registros de memória do computador-alvo
- Responsável por:
  - Gerenciamento de memória (alocação/liberamento)
  - Manutenção das informações necessárias para execução

Introdução

O que é um ambiente de execução?

- Estrutura de registros de memória do computador-alvo
- Responsável por:
  - Gerenciamento de memória (alocação/liberamento)
  - Manutenção das informações necessárias para execução
  - Controle do fluxo de chamadas de funções/procedimentos

- Baseado em pilhas
  - C/C++, Pascal

- Baseado em pilhas
  - C/C++, Pascal
- Totalmente estático
  - FORTRAN77

- Baseado em pilhas
  - C/C++, Pascal
- Totalmente estático
  - FORTRAN77
- Totalmente dinâmico
  - Linguagens funcionais como Lisp e Haskell

Ambientes Baseados em Pilhas

Aspectos Gerais

- Estrutura composta por pilha, heap e espaço livre

Ambientes Baseados em Pilhas

Aspectos Gerais

- Estrutura composta por pilha, heap e espaço livre
- Armazenamento de registros de ativação de procedimentos

Ambientes Baseados em Pilhas

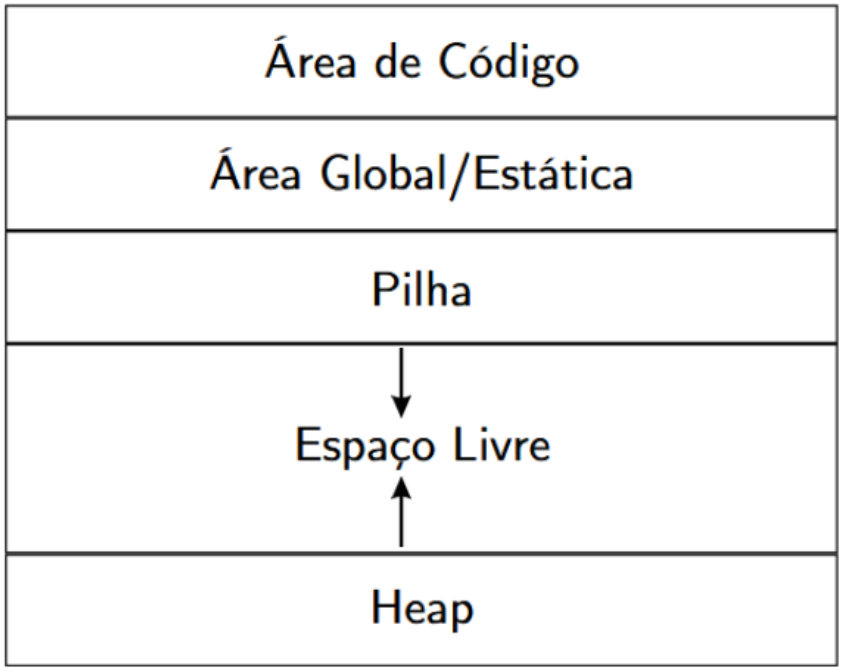
Aspectos Gerais

- Estrutura composta por pilha, heap e espaço livre
- Armazenamento de registros de ativação de procedimentos
- Controle de fluxo por registradores

Ambientes Baseados em Pilhas

Regiões de Memória

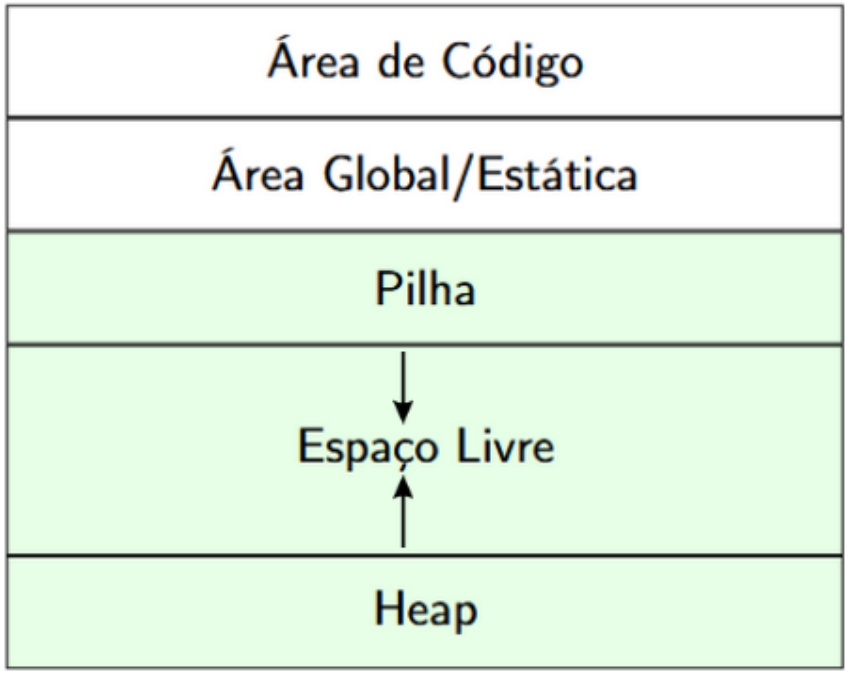
- Área de Código: Instruções, valores constantes e variáveis globais. Todas as informações para as quais o endereço de memória é conhecido e estático.
- Área de Dados: Todos os demais dados cujo endereço de memória muda conforme cada contexto de execução do programa.



Ambientes Baseados em Pilhas

Área de dados

- Área de Pilha: dados cuja alocação ocorre na forma LIFO.
- Área de Heap: onde são armazenados dinamicamente quaisquer outros dados que não seguem essa ordenação.
- Espaço Livre: área de memória disponível para alocação tanto pela pilha quanto pelo heap.



Ambientes Baseados em Pilhas

Registro de ativação

- Durante a execução do programa, cada chamada de função cria na pilha um registro de ativação.
- Um registro de ativação é composto por, no mínimo, espaço para os seguintes elementos:
  - Argumentos (ou parâmetros)
  - Dados locais
  - Endereço de retorno

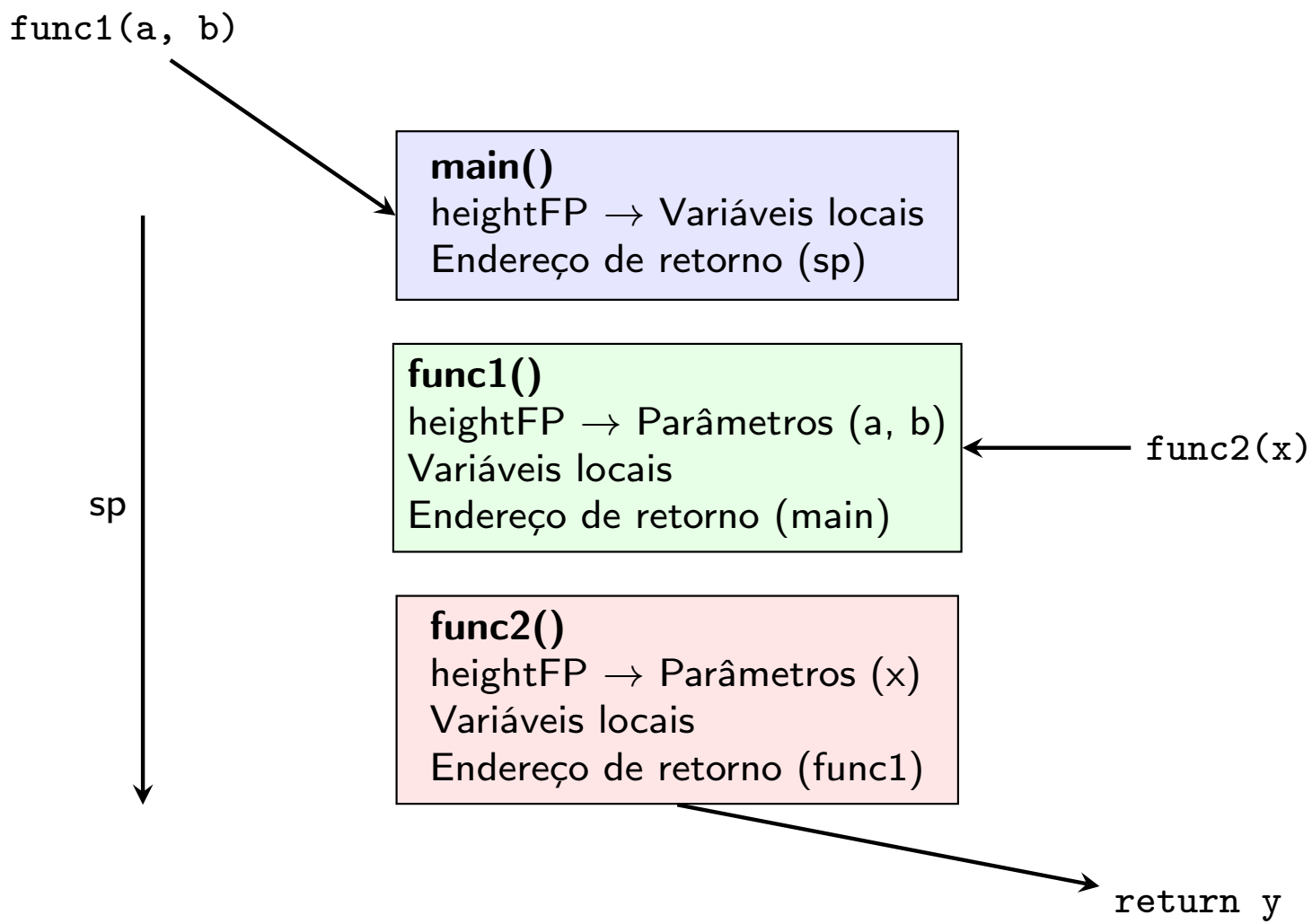
Ambientes Baseados em Pilhas

Registradores

- Armazenam valores pertinentes ao momento atual da execução de um programa
- Registradores de uso específico para acompanhar a execução:
  - Contador de programa (pc)
  - Ponteiro de pilha(sp)
  - Ponteiro de quadros (fp)
  - Ponteiro de argumentos (ap)
- Para que servem?

Ambientes Baseados em Pilhas

Exemplo de chamada e retorno de função



Ambientes Baseados em Pilhas

O que é uma Sequência de Ativação?

- Conjunto de ações realizadas ao chamar e retornar de um procedimento ou função.
- Responsável por preparar o ambiente de execução para a função chamada.
- Garante o correto armazenamento e restauração de informações essenciais.

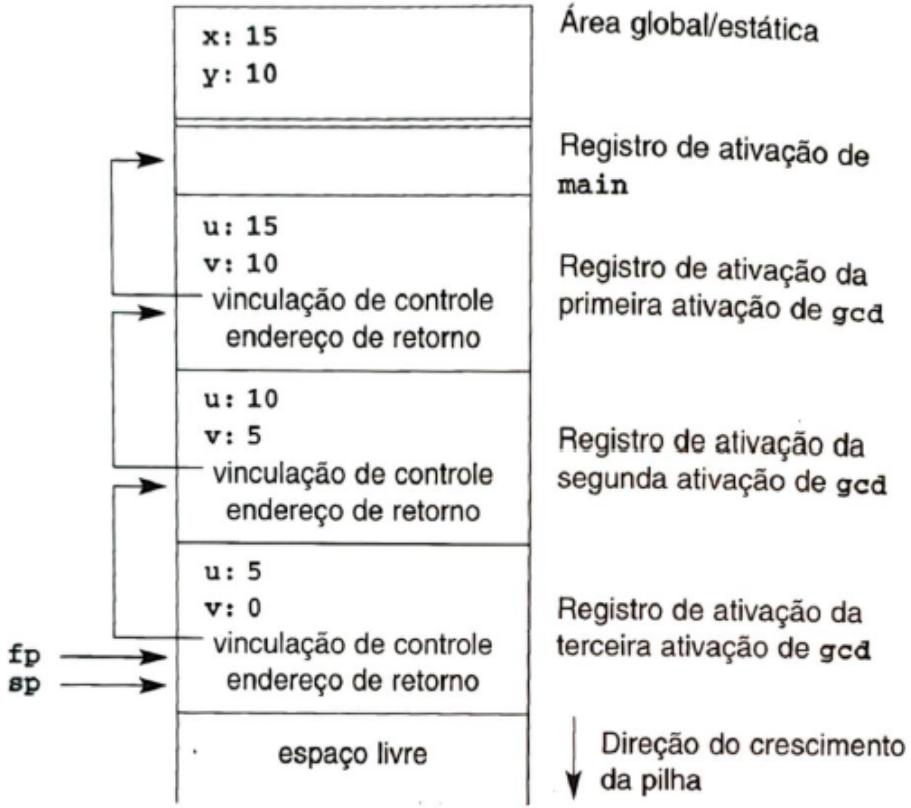
Ambientes Baseados em Pilhas

Etapas da Sequência de Ativação

1. Avaliação e passagem dos argumentos.
2. Salvamento do endereço de retorno.
3. Atualização dos registradores (FP, SP).
4. Alocação de espaço para variáveis locais.
5. Transferência de controle para o procedimento.

Ambientes Baseados em Pilhas

Exemplo Visual: Ativação de Função



- Pilha de execução cresce a cada chamada.
- Cada quadro de ativação contém parâmetros, variáveis locais e endereço de retorno.

Ambientes Baseados em Pilhas

Sequência de Retorno

- Restauração dos valores antigos de FP e SP.
- Recuperação do endereço de retorno.
- Liberação do espaço das variáveis locais.
- Retorno do controle para o chamador.

Ambientes Baseados em Pilhas

Importância das Sequências de Ativação

- Suporte à recursão e chamadas aninhadas.
- Isolamento de contextos de execução.
- Facilita o gerenciamento de memória e depuração.

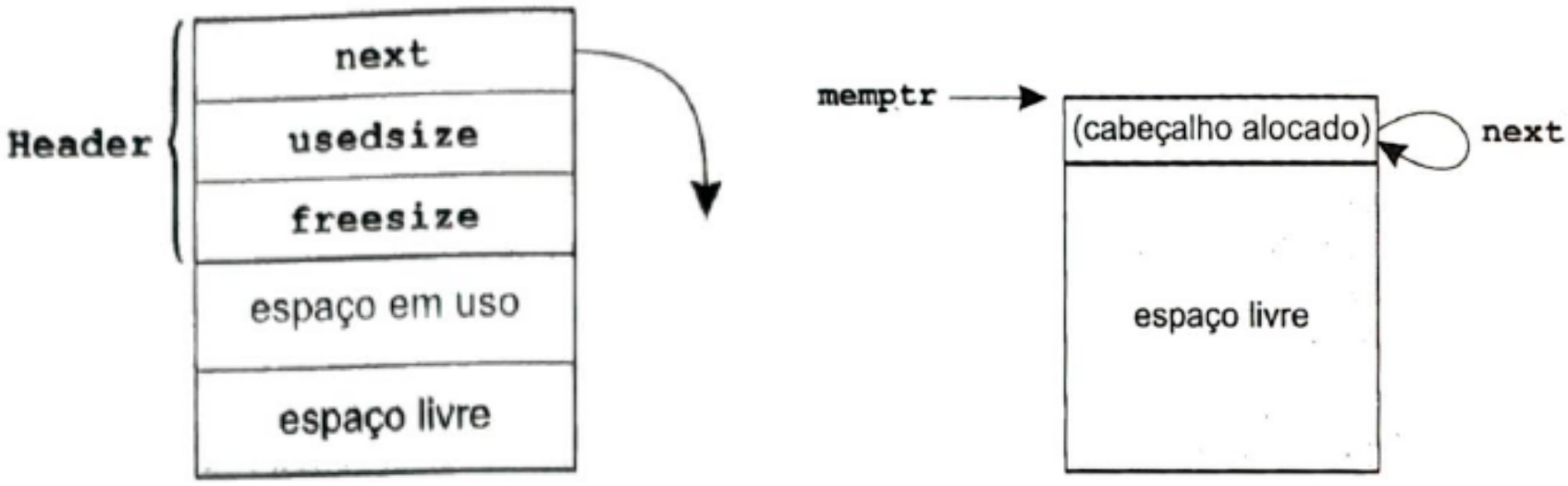
Ambientes Baseados em Pilhas

O que é o Heap?

- Região de memória para alocação dinâmica em tempo de execução.
- Utilizado para objetos, estruturas e variáveis cujo tempo de vida não é conhecido em tempo de compilação.
- Gerenciado pelo sistema de execução (runtime).

Ambientes Baseados em Pilhas

Estrutura do Heap



- Organização em blocos livres e ocupados.
- Listas ligadas (simples ou circulares) para rastrear blocos livres.
- Fragmentação interna e externa.

- `malloc`, `free` (C) ou `new`, `delete` (C++).
  - Ponteiro para um dos blocos livres da lista ligada circular (memptr)
  - Busca por espaço de alocação
- Algoritmos de alocação: first-fit, best-fit, worst-fit.

Ambientes Baseados em Pilhas

Desafios no Gerenciamento de Heap

- Fragmentação da memória.
- Overhead de gerenciamento.
- Concorrência e sincronização em ambientes multi-thread.
- Acessos inválidos.

Ambientes Baseados em Pilhas

Sem Procedimentos locais

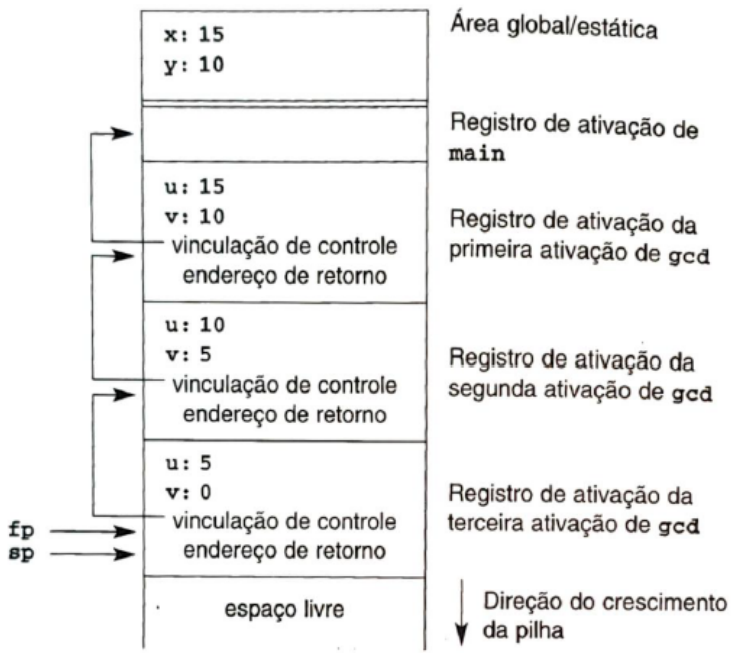
- Todas as funções são globais
  - Todas compartilham o mesmo escopo global, facilitando o acesso e a chamada entre elas.
- Controle de execução via registradores
  - O fluxo de chamadas e retornos é gerenciado por registradores como PC (Program Counter), SP (Stack Pointer) e FP (Frame Pointer), sem necessidade de estruturas adicionais para escopo local.

Ambientes Baseados em Pilhas

Exemplo em C

- GCD com funções globais

```c
#include <stdio.h>

int x, y;

int gcd(int u, int v) {
    if (v == 0) return u;
    else return gcd(v, u%v);
}

int main() {
    scanf("%d %d", &x, &y);
    printf("%d\n", gcd(x,y));
    return 0;
}
```



Ambientes Baseados em Pilhas

Com Procedimentos locais

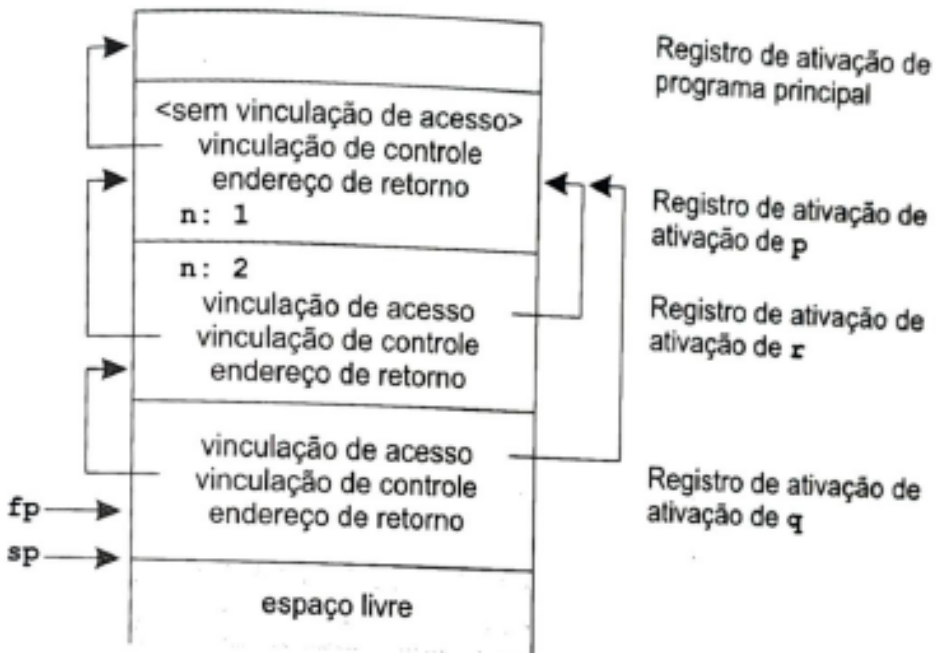
- **Funções aninhadas com contexto local:** Permite declarar funções dentro de outras funções, criando um escopo interno onde variáveis e procedimentos são visíveis apenas localmente.
- **Acesso a variáveis locais da função superior:** Funções internas podem acessar variáveis declaradas nas funções externas, possibilitando compartilhamento de contexto e mecanismos de escopo léxico.
- Nova variável de controle: *vinculação de acesso*.

Ambientes Baseados em Pilhas

Exemplo em Pascal

- Procedimentos aninhados e contexto local

```
progran nonLocalRef;

procedure p;
var n: integer;

  procedure q;
  begin
    (* uma referência a n é agora não
       local e não global *)
  end; (* q *)

  procedure r(n: integer);
  begin
    q;
  end; (* r *)

begin (* p *)
  n := 1;
  r(2);
end; (* p *)

begin (* main *)
  p;
end;
```



Ambientes Baseados em Pilhas

Com parâmetros de procedimentos

- Em algumas linguagens (ex: Pascal, Lua), é possível passar funções como parâmetros.
- Isso exige que o ambiente onde a função foi definida também seja transmitido.
- Esse ambiente contém as variáveis às quais a função ainda se refere.
- Assim, ao passar uma função, transmite-se um par <**ip, ep**>:
  - **ip** = ponteiro para o código da função
  - **ep** = ponteiro para o ambiente onde ela foi definida

Ambientes Baseados em Pilhas

Exemplo em Lua: Passagem de Funções com Ambiente

```lua
function p(func)
    func()
end

function q()
    local x = 10
    local function r()
        print(x)
    end
    p(r)
end

q()  --> Imprime 10
```

- A função `r` acessa a variável local `x` de `q`.
- Ao passar `r` como argumento, Lua preserva o ambiente onde `r` foi criada.

Ambientes Baseados em Pilhas

Resumo

- Ao passar funções como argumentos, o ambiente de definição deve ser preservado.
- Isso garante acesso às variáveis locais externas da função.
- Esse mecanismo é a base para **closures** em muitas linguagens.
- A dupla `<ip, ep>` viabiliza isso de forma eficiente.

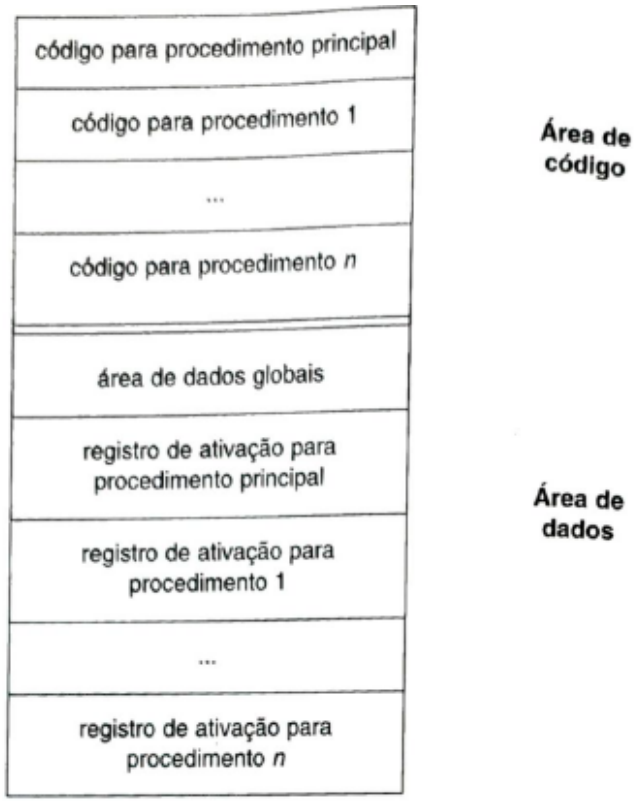
Ambientes de Execução Totalmente Estáticos

Introdução

- Todos os dados possuem posições fixas na memória.
- Não há pilha nem heap: o layout de memória é completamente previsível.
- Requisitos para esse ambiente:
  - Sem ponteiros
  - Sem alocação dinâmica
  - Sem recursão
- Vantagem: maior desempenho e uso eficiente de registradores.
- Só é necessário um registrador de controle: o **registrador de retorno**.

Ambientes de Execução Totalmente Estáticos

Organização da Memória Estática



- Todas as variáveis têm endereços fixos.
- Variáveis globais e locais coexistem na memória estática.
- Nenhuma mudança ocorre em tempo de execução.

Ambientes de Execução Totalmente Dinâmicos

Introdução

- Baseados em alocação dinâmica completa.
- Permitem reter variáveis locais mesmo após o fim de uma função.
- São comuns em linguagens funcionais como **Haskell** e **LISP**.
- A pilha tradicional não é suficiente para esse comportamento.
- Exigem estruturas de ambiente mais complexas, como **closures**.

Ambientes de Execução Totalmente Dinâmicos

Problema: Referência Pendente em C

```c
int *dangle(void) {
    int x;
    return &x;
}
```

- A variável `x` é alocada na pilha e liberada ao fim da função.
- Retornar o endereço de `x` cria um **dangling pointer (ponteiro pendente)**.
- Acesso posterior leva a comportamento indefinido ou falha de segurança.

Ambientes de Execução Totalmente Dinâmicos

Closures e Ambientes Dinâmicos

- Uma **closure** consiste em:
  - Ponteiro para o código da função
  - Estrutura com variáveis capturadas (ambiente)
- Essa estrutura é alocada dinamicamente — sobrevive fora da pilha.
- Ambientes de execução totalmente dinâmicos dão suporte nativo a isso.
- Muito usados para composição de funções e programação reativa.

Ambientes de Execução Totalmente Dinâmicos

Solução com Haskell

```haskell
SomaX :: Int -> Int -> Int
SomaX x = \y -> x + y  -- Retorna função que captura x

somaCinco :: Int -> Int
somaCinco = SomaX 5    -- "Fecha" sobre x = 5

resultado :: Int
resultado = somaCinco 10  -- Resultado é 15
```

- A função `somaCinco` mantém o valor de `x` mesmo após sua criação.
- Isso é possível porque o ambiente que contém `x` é **preservado dinamicamente**.
- A função carrega consigo o seu ambiente de definição — isso é uma **closure**.

Ambientes de Execução Totalmente Dinâmicos

Resumo

- Ambientes dinâmicos permitem reutilização de variáveis locais após o fim de sua função.
- Fundamentais para linguagens funcionais.
- Inviáveis em linguagens baseadas exclusivamente em pilha como C.
- Implementações usam **heap** e **closures** para preservar ambientes.

Ambientes de Execução Totalmente Dinâmicos

Coleta de Lixo (*garbage collection*)

Abordagem de gerenciamento de memória caraterística de ambientes de execução totalmente dinâmicos.

Ambientes de Execução Totalmente Dinâmicos

Coleta de Lixo (*garbage collection*)

Abordagem de gerenciamento de memória caraterística de ambientes de execução totalmente dinâmicos. Seu uso requer:

- o acompanhamento das referências durante a execução;

Ambientes de Execução Totalmente Dinâmicos

Coleta de Lixo (*garbage collection*)

Abordagem de gerenciamento de memória caraterística de ambientes de execução totalmente dinâmicos. Seu uso requer:

- o acompanhamento das referências durante a execução;
- capacidade para encontrar e liberar porções da memória já inacessíveis em instantes arbitrários durante a execução.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo. Neste, a cada sequência de ativação, ocorrem duas passadas pelo *heap*.

- Na primeira passada:

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo. Neste, a cada sequência de ativação, ocorrem duas passadas pelo *heap*.

- Na primeira passada:
  - Percorre-se todos os ponteiros ao heap recursivamente;

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo. Neste, a cada sequência de ativação, ocorrem duas passadas pelo *heap*.

- Na primeira passada:
  - Percorre-se todos os ponteiros ao heap recursivamente;
  - Marca-se cada bloco acessado desta forma com um bit de validação, indicando que este é "alcançável".
- Na segunda passada:

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo. Neste, a cada sequência de ativação, ocorrem duas passadas pelo *heap*.

- Na primeira passada:
  - Percorre-se todos os ponteiros ao heap recursivamente;
  - Marca-se cada bloco acessado desta forma com um bit de validação, indicando que este é "alcançável".
- Na segunda passada:
  - Percorre-se o *heap* linearmente, liberando a memória de todos os blocos que não foram marcados como sendo alcançáveis.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Principal algoritmo de coleta de lixo. Neste, a cada sequência de ativação, ocorrem duas passadas pelo *heap*.

- Na primeira passada:
  - Percorre-se todos os ponteiros ao heap recursivamente;
  - Marca-se cada bloco acessado desta forma com um bit de validação, indicando que este é "alcançável".
- Na segunda passada:
  - Percorre-se o *heap* linearmente, liberando a memória de todos os blocos que não foram marcados como sendo alcançáveis.
  - O novo registro de ativação é armazenado no primeiro bloco de memória a satisfazer seu requerimento de memória, **se este houver**.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

Nos casos em que o requerimento de memória não é satisfeito, ocorre compactação.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

São contrapontos ao uso de coleta de lixo:

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

São contrapontos ao uso de coleta de lixo:

1. Requer memória adicional para marcar blocos alcançáveis.

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

São contrapontos ao uso de coleta de lixo:

1. Requer memória adicional para marcar blocos alcançáveis.
2. Requer pelo menos duas passadas pela memória *heap* a cada sequência de ativação

Ambientes de Execução Totalmente Dinâmicos

Marcar e correr (*mark and sweep*)

São contrapontos ao uso de coleta de lixo:

1. Requer memória adicional para marcar blocos alcançáveis.
2. Requer pelo menos duas passadas pela memória *heap* a cada sequência de ativação

Mas existem otimizações que mitigam estas limitações.

Parar e copiar. Nesta, o heap é dividido em duas regiões de tamanho variável:

Parar e copiar. Nesta, o heap é dividido em duas regiões de tamanho variável: a primeira possui blocos ocupados e a segunda um bloco contíguo de memória livre.

Parar e copiar. Nesta, o heap é dividido em duas regiões de tamanho variável: a primeira possui blocos ocupados e a segunda um bloco contíguo de memória livre. A cada sequência de ativação,

- O novo registro de ativação é alocado no endereço do bloco livre;

Parar e copiar. Nesta, o heap é dividido em duas regiões de tamanho variável: a primeira possui blocos ocupados e a segunda um bloco contíguo de memória livre. A cada sequência de ativação,

- O novo registro de ativação é alocado no endereço do bloco livre;
- Realiza-se uma única passada

Parar e copiar. Nesta, o heap é dividido em duas regiões de tamanho variável: a primeira possui blocos ocupados e a segunda um bloco contíguo de memória livre. A cada sequência de ativação,

- O novo registro de ativação é alocado no endereço do bloco livre;
- Realiza-se uma única passada
- Trocam-se os papéis

Coleta de lixo generativa. Nesta, uma terceira região de memória é adicionada ao heap para dados "permanentes".

Considerações finais

Abarcamos os três principais ambientes de execução que fundamentam a uma variedade de linguagens programação.

Considerações finais

Abarcamos os três principais ambientes de execução que fundamentam a uma variedade de linguagens programação. Cada qual destaca a adaptabilidade das linguagens de programação às diferentes necessidades computacionais.

Considerações finais

Abarcamos os três principais ambientes de execução que fundamentam a uma variedade de linguagens programação. Cada qual destaca a adaptabilidade das linguagens de programação às diferentes necessidades computacionais.

Obrigado pela audiência! 👋