Linguagens de Programação

Localização de Variáveis

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Variáveis com ativações específicas
- Registros de ativação
  - Alocação estática
  - Alocação dinâmica com pilha
  - Funções aninhadas
  - Funções como parâmetros
  - Funções de vida longa

INTRODUÇÃO

Linguagens de Programação

Introdução

- Em linguagens imperativas há uma conexão óbvia entre variáveis e sua localização em memória

```
#include <stdio.h>
void main() {
  int x = 3;
  printf("%d (%p)\n", x, &x );
}
```

Onde a variável `x` desse exemplo foi alocada?

Dica: linguagens como C ou C++ permitem obter explicitamente o endereço de uma variável usando o operador `&`

- Já em linguagens sem atribuições, em linguagens funcionais como SML, a localização de variáveis é escondida

```
- fun f y = y + 3;
val f = fn : int -> int
```

Onde está a variável `y` desse exemplo?

VARIÁVEIS COM ATIVAÇÕES ESPECÍFICAS

Linguagens de Programação

Ativação de Funções

- O tempo de vida de uma execução de uma função, de sua chamada até seu retorno correspondente, é chamada de **ativação da função**

- Quando cada ativação tem sua própria amarração de uma variável a uma localização de memória, é uma **variável com ativação específica**

Variáveis com Ativações Específicas

- A maioria das LPs permitem declarar uma variável amarrada a uma localização de memória apenas para uma execução de uma função

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

As variáveis desse exemplo possuem tempo de vida menor

  - Quando essa função é chamada, localizações de memória são necessárias para armazenar `hours`, `minutes` e `seconds`
  - Chamadas posteriores a essa função terão valores diferentes para essas variáveis

A alocação dessas variáveis será feita na mesma localização na próxima chamada?

Ativação de Blocos

- Um bloco de código tem sua própria ativação (**ativação do bloco**), ou seja, tem seu próprio tempo de vida

```
fun fact n =
  if n = 0 then 1
  else let val b = fact (n-1) in n * b end;
```

  - Uma variável pode ser específica de uma ativação particular de um bloco, como a variável `b` do bloco `let` desse exemplo
  - Um bloco pode não ser atingido em uma função; o sistema de linguagens pode amarrar uma variável a uma localização de memória apenas se esse bloco for executado

Outros Tempos de Vida

- A maioria das LPs imperativas permitem declarar uma variável amarrada a uma localização de memória durante toda sua execução

```
int count = 0;
int next_count() {
   count = count + 1;
   return count;
}
```

Em C basta declarar uma variável fora de uma função; essas são chamadas de **variáveis estáticas**

  - A solução de amarração óbvia é a **alocação estática** (normalmente feita pelo carregador)

Escopo vs. Tempo de Vida

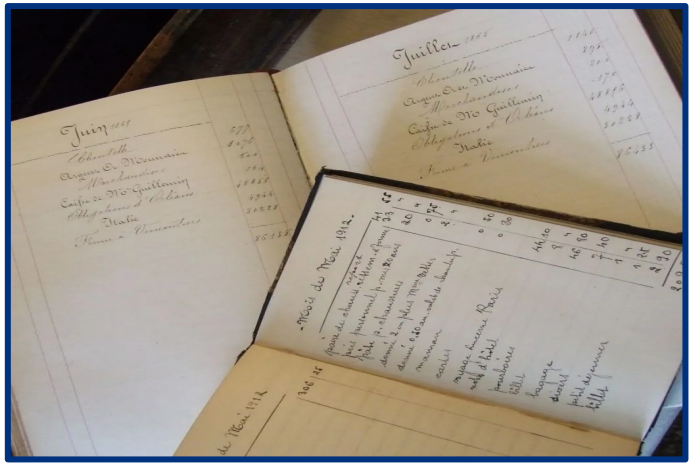
- Algumas linguagens como C permitem a definição de variáveis estáticas com diferentes tipos de escopo

```
int next_count() {
   static int count = 0;
   count = count + 1;
   return count;
}
```

Cuidado: uma variável com escopo local não necessariamente tem um tempo de vida com ativação específica

Como é o tempo de vida de variáveis em linguagens orientadas a objetos?

REGISTROS DE ATIVAÇÃO



Registros de Ativação

- Uma função pode ter muitas diferentes ativações durante a execução de um programa, já que ela pode ser chamada várias vezes

- Mesmo em programas comuns, de apenas uma linha de execução (*single-threaded*), pode-se ter várias ativações simultâneas
  - Basta que uma função chame outra: a ativação da função chamada começa antes de terminar a ativação da função chamadora
  - Elas não estão executando ao mesmo tempo: a ativação da chamadora é suspensa até que a ativação da chamada retorne; após esse retorno, resume-se a ativação da chamadora e espera-se que variáveis com ativação específica tenham sido preservadas

Registros de Ativação

- O sistema da linguagem usualmente aloca todas as variáveis com ativação específica de uma função em um **registro de ativação**

- O registro também contém outros dados de ativação específicos, como
  - Endereços de retorno: para onde o programa deve ir quando essa ativação retornar
  - Ligação para o registro de ativação do chamador: restaurar a memória para resumir a execução da função chamadora

Registros de Ativação de Blocos

- Quando se entra em um bloco em uma função, a localização de memória para as variáveis desse bloco precisam ser encontradas

- Sistemas de linguagem podem implementar essa funcionalidade de várias maneiras
  - Pré-alocar espaço para as variáveis do bloco no registro de ativação da função que o contém
  - Estender o registro de ativação da função quando se entra no bloco e revertê-lo ao sair do bloco
  - Alocar registros de ativação separados para os blocos

Qual dessas estratégias são comumente utilizada?

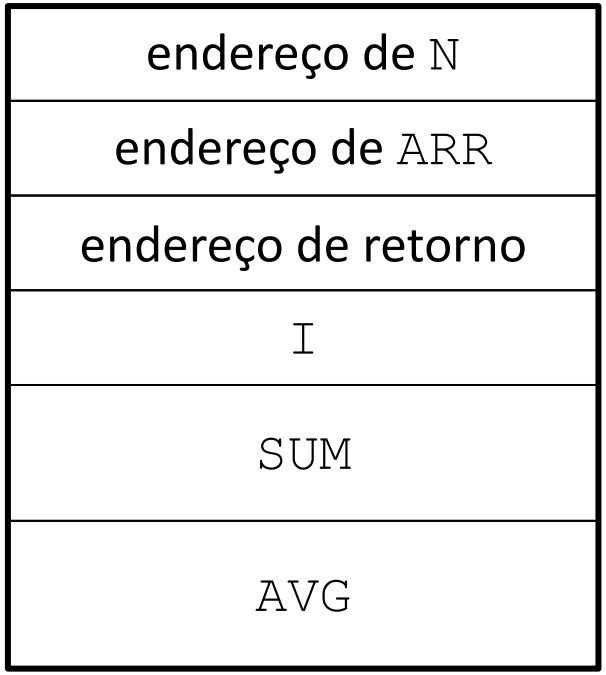
REGISTROS DE ATIVAÇÃO > ALOCAÇÃO ESTÁTICA

Alocação Estática de Registros de Ativação

- A abordagem mais simples: alocar um registro de ativação para cada função estaticamente
  - Ou seja, um registro para cada função antes do programa começar a executar

- Antigos dialetos de FORTRAN e COBOL usam esse sistema

```
 FUNCTION AVG (ARR, N)
      DIMENSION ARR(N)
      SUM = 0.0
      DO 100 I = 1, N
        SUM = SUM + ARR(I)
100   CONTINUE
      AVG = SUM / FLOAT(N)
      RETURN
      END
```



Qual a vantagem de usar esse sistema? E a desvantagem?

REGISTROS DE ATIVAÇÃO > ALOCAÇÃO DINÂMICA COM PILHA

Linguagens de Programação

Pilha Dinâmica de Registros de Ativação

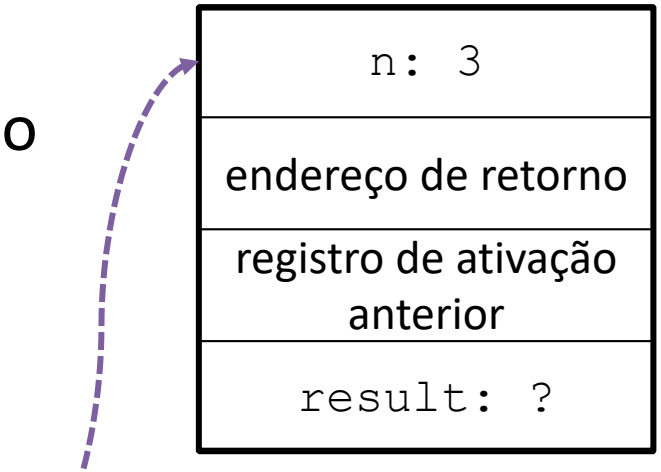
- Linguagens que suportam recursão precisam alocar um novo registro de ativação para cada ativação de função
  - Isso significa que os registros de ativação precisam ser alocados dinamicamente
  - Quando uma função é chamada, o sistema da linguagem prepara para executá-lo alocando um novo registro de ativação para ela
- Muitas linguagens, como C, os registros podem ser desalocados quando a função retorna
- Os registros de ativação formam uma pilha na execução
  - Registros são empilhados na chamada e desempilhados no retorno

Por essa razão, registros de ativação também são conhecidos como *stack frames*

Como a função sabe onde está o endereço de seu registro de ativação?

Pilha Dinâmica de Registros de Ativação

- Considere a implementação em C da função `fact` recursiva que calcula o fatorial de um número inteiro



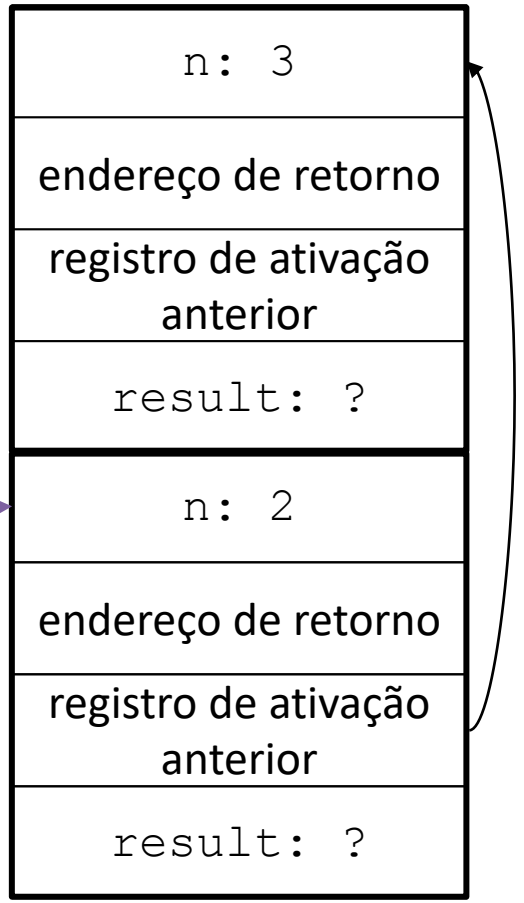
Registro de ativação atual

```
int fact(int n) {
  int result;
  if (n <= 1) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

Como os registros de ativação são criados para a chamada `fact(3)`?

Pilha Dinâmica de Registros de Ativação

- Considere a implementação em C da função `fact` recursiva que calcula o fatorial de um número inteiro

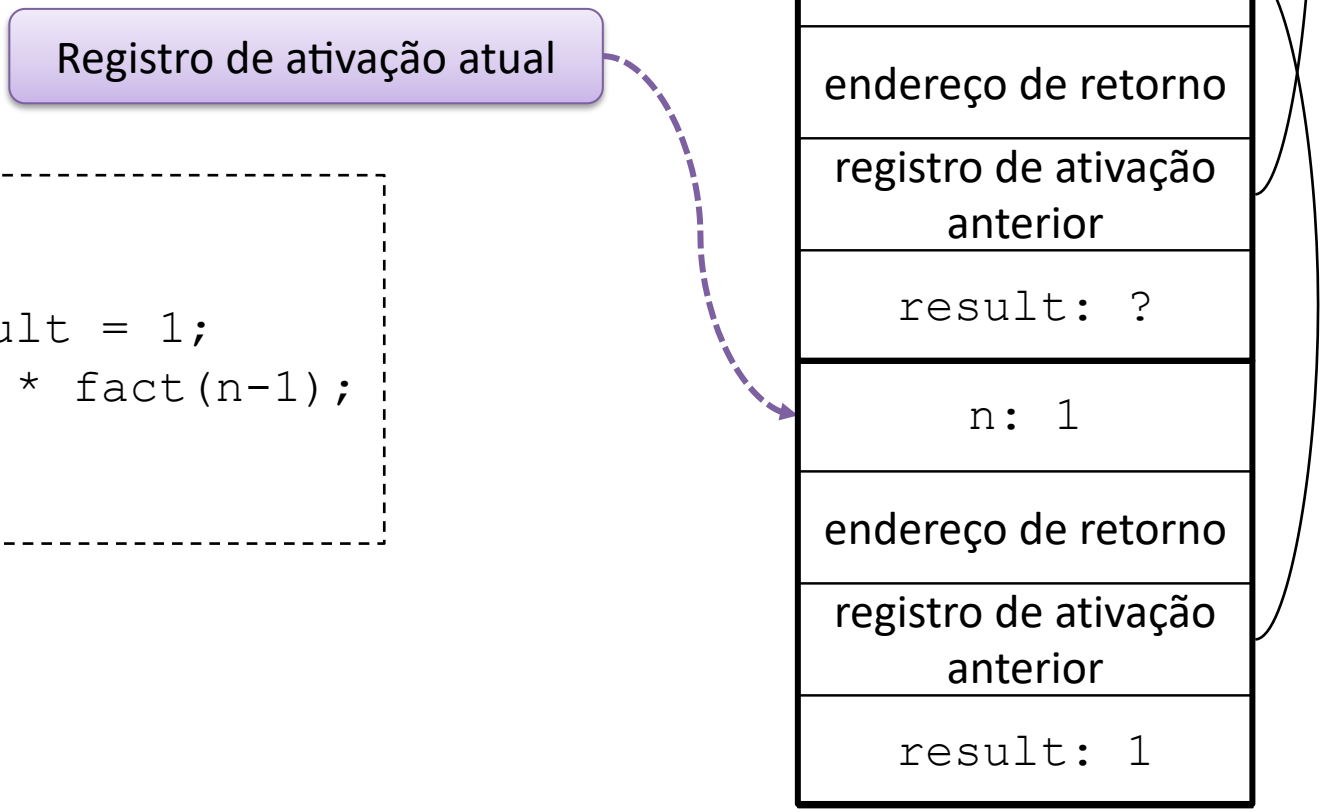



```
int fact(int n) {
  int result;
  if (n <= 1) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

Pilha Dinâmica de Registros de Ativação

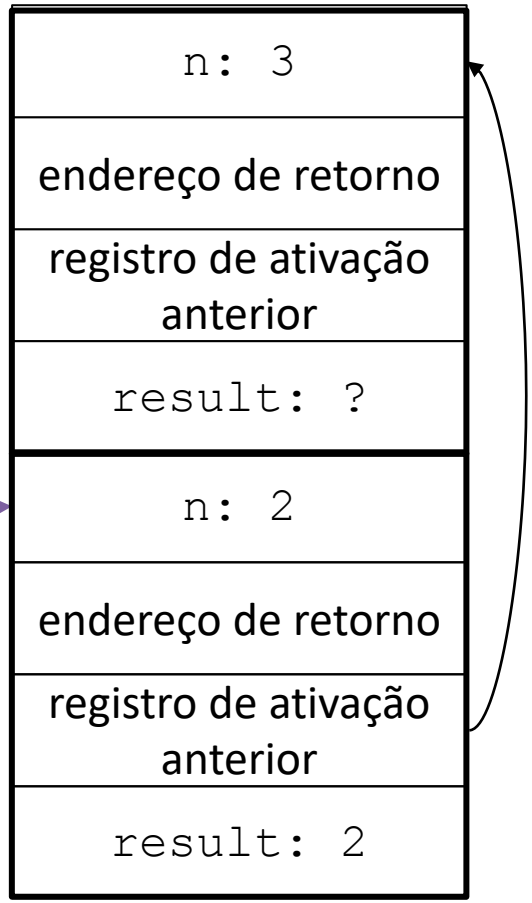
- Considere a implementação em C da função `fact` recursiva que calcula o fatorial de um número inteiro

```
int fact(int n) {
  int result;
  if (n <= 1) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Pilha Dinâmica de Registros de Ativação

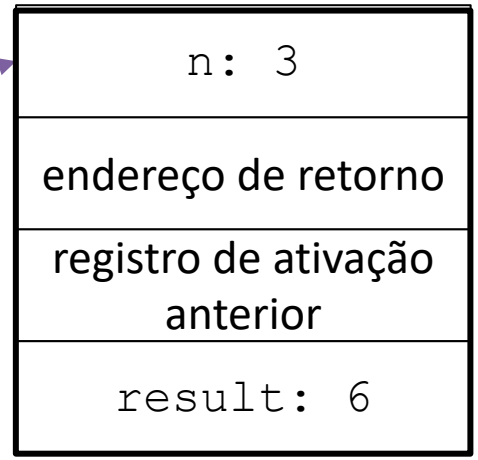
- Considere a implementação em C da função `fact` recursiva que calcula o fatorial de um número inteiro




```
int fact(int n) {
  int result;
  if (n <= 1) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

Pilha Dinâmica de Registros de Ativação

- Considere a implementação em C da função `fact` recursiva que calcula o fatorial de um número inteiro

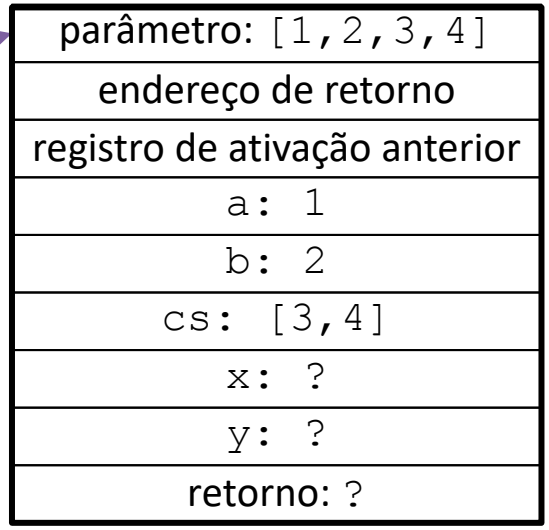


Registro de ativação atual

```
int fact(int n) {
  int result;
  if (n <= 1) result = 1;
  else result = n * fact(n-1);
  return result;
}
```

Pilha Dinâmica de Registros de Ativação

- Considere a implementação em SML da função `halve` recursiva que divide uma lista em duas partes



Registro de ativação atual

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

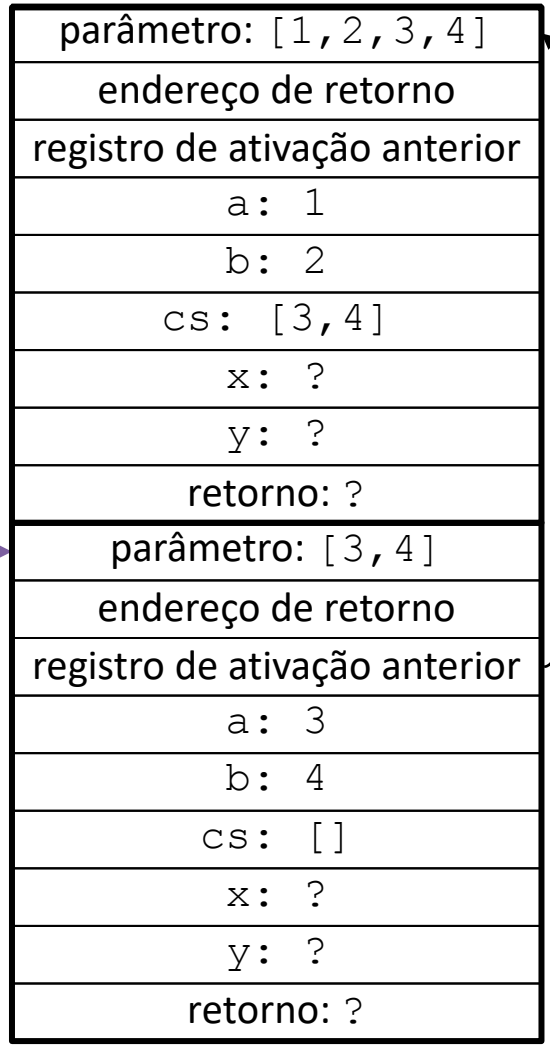
Como os registros de ativação são criados para a chamada `halve([1,2,3,4])`?

Pilha Dinâmica de Registros de Ativação

- Considere a implementação em SML da função `halve` recursiva que divide uma lista em duas partes

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

Registro de ativação atual



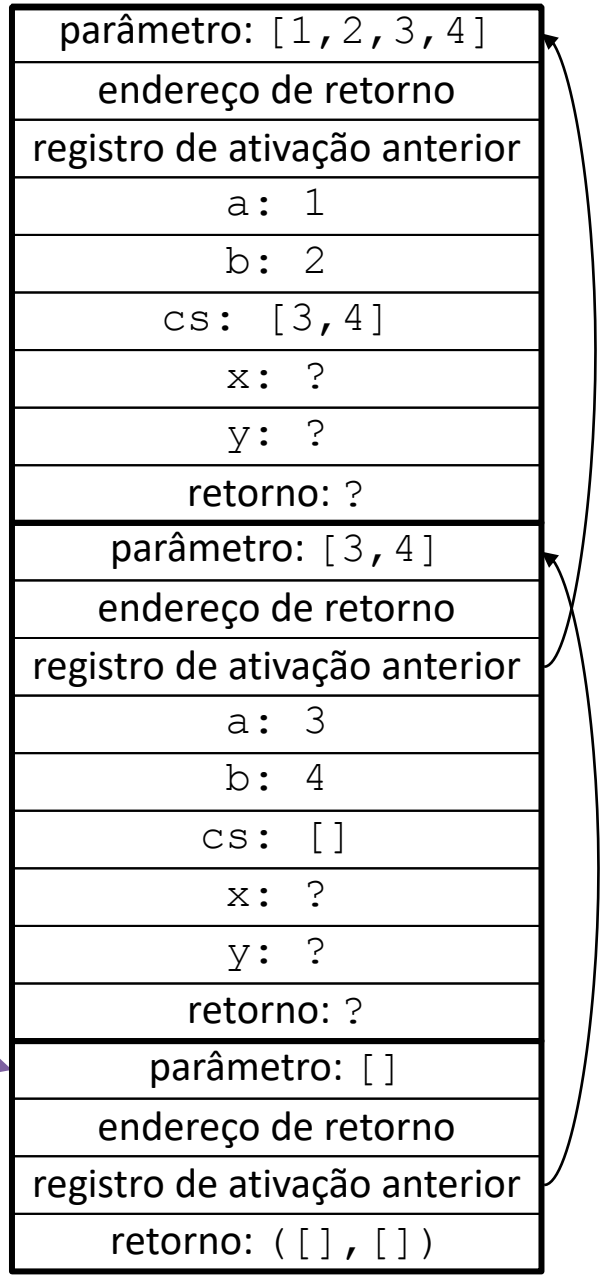
Pilha Dinâmica de Registros de Ativação

- Considere a implementação em SML da função `halve` recursiva que divide uma lista em duas partes

Registro de ativação atual

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

Repare que neste caso o bloco não foi executado, portanto não foi preciso alocar suas variáveis

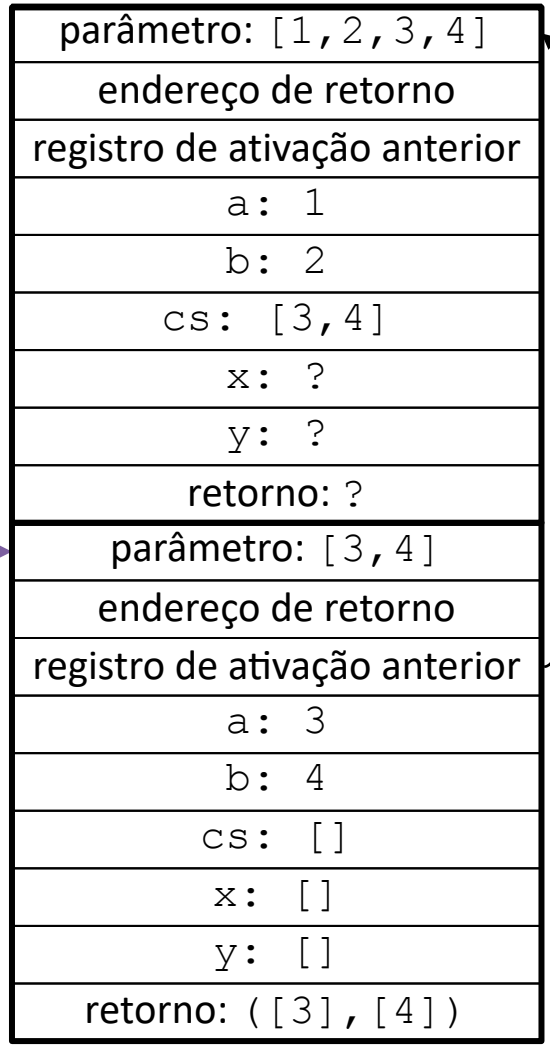


Pilha Dinâmica de Registros de Ativação

- Considere a implementação em SML da função `halve` recursiva que divide uma lista em duas partes

Registro de ativação atual

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```



Pilha Dinâmica de Registros de Ativação

- Considere a implementação em SML da função `halve` recursiva que divide uma lista em duas partes

parâmetro: `[1,2,3,4]`
endereço de retorno
registro de ativação anterior
`a: 1`
`b: 2`
`cs: [3,4]`
`x: [3]`
`y: [4]`
retorno: `([1,3],[2,4])`

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

REGISTROS DE ATIVAÇÃO > FUNÇÕES ANINHADAS

Linguagens de Programação

Funções Aninhadas

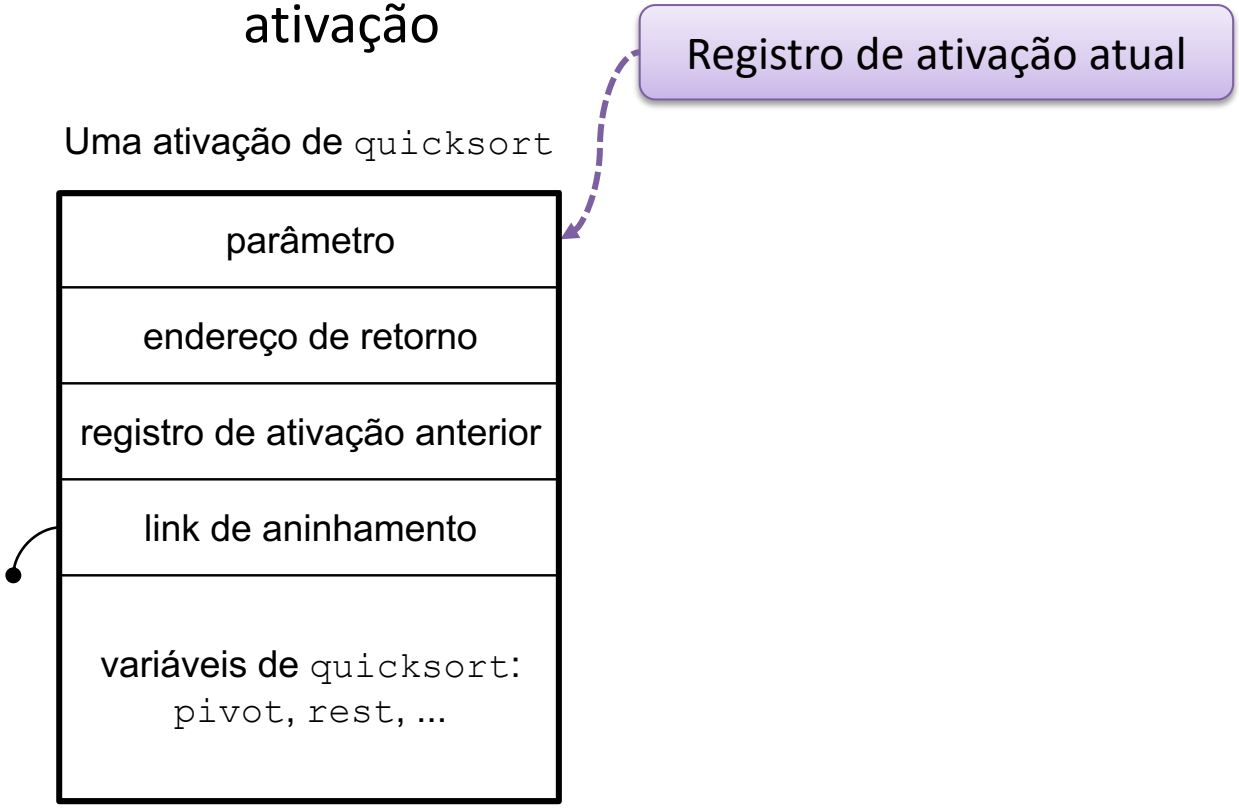
- Algumas linguagens como Pascal, Ada e ML permitem funções aninhadas com referências a variáveis não locais
  - Elas não são locais e nem estáticas, são referências a variáveis que estão em outros registros de ativação

```
fun quicksort nil = nil
|   quicksort ( pivot :: rest) =
  let
    fun split nil = (nil, nil)
    |   split (x::xs) =
      let
        val (below, above) = split(xs)
      in
        if x < pivot then (x::below, above)
        else (below, x::above)
      end;
    val (below, above) = split(rest)
  in
    quicksort below @ [pivot] @ quicksort above
  end;
```

Como lidar com isso?

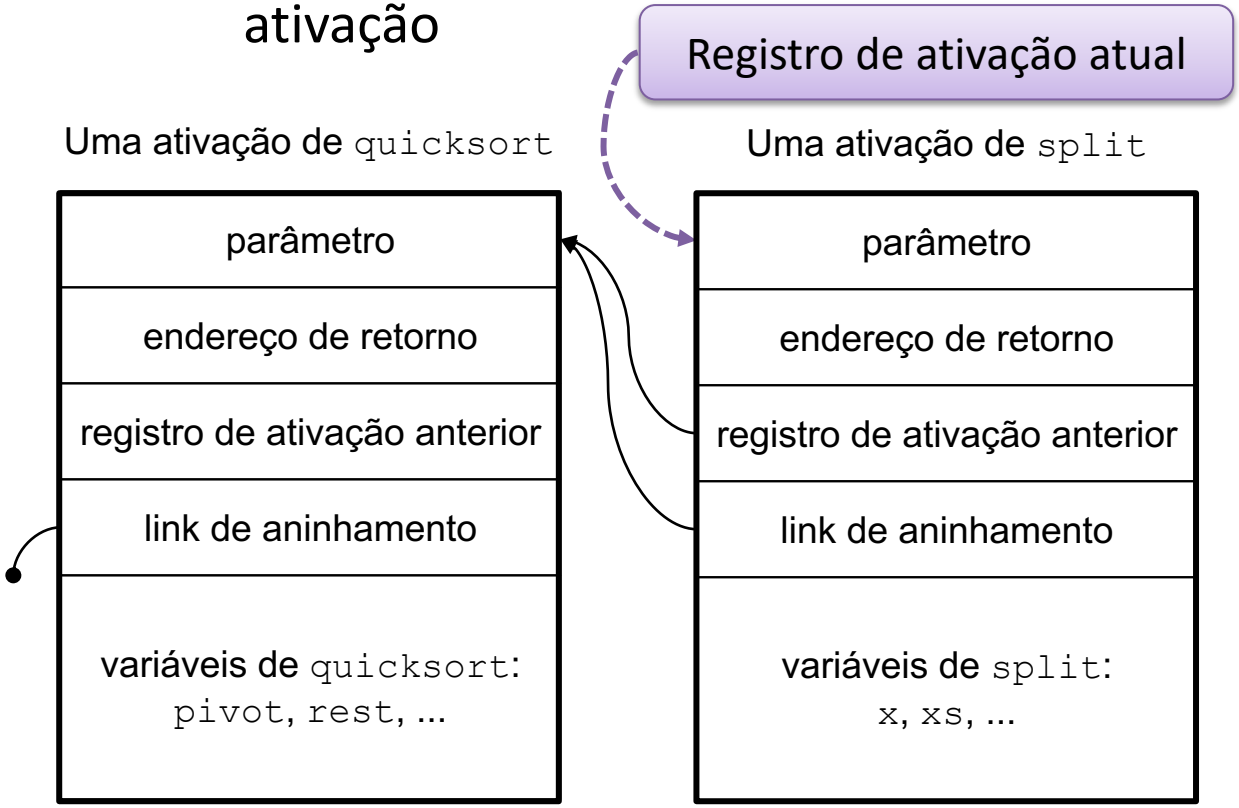
Links de Aninhamento

- Uma função interna precisa ser capaz de achar o endereço do registro de ativação mais recente da função externa
  - Isso é feito através de um link de aninhamento no registro de ativação



Links de Aninhamento

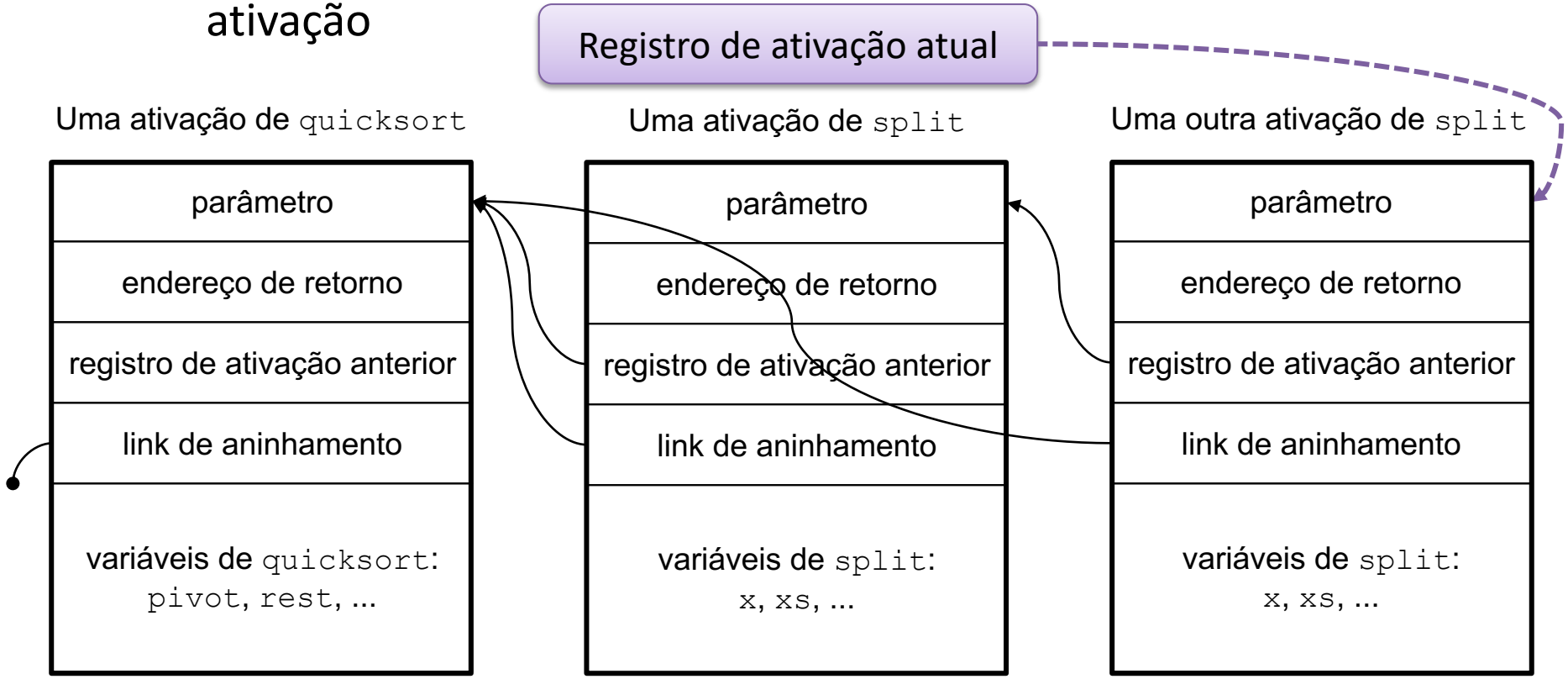
- Uma função interna precisa ser capaz de achar o endereço do registro de ativação mais recente da função externa
  - isso é feito através de um link de aninhamento no registro de ativação



Links de Aninhamento

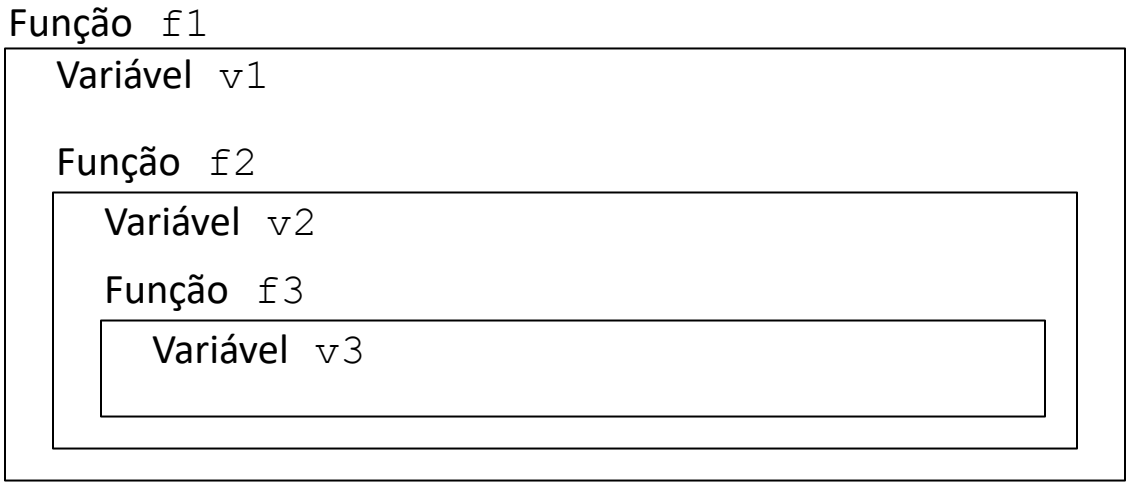
E se houver múltiplos níveis de aninhamento?

- Uma função interna precisa ser capaz de achar o endereço do registro de ativação mais recente da função externa
  - isso é feito através de um link de aninhamento no registro de ativação



Múltiplos Níveis de Aninhamento

- Considere o seguinte diagrama que mostra 3 níveis de aninhamento com suas respectivas variáveis em cada nível

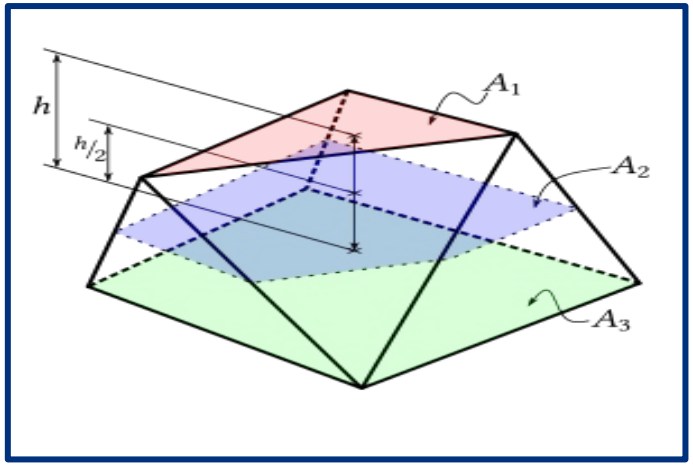


  - Variáveis no mesmo nível são resolvidas no registro de ativação da própria função: `f1` para `v1`, `f2` para `v2` e `f3` para `v3`
  - Referências que estão a `n` níveis de distância são obtidas usando `n` links de aninhamento
    - `f2` para `v1` usa o link 1 vez, enquanto `f3` para `v1` usa 2 vezes

Outras soluções

- Links de endereçamento não é a única forma de implementar a ideia de funções internas referenciando variáveis externas em outras funções
  - *Displays*: links de endereçamentos não são mantidos no registro de ativação, mas coletados em um único arranjo estático
  - *Lambda lifting*: referências internas são substituídas por referências passadas como parâmetros (ocultos)

REGISTROS DE ATIVAÇÃO > FUNÇÕES COMO PARÂMETROS



Funções como Parâmetros

- Em linguagens de baixo nível como C não permitem que funções sejam passadas como parâmetros, somente o endereço delas

```
void quicksort(void* list[], int size, int (*cmp)(void*, void*));
```

  - Nesse exemplo, o endereço de uma função (e não a função em si) é passado como parâmetro; essa função é utilizada para comparar dois valores para a ordenação (menor que, maior ou igual a, ...)
    - Nesse caso, para chamar a função `cmp` é preciso primeiramente dereferenciá-la: `(*cmp)(i1, i2)`

E como linguagens como SML lidam com isso?

Funções como Parâmetros

- Em SML não basta passar a implementação da função como parâmetro

```
fun addXToAll (x, theList) =
  let
    fun addX y = y + x
  in
    map addX theList
  end;
```

Cuidado: o link de endereçamento do chamador não está correto, já que a função `addX` não está aninhada em `map`

Como lidar com essa situação?

Funções como Parâmetros

- Para passar uma função como parâmetro em SML, deve-se passar tanto a sua implementação como seu link de endereçamento
  - O sistema de linguagens irá manter essas informações juntas

- Uma declaração de função nada mais é que uma declaração de uma variável com a implementação dessa função na forma `fn`

```
fun addXToAll (x, theList) =
  let
    fun addX y = y + x
  in
    map addX theList
  end;
```

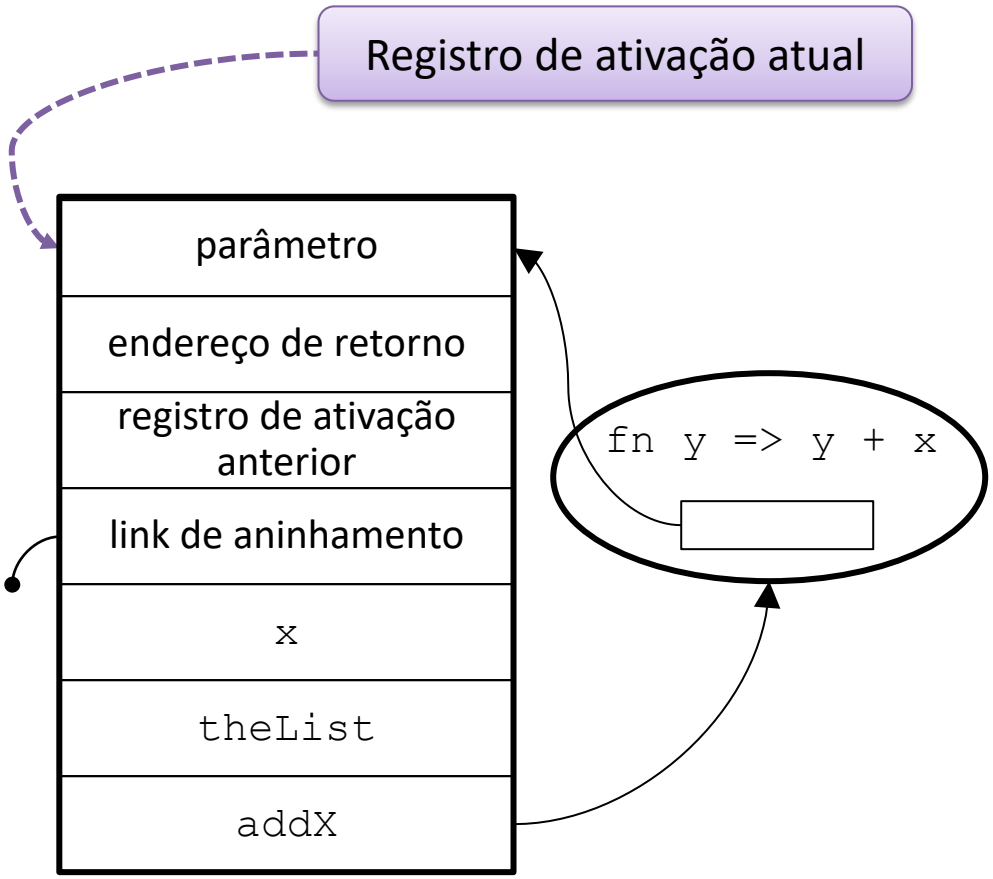
```
fun addXToAll (x, theList) =
  let
    val addX = fn y => y + x
  in
    map addX theList
  end;
```

Dica: funções como valores aparecem em muitos lugares em SML: parâmetros, valores de expressões ou variáveis, retorno de funções, etc.

Funções como Parâmetros

- Exemplo de como o link de endereçamento fica associado à implementação da função

```
fun addXToAll (x, theList) =
  let
    fun addX y = y + x
  in
    map addX theList
  end;
```



REGISTROS DE ATIVAÇÃO > FUNÇÕES DE VIDA LONGA

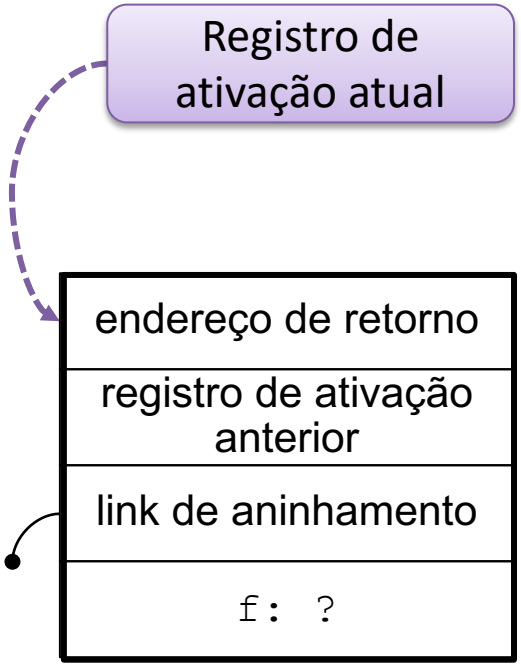
Linguagens de Programação

Funções de Vida Longa

- Algumas linguagens permitem que valores de funções persistam após a função que a criou tenha retornado

```
fun funToAddX x =
  let
   fun addX y = y + x
  in
    addX
  end;

fun test () =
  let
    val f = funToAddX 3;
  in
    f 5
  end;
```



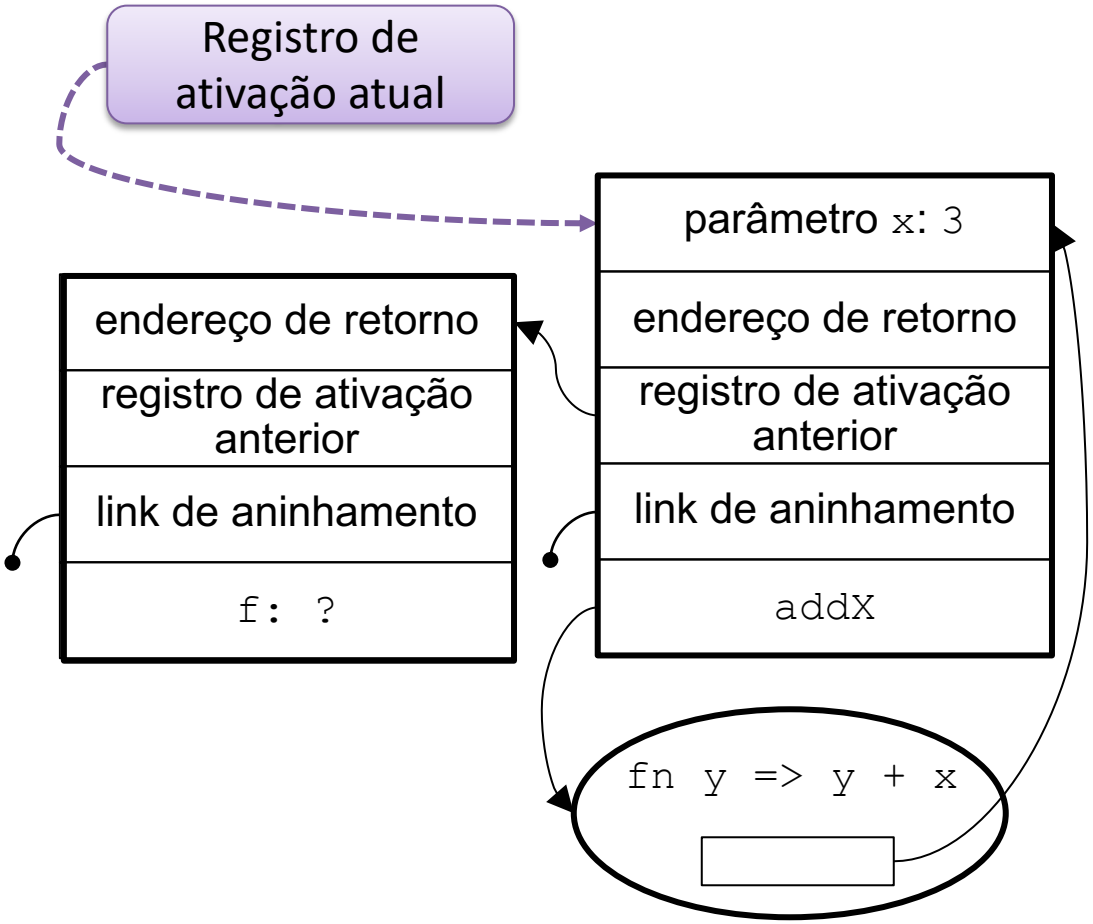
Dica: use o parâmetro `()`, o único de tipo `unit` de SML, para passar um valor *dummy* para uma função

Funções de Vida Longa

- Algumas linguagens permitem que valores de funções persistam após a função que a criou tenha retornado

```
fun funToAddX x =
  let
   fun addX y = y + x
  in
    addX
  end;

fun test () =
  let
    val f = funToAddX 3;
  in
    f 5
  end;
```

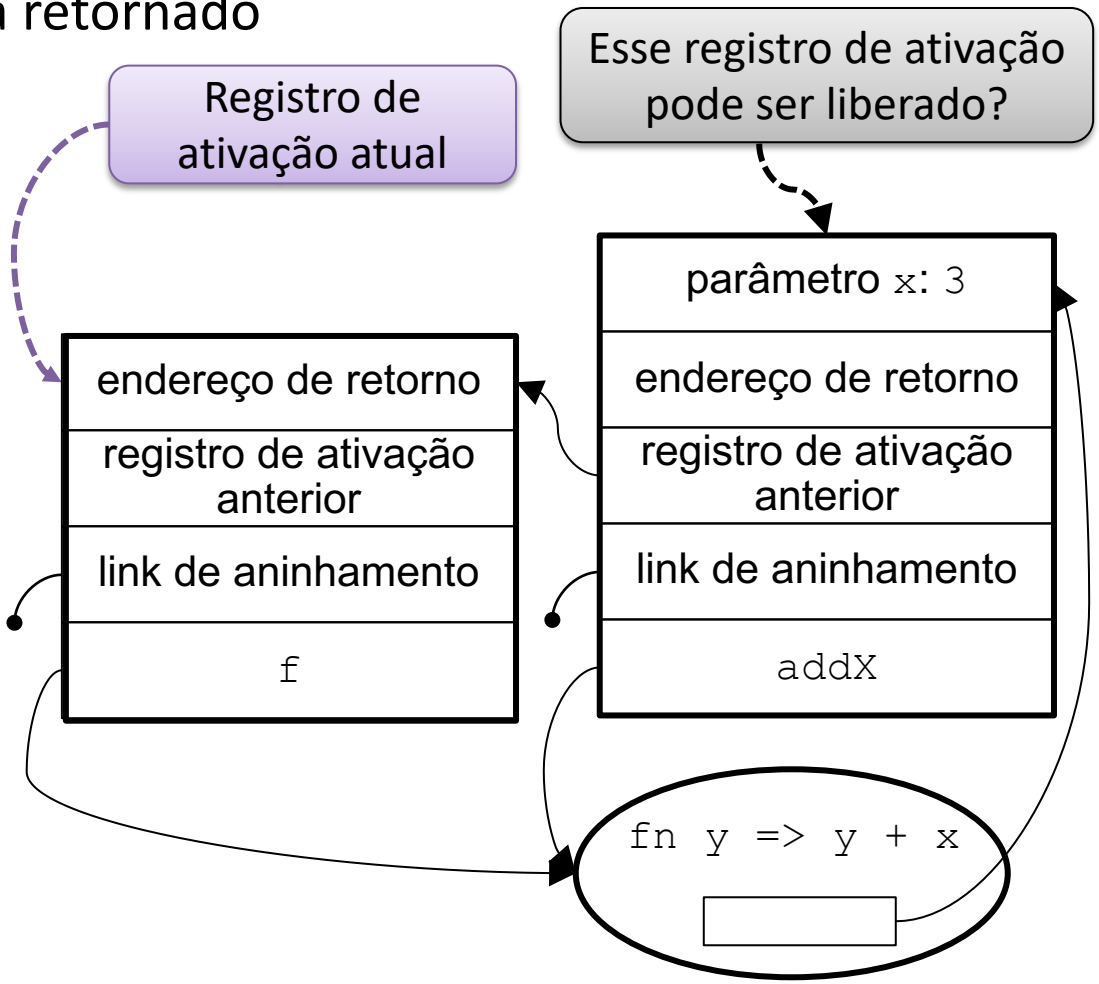


Funções de Vida Longa

- Algumas linguagens permitem que valores de funções persistam após a função que a criou tenha retornado

```
fun funToAddX x =
  let
   fun addX y = y + x
  in
    addX
  end;

fun test () =
  let
    val f = funToAddX 3;
  in
    f 5
  end;
```



Funções de Vida Longa

- O sistema de linguagens de SML, e de outras linguagens com problemas similares, nem sempre podem alocar e desalocar registros de ativação na ordem do empilhamento

- Às vezes, um registro de ativação precisa ser mantido mesmo após uma função retornar
  - Ele não pode ser desalocado enquanto houver links de aninhamento para ele

Quando desalocar esse registro de ativação então?

ISSO É TUDO, PESSOAL!