Linguagens de Programação

Escopo

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Escopo com blocos
- Escopo com espaço de nomes rotulados
- Escopo com espaço de nomes primitivo
- Tempo de vida
- Ambiente de referenciamento
- Escopo dinâmico
- Compilação separada

INTRODUÇÃO

Linguagens de Programação

Introdução

- Escopo é trivial quando se usam nomes únicos para tudo

```
fun square a = a * a;
fun double b = b + b;
```

- Porém, em linguagens modernas, usa-se constantemente o mesmo nome repetidamente

```
fun square n = n * n;
fun double n = n + n;
```

Como linguagens lidam com isto?

Definições

- Decidir qual definição deveria ser aplicada a alguma instância de um nome é um problema de **amarração**
  - Quando existem diferentes variáveis com o mesmo nome, existem diferentes possíveis amarrações para esse nome

- A questão da amarração não ocorre somente com **variáveis**, mas também com outras construções de linguagens associadas a nomes, como **funções**, **tipos**, **constantes**, **classes**, etc

- Uma **definição** é qualquer coisa que estabelece uma possível amarração para um nome

Definições

- Em ML, a seguinte função possui duas definições

```
fun square n = n * n;
```

  - A definição de `square` como nome desta função
  - A definição de `n` como o nome do parâmetro formal desta função

- Exemplo em Pascal com múltiplas definições

```
const
  Low = 1;
  High = 10;
type
  Ints = array [Low..High]
           of Integer;
var
  x: Ints;
```

  - Os nomes `Low` e `High` para as constantes `1` e `10`
  - O nome `Ints` para o novo tipo de arranjo
  - O nome `x` para a variável do tipo `Ints`

Repare que estes mesmos nomes poderiam ser reutilizados em qualquer outro lugar do programa

Escopo

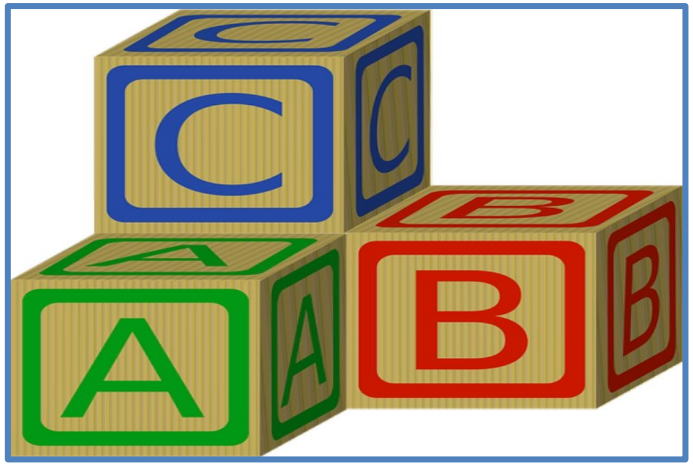
- Uma ocorrência de um nome **está no escopo** de uma determinada definição daquele nome sempre que esta definição governe a amarração para esta ocorrência

```
- fun square square = square * square;
val square = fn : int -> int
- square 3;
val it = 9 : int
```

  - As duas ocorrências de **square** no corpo da função **estão no escopo** da definição do nome do parâmetro formal **square**
  - Já a ocorrência de **square** na chamada **está no escopo** da definição do nome da função **square**

Como ML decide isto?

ESCOPO COM BLOCOS



Escopo com Blocos

- Um bloco é qualquer construção de linguagem que contém definições e também contém a região do programa em que estas definições se aplicam

```
let
   val x = 1
   val y = 2
in
   x + y
end
```

  - Esta construção `let` possui duas definições (`x` e `y`) e também a região em que estas definições se aplicam (do ponto da definição até o `end`)

A construção `let` nada mais é que um bloco sem nenhum outro propósito

Escopo com Blocos

- Outras construções de ML possuem outros propósitos, mas também servem como blocos
  - `fun` define uma função, mas também inclui um bloco

```
fun cube x = x * x * x;
```

  - `fun` com múltiplas alternativas possui múltiplos blocos

```
fun f (a::b::_) = a + b
|   f [a] = a
|   f [] = 0;
```

  - Cada regra em uma cláusula *match* é um bloco

```
case x of (a,0) => a | (_,b) => b;
```

Escopo com Blocos

- Em C e em linguagens derivadas, como C++ e Java, pode-se combinar vários comandos em um comando composto usando `{` e `}`

```
while (i < 0) {
    int c = i * i * i;
    p += c;
    q += c;
    i -= step;
}
```

A variável `c` declarada dentro `while` pode ser usada depois dele?

O comando composto, entre `{` e `}`, também cria um bloco

Blocos Aninhados

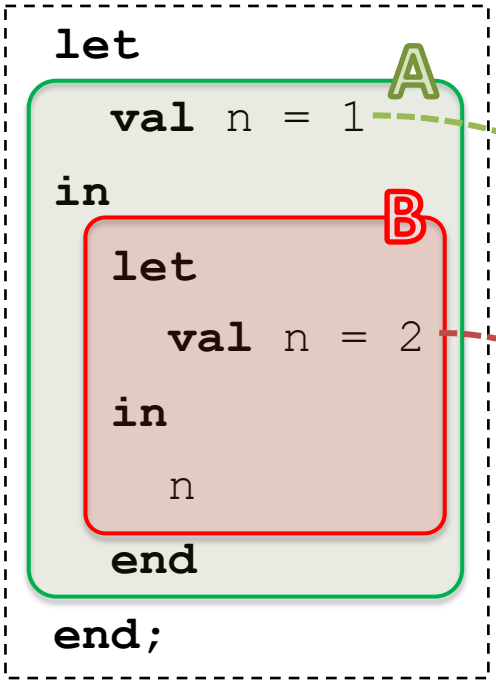
- Blocos aninhados são blocos definidos dentro de outros blocos

```
let
  val x = 1
in
  let
    val y = 2
  in
    x + y
  end
end;
```

O que acontece quando um bloco contém outro bloco e ambos possuem definição de um mesmo nome?

Blocos Aninhados

- O escopo de uma definição é o bloco que contém esta definição, do ponto desta definição até o final do bloco, menos o escopo de qualquer redefinição do mesmo nome em blocos internos



O escopo da definição
`val n = 1` é **A – B**

O escopo da definição
`val n = 2` é **B**

ML usa esta regra, assim como a maioria das linguagens com escopo estático

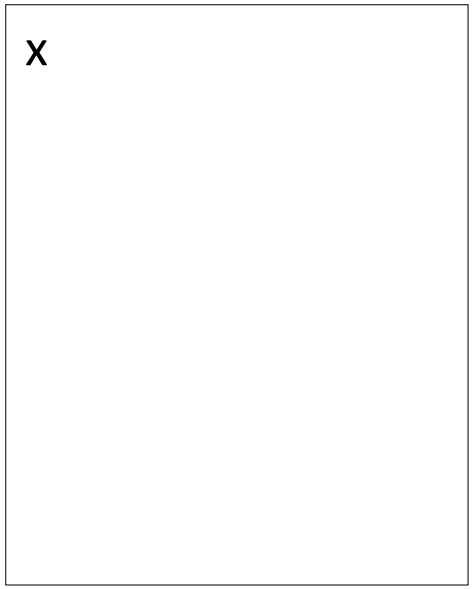
Blocos Aninhados

- A redefinição de uma variável com o mesmo nome em um escopo interior de uma definição já existente em um escopo exterior, permite "esconder" (*shadow*) a definição exterior

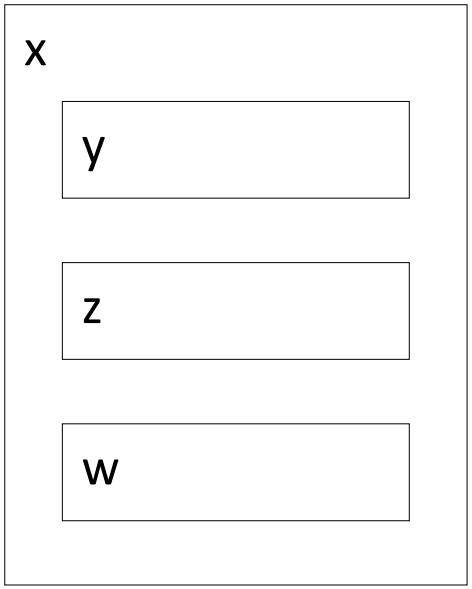
```
#include <stdio.h>

void main() {
    int i = 0;
    int x = 10;
    while (i++ < 100) {
        float x = 3.231;
        printf("x = %f\r\n", x * i);
    }
}
```

Java não permite tal ocultamento em blocos internos

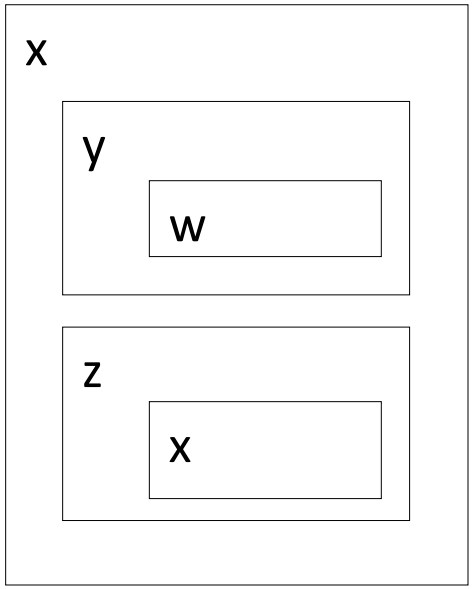
Modelos de Blocos



Bloco Monolítico

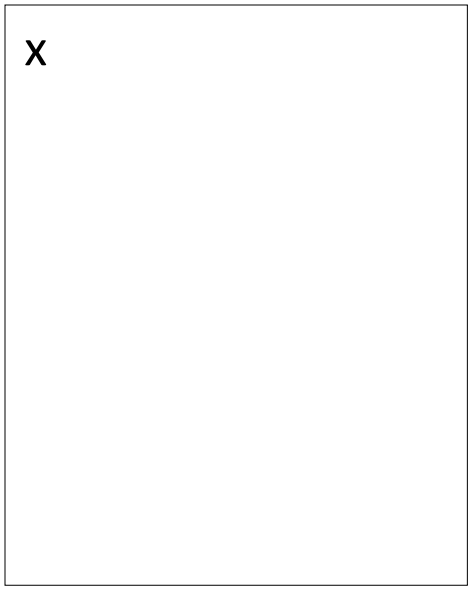


Blocos Não Aninhados



Blocos Aninhados

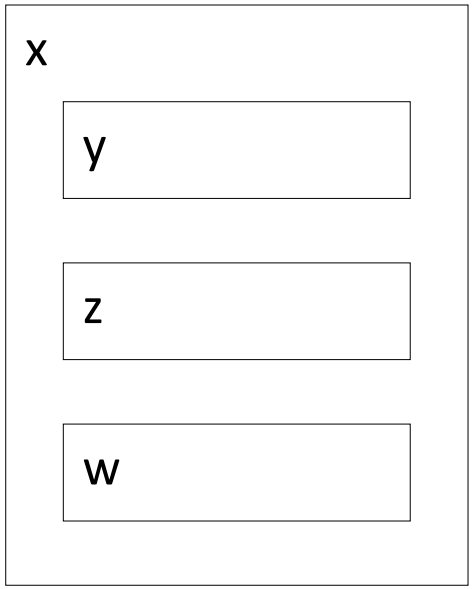
Modelos de Blocos



Bloco Monolítico

- Programa com um único bloco
  - Amarrações com visibilidade global
- Estrutura muito elementar
  - Não apropriada para grandes programas
  - Dificulta a codificação por grandes equipes de programadores
- Exemplo: BASIC e COBOL

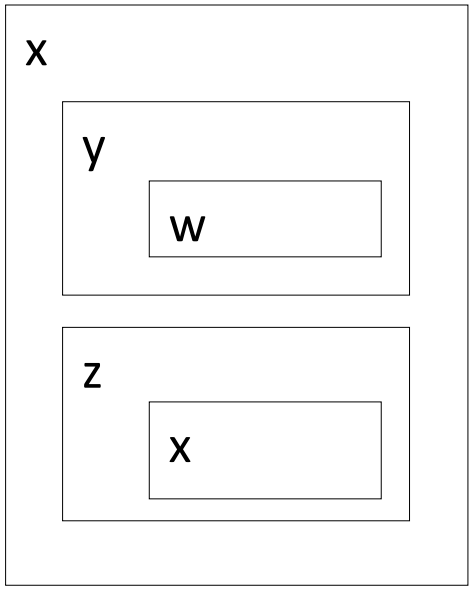
Modelos de Blocos



Blocos Não Aninhados

- Programa dividido em blocos, somente locais ou globais
  - Qualquer variável não-local deve ser global
- Exemplo: em FORTRAN todos os subprogramas são separados e cada um atua como um bloco

Modelos de Blocos

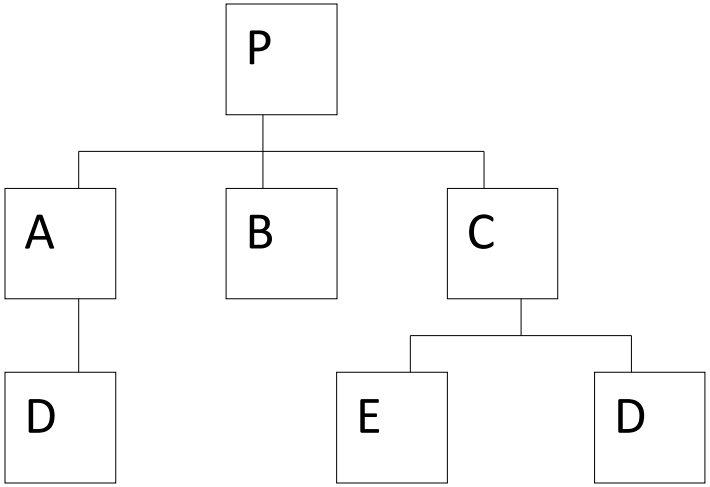


Blocos Aninhados

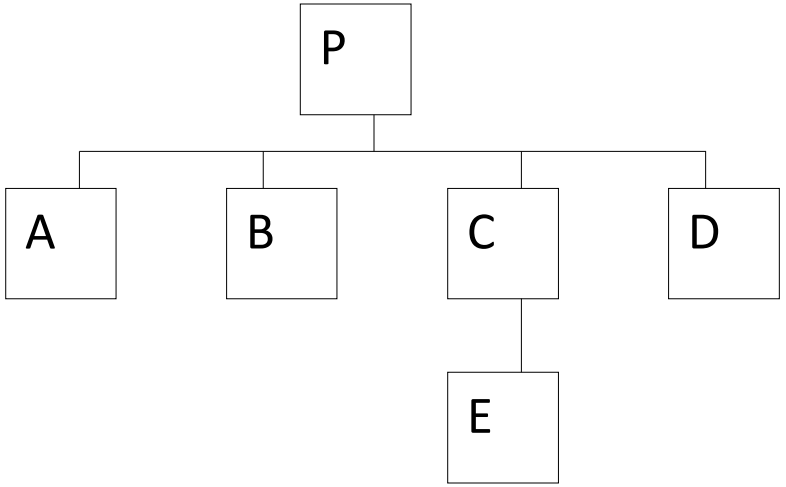
- Blocos podem ser aninhados dentro de outros blocos
- Identificadores podem ser amarrados em cada bloco
  - Identificadores são procurados de "dentro para fora"
- Vantagem: programas mais legíveis
- Desvantagem: ocultamento de variáveis

Modelos de Blocos

- Problemas com estrutura aninhada



Bloco D duplicado



Bloco D acessível por mais blocos que intencionado

Modelos de Blocos

- C utiliza uma abordagem mista
  - Funções são modeladas como **blocos não-aninhados**
    - Funções podem ser reusadas e variáveis globais podem ser compartilhadas (visibilidade global)
  - Blocos internos às funções são modelados como **blocos aninhados**
    - Estes blocos não podem ser reaproveitados em outros lugares

ESCOPO COM ESPAÇO DE NOMES ROTULADOS

Espaço de Nomes Rotulados

- Um espaço de nomes rotulado é qualquer construção de linguagens que **contém definições** e uma região no programa em que estas definições se aplicam, e também **possui um nome** que pode ser usado para acessar estas definições do lado de fora destas construções

```
structure Fred = struct
  val a = 1;
  fun f x = x + a;
end;
```

  - É parecido com um bloco: a definição `a` pode ser usada em qualquer lugar a partir de sua definição até `end`
  - Porém, as definições também estão disponíveis externamente usando o nome da estrutura

```
- Fred.a;
val it = 1 : int
- Fred.f 4;
val it = 5 : int
```

Espaço de Nomes Rotulados

- `structure` em SML tem como único propósito rotular um espaço de nomes; outras linguagens possuem construções similares para este mesmo propósito
  - C++ com `namespace`, Modula-3 com `module`, Ada com `package` e Java com `package`

- Definições de classes servem para outros propósitos, mas também podem ser utilizados como um espaço de nomes rotulados

```
public class Month {
  public static final int MIN = 1;
  public static final int MAX = 12;
  // ...
}
```

  - As variáveis `MIN` e `MAX` são visíveis no restante da classe, mas também acessadas externamente via `Month.MIN` e `Month.MAX`

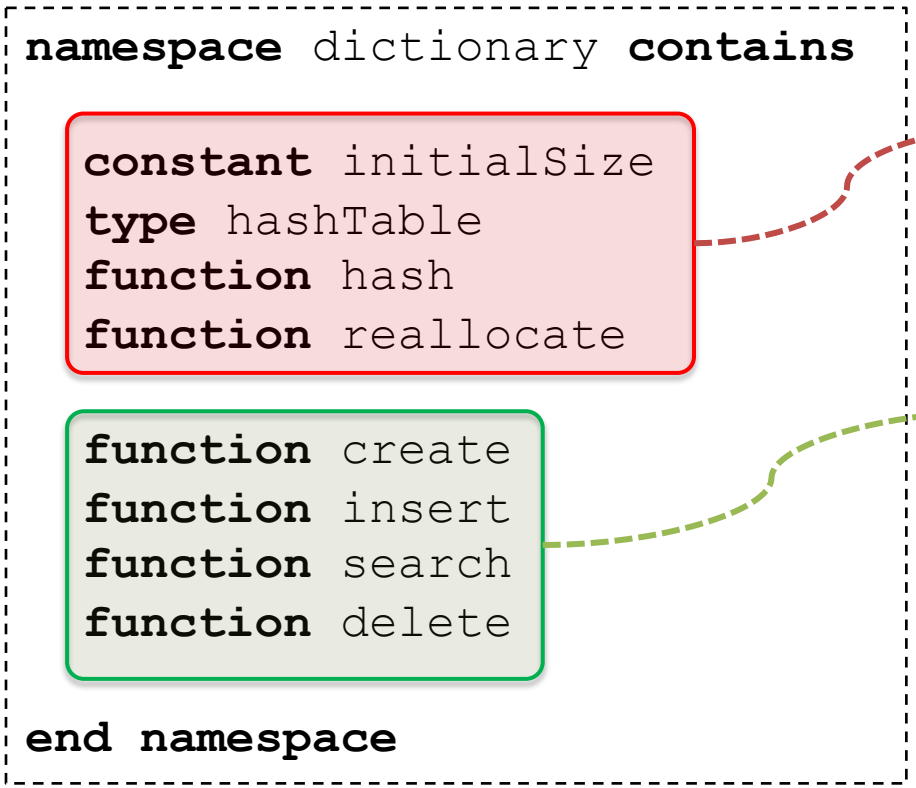
Qual a vantagem de se ter espaço de nomes rotulados?

Espaço de Nomes Rotulados

- Uma vantagem de se ter espaço de nomes rotulados é evitar **poluição no espaço de nomes**
  - Uma definição muito ampla, em um escopo global por exemplo, pode acarretar problemas de nomes únicos
    - Nomes mais memoráveis como `maxSupplierBid` ao invés de somente `max` para evitar conflitos
  - Com espaço de nomes rotulados, pode-se usar `max` em ambos os casos
    - Dentro do espaço de nomes como `max` e do lado de fora como `SupplierBid.max`

Espaço de Nomes Rotulados

- Considere um tipo de dados abstrato para implementação de um dicionário em uma linguagem hipotética



Detalhes de implementação deveriam ser ocultos

Definições da interface deveriam ser visíveis

Como lidar com este problema?

Com Modificadores de Visibilidade

- Em algumas linguagens, como C++, o espaço de nomes especifica a visibilidade de seus componentes

```
namespace dictionary contains
  private:
    constant initialSize
    type hashTable
    function hash
    function reallocate

  public:
    function create
    function insert
    function search
    function delete

end namespace
```

Existe outra abordagem?

Com Interface Separada

- Outras, como em ML, uma construção separada define uma interface para um espaço de nomes

```
interface dictionary contains
  function create
  function insert
  function search
  function delete
end interface
```

```
namespace myDictionary implements dictionary contains
  constant initialSize
  type hashTable
  function hash
  function reallocate
  function create
  function insert
  function search
  function delete
end namespace
```

ML usa esta abordagem com a construção `signature`, já Java e Ada combinam estas duas abordagens

ESCOPO COM ESPAÇO DE NOMES PRIMITIVO

Linguagens de Programação

Espaço de Nomes Primitivo

- Considere o seguinte trecho de código

```
- fun f int = int * int;
val f = fn : int -> int
- f 3;
val it = 9 : int
```

Embora válido, o que tem de estranho neste código?

Espaço de Nomes Primitivo

- A sintaxe de ML mantém tipos e expressões firmemente separados
  - ML sempre sabe quando está se procurando por um tipo (tipicamente após :) ou por qualquer outra coisa

- Existe um espaço de nomes separado para tipos

```
fun f (int : int) = (int : int) * (int : int);
```

  - `int` está no espaço do nomes ordinário
  - `int` está no espaço de nomes para tipos

Por causa desta segregação, ML não faz confusão com estes nomes

TEMPO DE VIDA

Tempo de Vida

- Escopo e tempo de vida possuem uma relação próxima, mas são conceitos **diferentes**
  - Escopo estático é um conceito **textual**
  - Tempo de vida é um conceito **temporal**

Tempo de Vida

```
program P
var m: integer;
   procedure R (n: integer);
   begin
      if n > 0 then R (n-1)
   end;
begin
   R(2)
end.
```

Escopo de n

Escopo de m

Tempo de Vida

- Comparação entre escopo e tempo de vida em C

```
void foo() {
    static int y;
    /* ... */
}

void main() {
    int x;
    foo();
}
```

- Variável `y`
  - **Escopo:** visível em `foo`
  - **Tempo de vida:** estende por todo o programa
- Variável `x`
  - **Escopo:** não estende `foo`
  - **Tempo de vida:** estende `foo`

AMBIENTES DE REFERENCIAMENTO

Linguagens de Programação

Ambientes de Referenciamento

- Conjunto de todos os nomes visíveis em uma determinada instrução ou ponto de programa
  - O ambiente de referenciamento é constituído de todas as variáveis declaradas em seu escopo local, com o conjunto de todas as variáveis de seus escopos ascendentes visíveis

```
program example;
    var a, b: integer;
    procedure sub1;
        var x, y: integer;
        begin (1) end;
    procedure sub2;
        var x: integer;
        procedure sub3;
            var x: integer;
            begin (2) end;
        begin (3) end;
    begin (4) end;
```

Ponto (1)

sub1: x, y

example: a, b

Ponto (2)

sub3: x

example: a, b

Ponto (3)

sub2: x

example: a, b

Ponto (4)

example: a, b

ESCOPO DINÂMICO

Escopo Dinâmico

- Todas as técnicas de escopo vistas até agora foram estáticas (**escopo estático** ou **escopo léxico**)
  - A verificação se uma ocorrência de um nome está no escopo de uma dada definição é feita em **tempo de compilação**

- Uma outra abordagem, embora raramente utilizada por linguagens modernas, é o **escopo dinâmico**
  - A linguagem adia esta decisão quando o programa roda, ou seja, em **tempo de execução**

Escopo Dinâmico

- No escopo dinâmico toda função é associada a um ambiente de definições
  - Se um nome que ocorre numa função não for encontrado neste ambiente, então procura-se no ambiente de seu chamador
  - E se não for encontrado lá, a busca continua seguindo a cadeia de chamadas

- O escopo de uma definição é a função que contém esta definição, a partir do ponto de sua definição até o final da função, além de quaisquer funções que sejam chamadas (mesmo que indiretamente) dentro deste escopo menos o escopo de qualquer redefinição de mesmo nome nestas funções chamadas

Escopo Estático vs Escopo Dinâmico

- Considere o seguinte trecho em ML

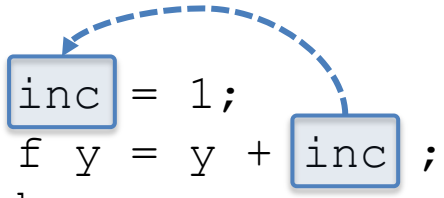
```
fun g x =
  let
    val inc = 1;
    fun f y = y + inc;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```

Qual o resultado da chamada `g 5` utilizando o escopo clássico de ML (escopo estático)?

Escopo Estático

- No escopo estático, a referência para `inc` é vinculada à definição prévia no mesmo bloco (a definição de `inc` no chamador de `f` é inalcançável)
  - Portanto, o resultado da chamada `g 5` é `6`

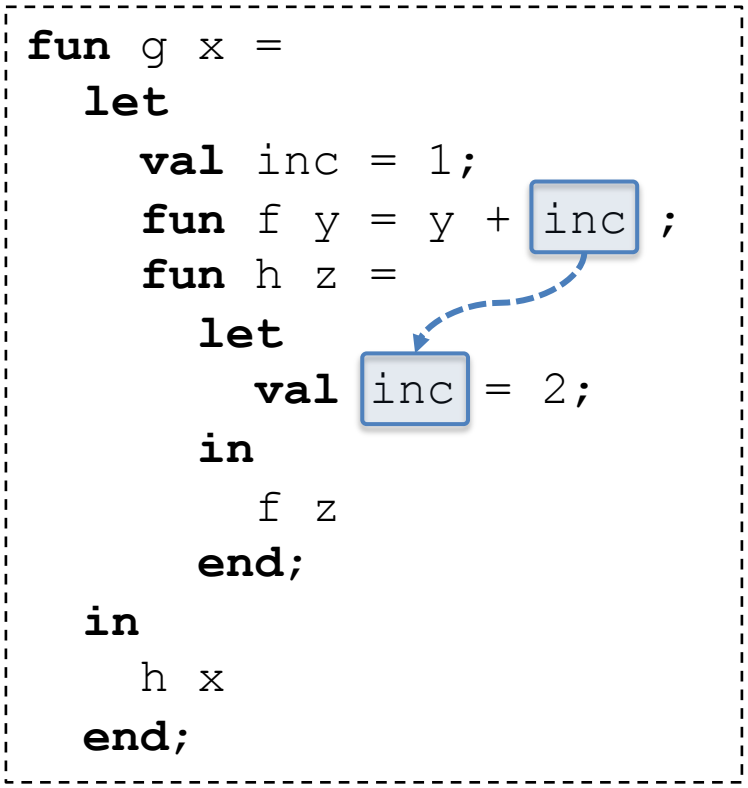
```
fun g x =
  let
    val inc = 1;
    fun f y = y + inc ;
    fun h z =
      let
        val inc = 2;
      in
        f z
      end;
  in
    h x
  end;
```



E se ML utilizasse escopo dinâmico, qual seria o resultado de `g 5`?

Escopo Dinâmico

- No escopo dinâmico, a referência para `inc` é vinculada à definição do ambiente do chamador
  - Portanto, o resultado da chamada `g 5` seria `7`



Escopo Dinâmico

- Escopo dinâmico é usado em poucas linguagens: em alguns dialetos de Lisp, APL e como uma opção em Common Lisp
- Shell Script suporta escopo dinâmico se utilizado o modificador `local` ou `typeset` na definição da variável em funções

```
#!/bin/bash

x=1
function sub1() {
  echo $x
}

function sub2() {
  local x=3
  sub1;
}

sub2;
sub1;
```

Qual o resultado da execução deste script? E se remover o modificador `local`? O que pode se dizer sobre o escopo de Shell Script?

Escopo Dinâmico

- Problemas
  - **Perda de eficiência**: verificação de tipos em tempo de execução
  - **Redução na legibilidade**: difícil determinar a sequência de chamadas
  - **Acesso lento a variáveis não-locais**: difícil identificar os identificadores da sequência de chamadas
  - **Redução da confiabilidade**: variáveis locais acessadas por quaisquer funções chamadas

COMPILAÇÃO SEPARADA

Linguagens de Programação

Compilação Separada

- Na sequência clássica foi visto que partes do programa podem ser compiladas separadamente para então serem ligadas posteriormente

- Problemas de escopo também são estendidos para o ligador
  - Ele precisa conectar referências para definições através de separadas compilações

Como linguagens lidam com isto?

Compilação Separada

- A linguagem C suporta dois tipos diferentes de definição
  - Uma definição completa
  - Um protótipo somente com nome e tipo
    (sem corpo se for função ou sem valor se for variável)

- Por exemplo, se várias compilações separadas quiserem usar a mesma variável inteira `x`, deve-se
  - Criar uma única definição completa em algum lugar

```
int x = 3;
```

  - E usar a seguinte definição em todos os outros lugares

```
extern int x;
```

Compilação Separada

- Dialetos antigos de FORTRAN usam blocos `COMMON` para o mesmo propósito
  - Todas as compilações separadas definem variáveis normalmente (por exemplo: `A, B, C`)
  - E todas as compilações incluem a declaração `COMMON` (por exemplo: `COMMON A, B, C`)

- Dialetos modernos de FORTRAN possuem mecanismos adicionais
  - Uma forma de definir dados em uma compilação separada como um módulo (`MODULE`)
  - E uma forma de importar estas definições em uma compilação separada (`USE`)

`USE` diz qual módulo utilizar, mas não diz quais são as definições

Compilação Separada

- Em linguagens recentes, compilação separada é *menos separada* que antigamente
  - Classes em Java podem depender uma das outras de forma circular, portanto o compilador de Java deve ser capaz de compilar classes separadas simultaneamente
  - ML não foi projetado para compilação separada de forma alguma, embora *CM* (uma ferramenta auxiliar do sistema, o gerenciador de compilação) consegue fazer isto para a maioria dos programas

ISSO É TUDO, PESSOAL!