nada em linguagens de programação é tão requisitado quanto o mecanismo de 'nomes'. são os nomes que nos permitem, na programação, 'apontar' coisas sobre as quais queremos causar algum efeito. e um nome sempre vale por aquilo que ele aponta, ou seja, o elemento ao qual ele está 'vinculado'. não são os nomes, mas os 'vínculos', o real conceito a se explorar. sua atuação tem um ciclo de vida, o que nos remete ao conceito de escopo. as linguagens de programação revelam diversidade também nesse tópico.

1. Vínculos (*bindings*)

Nomes estão por toda parte em linguagens de programação. Nomes não têm valor por si sós, mas por aquilo que eles simbolizam, algo na execução de um programa que pode ser manipulado através deles. Vínculo (*binding*) é como se chama tecnicamente essa relação entre um nome e uma entidade sob efeito da programação.

Exemplos de nomes e vínculos:

- **Palavras reservadas e operadores da linguagem**. `if`, `then`, `for`, `+` são alguns exemplos de nomes vinculados a mecanismos da linguagem de programação. Pode parecer surpreendente, mas eles também podem ser vistos como um tipo de vínculo. Em algumas linguagens de programação, eles são definidos (vinculados) quando se implementa o interpretador ou compilador (na verdade, quando se *projeta* a linguagem de programação). Em outras, eles são nomes como quaisquer outros, podendo ser reconfigurados durante a execução (!).
- **Identificadores**. O termo 'identificador' é um termo técnico em linguagens de programação. Ele designa os nomes criados pelo programador. Ou seja, nomes de variáveis, nomes de funções (procedimentos, sub-rotinas), nomes de pacotes (módulos, espaços de nomes), etc. Em geral, existem regras para a criação de identificadores. Todo curso de programação tem um momento para explicar essas regras na linguagem de programação que adota. Variáveis são vínculos entre nomes e unidades de memória, funções são vínculos entre nomes e código, e assim por diante.

2. Vinculação estática

Um vínculo tem um ciclo de vida, isto é, ele se inicia de alguma forma, participa da execução do programa por algum tempo podendo sofrer alterações ou não, e termina.

Vinculação estática é aquela que pode ser definida sem depender de informações que só estejam disponíveis durante a execução do programa. Por exemplo, assim acontece com as variáveis globais. O programador cria uma variável global em seu programa. O compilador reserva uma posição para ela no código objeto gerado em linguagem de montagem, e mantém o nome da variável no arquivo. O vínculo se formou entre aquele nome e a posição de memória prevista. O montador (*assembler*) gera o código objeto a partir disso, e mantém a reserva de espaço para a variável, assim como seu nome numa tabela de símbolos. O link-editor (*linker*) constrói um arquivo objeto final rearranjando os códigos objetos do programa e das bibliotecas. Talvez o nome não conste mais desse arquivo, mas a posição reservada tem um endereço de arquivo, e todas as operações que originalmente envolviam o nome da variável possivelmente agora usem o endereço de arquivo. Por fim, o *loader* posiciona o arquivo na memória antes de executar, realizando o endereço efetivo final daquela variável primeira. Embora várias transformações tenham ocorrido ao longo do processo, e os valores de endereço vinculados à variável aparentemente tenham mudado, é uma mudança física, mas não lógica, no sentido em que essas palavras são usadas no jargão de bancos de dados. **Durante a execução do programa, esse vínculo é fixo, isto é, todas as referências originais à variável acontecem agora como referências a um endereço determinado, e que corresponde à variável. O vínculo não muda de acordo com algum fator presente na execução, e foi completamente definido antes dela, portanto é um caso de vinculação estática.**

Variáveis locais são alocadas dinamicamente em pilha. Sua existência é incerta até que o programa seja executado. Diversas instâncias da mesma variável podem coexistir na pilha. Ainda assim, **variáveis locais também são um exemplo de vinculação estática.** Isso porque, embora o endereço físico da variável só apareça, se aparecer, na pilha durante a execução do programa, ele é acessado por um ponteiro cuja existência e funcionamento são completamente definidos previamente. A variável local `i` no programa em linguagem de alto nível de abstração se transforma, por exemplo, na posição 8 unidades de memória abaixo do ponteiro de topo de pilha[1]. Ou seja, o vínculo já está bem definido independente de fatores.

[1] Na verdade, um ponteiro auxiliar que registra qual era a posição do ponteiro de topo de pilha quando a sub-rotina foi chamada.

3. Vinculação dinâmica na orientação a objetos (*dispatcher*)

Um exemplo emblemático de vinculação dinâmica (*dynamic binding*) é o que acontece com os métodos na orientação a objetos, e que é chave para o funcionamento dos mecanismos de herança e polimorfismo.

No programa abaixo, contido em um arquivo `MyDate.java`, a classe criada herda o código de `java.util.Date`, exceto pelo método `toString()`, que é sobrescrito. Ou seja, a nova classe modifica o método, e duas versões do mesmo passam a existir: a versão original de `java.util.Date` e a versão de `MyDate` (polimorfismo é isso: um nome, várias formas).

O método `toString()` é usado na função `printDate`, na chamada destacada do programa. Qual código está vinculado a esse nome? Qual endereço? Como a execução do programa mostra, na verdade não há essa definição. A chamada da função deve passar por uma seleção (um *switch* no jargão da linguagem C) baseada no tipo efetivo da variável `d`. Se `d` é do tipo `java.util.Date`, o método `toString()` original é invocado; se `d` é do tipo `MyDate`, o novo método `toString` entra em ação. **O vínculo só é definido no momento exato da chamada do método, por isso essa é uma vinculação dinâmica.** O processo de definição do vínculo com base no tipo é conhecido como *dispatcher*, e é, para muitos pesquisadores da área de teoria de linguagens de programação, a característica mais essencial da orientação a objetos.

```
import java.util.Date;
public class MyDate extends Date {
	public String toString() {
		return "**TODO DIA É FERIADO**";
	}
	public static void printDate(Date d) {
		System.out.println( d.toString() );
	}
	public static void main(String[] args) {
		printDate(new Date());
		printDate(new MyDate());
	}
}
```

Compilação e execução do programa:

```
$ javac MyDate.java
$ java MyDate
Wed Oct 14 22:04:40 BRT 2015
**TODO DIA É FERIADO**
```

Obviamente, a vinculação dinâmica tem um impacto negativo no desempenho. Gasta-se tempo para analisar o tipo da variável. É nesse quesito que encontramos um dos exemplos mais citados de técnica de otimização em compiladores JIT. Quando um método possui uma única versão carregada na memória, o compilador JIT a invoca diretamente, sem passar por um *dispatcher*.

4. Escopo e escopamento

A palavra escopo quer dizer *abrangência*, *alcance*. “O escopo do projeto era comprar uma nova mesa, e não reformar a mobília”. **Em linguagens de programação, fala-se em “escopo de uma variável”, que na verdade é o escopo de um nome, ou melhor o escopo de um vínculo.**

Escopamento (*scoping*) é o tópico mais relevante. Trata-se da forma como a linguagem de programação define o escopo de seus vínculos. O problema fundamental do escopamento está definido a seguir:

> problema fundamental do escopamento em L.P.
>
> dada um nome não definido localmente em uma sub-rotina, como determinar a quem está vinculado?
>
> por exemplo, havendo mais de uma definição de variável com o nome, a qual delas o nome se refere?

O programa abaixo é escrito em Pascal. Nessa linguagem, há a distinção entre funções e procedimentos (*procedures*). Enquanto funções sempre resultam em algo, isto é, têm algum valor de retorno, procedimentos, não (procedimentos correspondem a funções em C cujo tipo de retorno é declarado com `void`). Em Pascal, funções e procedimentos podem ser declarados dentro de outras funções e procedimentos. Em outras palavras, Pascal adota declarações aninhadas (*nested*). Talvez essa característica justifique a presença dominante do Pascal nos exemplos sobre escopamento em livros sobre linguagens de programação.

No exemplo abaixo, a variável `z` em destaque é um nome não local da procedure `alpha`. Existem duas variáveis `z` declaradas no programa, uma global e uma local do procedimento `g`. **É preciso conhecer o escopamento da linguagem para resolver `z`.** Se a linguagem adota o escopamento estático, como é o caso do Pascal, então a variável global `z` é usada em `alpha`. Contudo, nem todas as linguagens funcionam dessa forma.

```
program clp_scope;
      var x,y,z:integer;

      function f(i:integer):real;
            var x:real;

            procedure alpha();
                  var y:real;
                  begin
                    y := 6.23;
                    write(y + z);
                  end;

            begin
              x := 3.14;
              alpha;
              f := x * i;
            end;

      procedure g();
            var z:real;
            begin
              z := 2.718;
              z := f(7);
            end;

begin
  x := 1024;
  y := 625;
  z := 333;
  g;
  write( f(0) );
end.
```

Em toda a discussão que se segue, emerge um dos conceitos mais importantes em toda a teoria de linguagens de programação: o conceito de *ambiente de referenciamento*. A ideia é simples: ambiente de referenciamento é o conjunto de todas os vínculos ativos em um determinado momento da execução de um programa.

5. Escopamento dinâmico

O escopamento dinâmico possui a seguinte regra para a definição do ambiente de referenciamento em um dado momento da execução do programa:

1. Todas as variáveis locais da sub-rotina em execução são incluídas no ambiente de referenciamento.
2. Todas as variáveis da sub-rotina (se houver essa sub-rotina) que invocou a sub-rotina do passo 1 são incluídas no ambiente de referenciamento, exceto aquelas cujos nomes já constem nele.
3. Todas as variáveis da sub-rotina (se houver) que invocou a sub-rotina do passo 2 são incluídas no ambiente de referenciamento, exceto aquelas cujos nomes já constem nele...
4. ...e, assim, por diante, até chegar à sub-rotina que começou o programa.

No programa de exemplo (`clp_scope`), na linha do procedimento `alpha` em que `z` é mencionado, o cálculo do ambiente de referenciamento, realizado segundo a regra, é como segue:

Inicialmente, o ambiente de referenciamento contém as variáveis locais de `alpha`.

variável	definida_em
y	alpha

Quem invocou `alpha`? Examinando todo o programa, constata-se que ela é sempre chamada a partir de `f`. Portanto, as variáveis de `f` são incluídas no ambiente de referenciamento.

variável	definida_em
y	alpha
i	f
x	f

Quem invocou `f`? Depende. Há duas chamadas de `f` no programa: uma em `g` e outra em "`main`". Supondo que `f` tenha sido chamada de `g`, então as variáveis de g são incluídas no ambiente de referenciamento:

variável	definida_em
y	alpha
i	f
x	f
z	g

`g` só pode ter sido chamado de "`main`". Há três variáveis a serem incluídas, `x`, `y` e `z`, mas já há variáveis com esses nomes no ambiente de referenciamento, portanto, nenhuma delas entra. "`main`" não pode ser invocada por ninguém, é a sub-rotina que inicia o programa, portanto o ambiente de referenciamento obtido até aqui é o ambiente de referenciamento final.

Mas, e se a linha de execução em que `f` é chamada de "`main`" tivesse sido seguida? Voltando àquele ponto, e continuando, as variáveis `x`, `y` e `z` teriam que ser incluídas, mas `x` e `y` já estariam lá, logo apenas `z` entraria. O ambiente de referenciamento obtido (e final, pois é "`main`"):

variável	definida_em
y	alpha
i	f
x	f
z	main

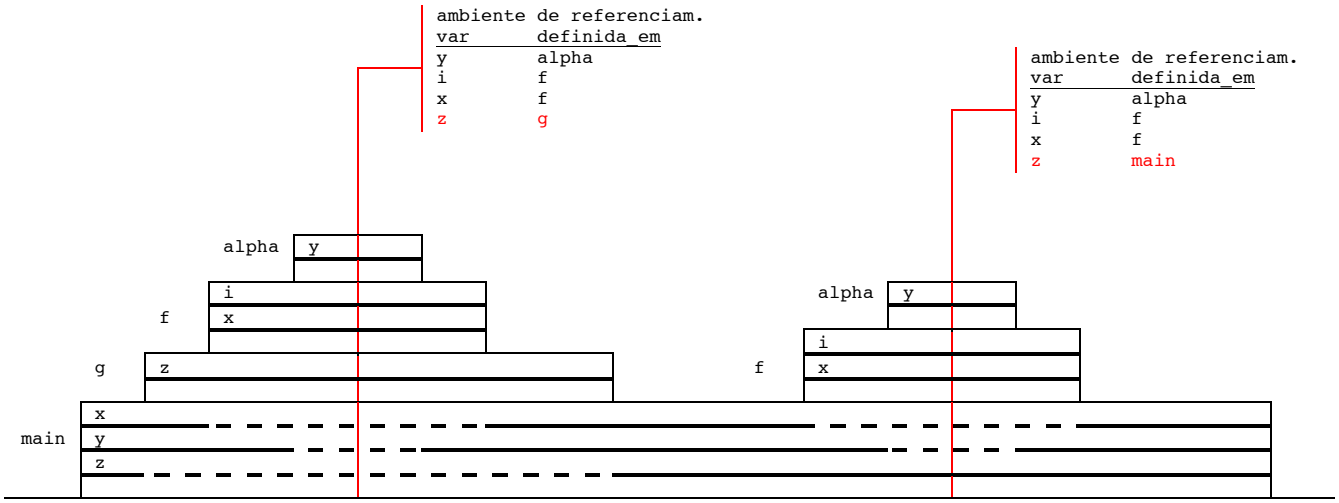
Comparando os dois possíveis ambientes de referenciamento, conclui-se que a definição do nome não local no escopamento dinâmico depende da linha de execução. Portanto, ela não pode ser definida antes do programa entrar em funcionamento, ou seja, é uma propriedade **dinâmica**.

Em exercícios sobre escopamento dinâmico, o enunciado deve mencionar uma linha de execução definida a fim de que se obtenha um ambiente de referenciamento. Para resolver a questão, como visto, basta seguir a pilha de chamadas no sentido contrário, da última sub-rotina chamada (isto é, a mais recente) até a mais antiga (a primeira).

Uma forma alternativa de resolver exercícios de escopamento dinâmico é desenhar o histórico da pilha segundo a sequência de chamadas informada na questão.

O tempo é o eixo horizontal. Cada função invocada cria um bloco no topo da pilha. Dentro do bloco, linhas horizontais contínuas representam variáveis ativas. Se uma variável de mesmo nome é definida num bloco superior, a linha da variável inferior deve ser tracejada, indicando que ela não está ativa. (Tecnicamente, diz-se que uma variável faz sombra [*shadow*] à outra.) Cortes (linhas verticais em qualquer ponto) interceptam as linhas ativas das variáveis do ambiente de referenciamento naquele ponto da execução do programa.

O diagrama para o programa `clp_scope` é mostrado a seguir. Esse programa é uma exceção no quesito da linha de execução: ele não tem condicionais, portanto segue sempre a mesma sequência. Em outras palavras, não é preciso fornecer no enunciado uma sequência a ser seguida na definição do ambiente de referenciamento.



6. Vantagens e desvantagens do escopamento dinâmico

É fácil implementar o escopamento dinâmico, basta empregar o mecanismo de pilha já existente no ambiente de execução da linguagem de programação. Essa é sua principal vantagem.

Por outro lado, o escopamento dinâmico **não dá garantias ao programador** sobre o funcionamento do seu programa. Essa desvantagem é significativa, como se segue.

Um programador cria uma biblioteca, mostrada abaixo em uma pseudolinguagem de programação. Na função, a variável `total_clientes` é não local. O programador pensou a função atuando com a variável global associada. O bom funcionamento do programa depende disso.

```
{cobranca.lib}

var total_clientes

function calcular_produtividade()
       return ( somar(obter_bd(vendas)) – somar(obter_bd(custos)) ) / total_clientes
```

Outro programador usa a biblioteca. Desavisadamente, ele aciona a função num contexto em que existe outra variável com o nome `total_clientes`. Isso muda o funcionamento da função na biblioteca.

```
import cobranca.lib

function relatorio_setor()
    var total_clientes, total_vendedores, …
    …
    print calcular_produtividade()
```

Erros desse tipo são difíceis de encontrar.

O programador não tem nenhum controle sobre isso, a não ser que evite variáveis não locais, o que não é uma solução.

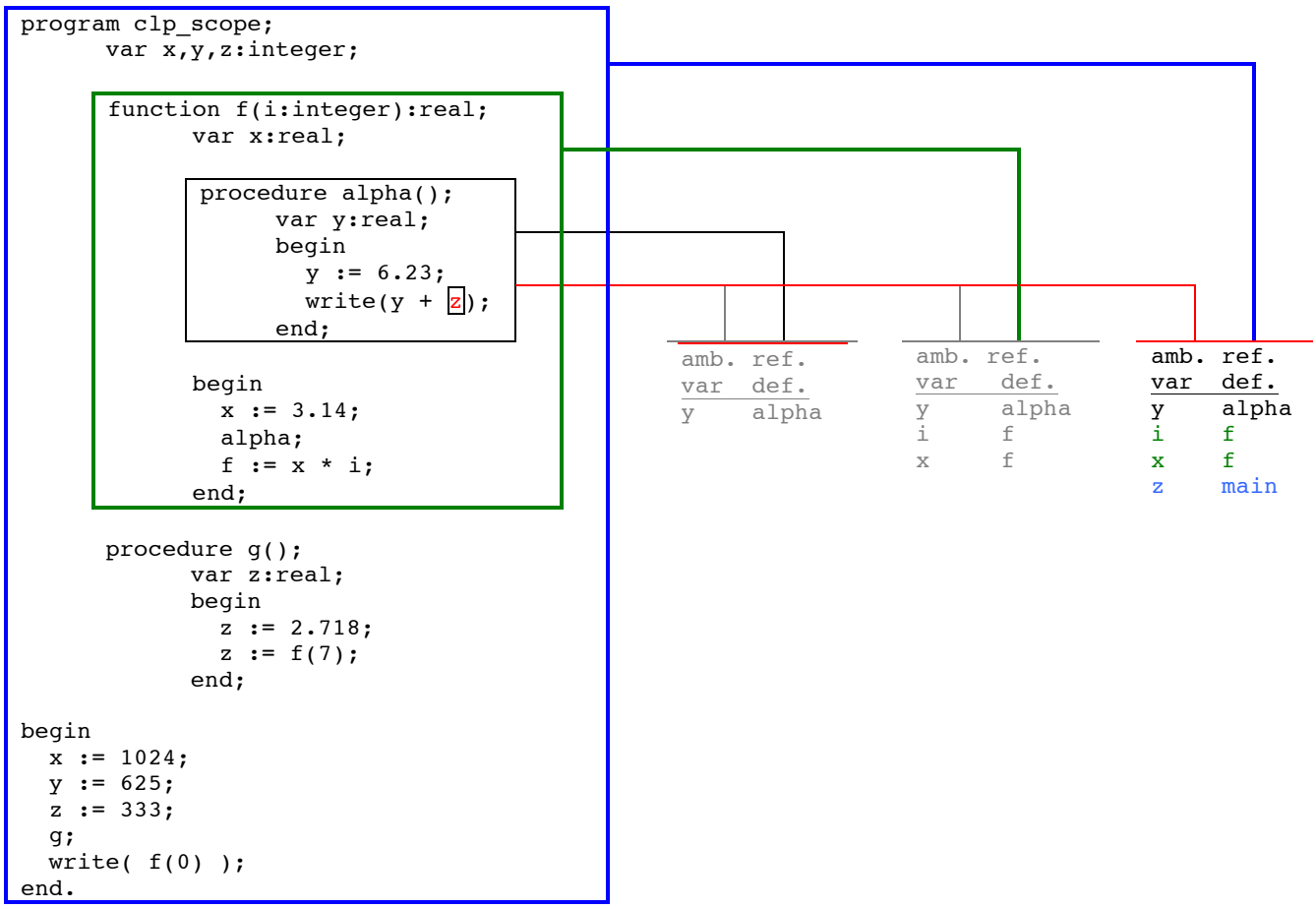
7. Escopamento estático (ou escopamento léxico)

No escopamento estático, a definição dos vínculos é feita apenas com base no código-fonte do programa. Também é chamado de escopamento léxico, no sentido de "como está escrito" (*lexis* em grego = palavra). Portanto, ocorre antes da execução do programa (propriedade estática).

A definição dos ambientes de referenciamento se dá por regiões do programa:

1. Todas as variáveis locais da sub-rotina em questão são incluídas no ambiente de referenciamento.
2. Todas as variáveis da sub-rotina (se houver essa sub-rotina) **que engloba, no código fonte,** a sub-rotina do passo 1 são incluídas no amb. de ref., exceto aquelas cujos nomes já constem nele.
3. Todas as variáveis da sub-rotina (se houver) que **engloba** a sub-rotina do passo 2 são incluídas no ambiente de referenciamento, exceto aquelas cujos nomes já constem nele...
4. ...e, assim, por diante, até chegar ao **escopo global**.

O cálculo do ambiente de referenciamento estático para o exemplo estudado anteriormente é ilustrado a seguir:



Os vínculos assim definidos não dependem da ordem de execução das sub-rotinas. O escopamento estático, assim, **torna previsível o funcionamento do programa**, conferindo garantias ao programador.

Essa é a principal razão do predomínio do escopamento estático nas linguagens de programação atuais. A ideia de escopamento estático ficou bem estabelecida na comunidade científica em 1960 na especificação da linguagem Algol-60, que nunca foi implementada completamente, e da qual derivam as principais linguagens históricas, como C e Pascal. Linguagens caracterizadas pelo escopamento dinâmico, como Lisp e Perl, foram adotando o escopamento estático paulatinamente. Na família Lisp, Scheme já surgiu com escopamento estático nos anos 1970, e o Emacs Lisp está entrando no grupo agora. Perl passou a adotar escopamento estático na versão 5.

O empecilho para a adoção do escopamento estático é a maior dificuldade de implementação. Sobre técnicas de implementação de escopamento estático, confira **Conceitos de Linguagens de Programação**, de Robert Sebesta, no capítulo de implementação de subprogramas.

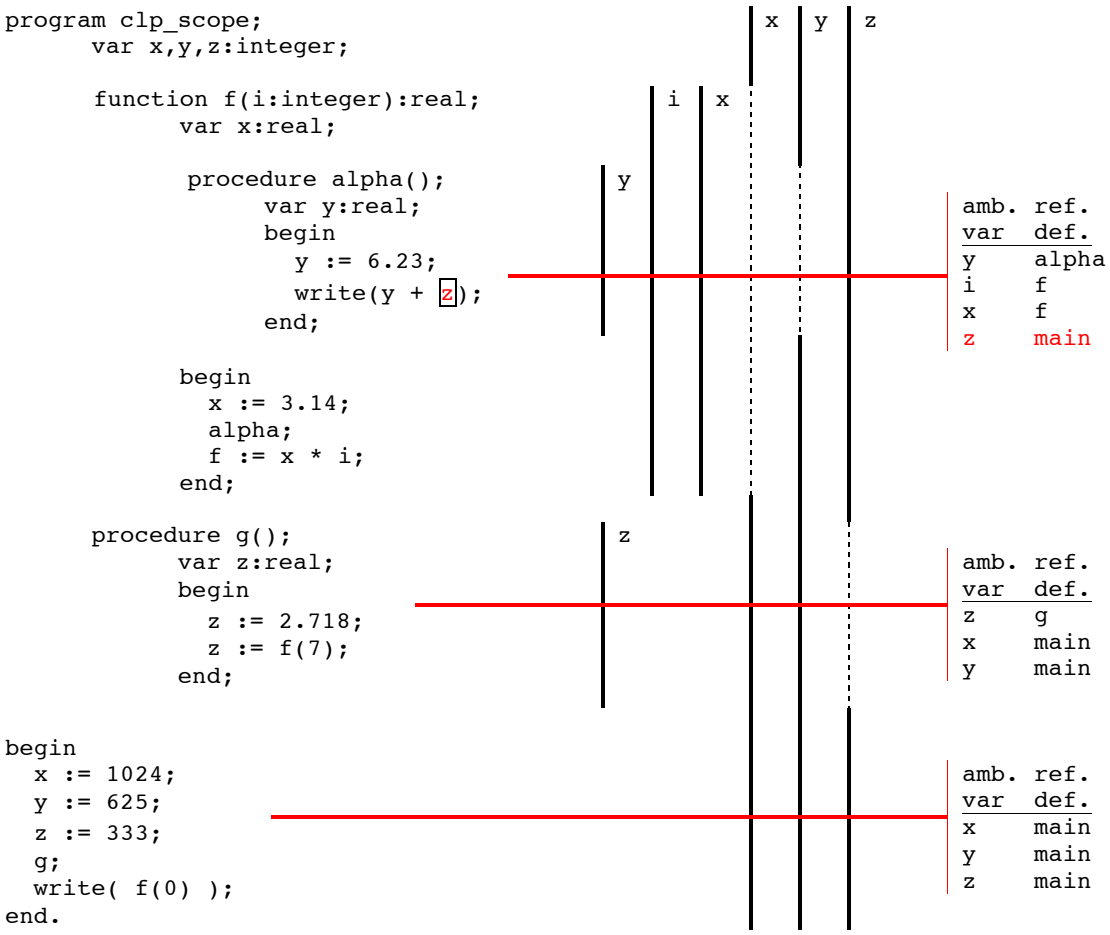
Uma forma alternativa de calcular os ambientes de referenciamento do escopamento estático consiste em **traçar linhas verticais ao lado do código-fonte** do programa para cada variável, **assinalando a região em que a variável está ativa (seu escopo)**. Começa-se com as variáveis locais, passando para as sub-rotinas envolventes (de dentro para fora) terminando com o escopo global.

Quando, numa região do código, houver várias linhas referentes a um mesmo nome, todas devem ser tracejadas com exceção de uma, que é a mais próxima do nível local. Essa variável é a única ativa naquela região com esse nome, fazendo sombra (*shadow*) às demais.

C++ possui um operador escopo (`::`), mediante o qual é possível acessar variáveis sombreadas. Se houvesse esse operador em Pascal, a variável `y` global, sombreada em `alpha`, poderia ser acessada ali com a expressão `main::y`.

Para definir o ambiente de referenciamento em uma região do programa, basta traçar uma linha horizontal. O ambiente de referenciamento consiste em todas as linhas verticais ativas que ela corta.

A vantagem dessa forma de cálculo é possibilitar a definição de ambientes de referenciamento de diversas regiões do programa de uma só vez. Por exemplo, na ilustração, estão indicados também os ambientes de referenciamento em `g` e `main`.



8. O escopamento dinâmico não acabou

Hoje em dia, linguagens de programação já estreiam adotando escopamento estático. Algumas delas, no entanto, oferecem **como opção** algum mecanismo de escopamento dinâmico. Esse fenômeno nos leva a refletir sobre o papel de variáveis não locais nas linguagens de programação.

Por que existem variáveis não locais? Qual seria a implicação da proibição do uso de variáveis não locais na programação?

Variáveis não locais são um tipo de parâmetro das sub-rotinas, por onde também fluem informações para o interior das sub-rotinas. Se não fosse permitido o uso de variáveis não locais, as sub-rotinas teriam muitos parâmetros, o que deixaria muito carregadas as declarações e invocações das sub-rotinas. **Variáveis não locais são parâmetros ocultos das subrotinas**, como se fossem "portas laterais", "portas dos fundos", "entradas não explícitas". Elas proporcionam alguma forma de agilidade à programação.

O escopamento estático implica a definição prévia de todas as variáveis não locais. O escopamento dinâmico permite substituir uma variável não local, nem que seja temporariamente, o que se discute a seguir.

Muitas linguagens da família Lisp adotam algum tipo de escopamento dinâmico *ad hoc*, por exemplo, Common Lisp e Clojure. O exemplo é escrito em Clojure:

```
(def ^:dynamic *saudação* "Olá")

(defn saudar [alguém]
   (str *saudação* ", " alguém "!"))

(saudar "Pixinguinha")
;=> "Olá, Pixinguinha!"
```

```
(binding [*saudação* "Bom dia"]
   (saudar "Caymmi"))
;=> "Bom dia, Caymmi!"

(saudar "Braguinha")
;=> "Olá, Braguinha!"
```

Na primeira linha, uma variável é criada. O nome da variável é `*saudação*`, e seu valor, a string `"Olá"`. Ela é configurada como dinâmica através do mecanismo de metadado de variável existente em Clojure. Os asteriscos no nome da variável (também chamados de "fones de ouvido") são uma convenção adotada em muitos Lisps para indicar variáveis com escopamento dinâmico.

Em seguida é criada a função `saudar`, na qual a variável `*saudação*` aparece como variável não local. A função cria uma string de saudação mediante o nome de uma pessoa fornecido como parâmetro.

Na primeira invocação de `saudar`, o valor originalmente atribuído a `*saudação*` é usado.

A segunda invocação de `saudar` ocorre num contexto em que uma nova variável com o nome `*saudação*` é criada no ambiente de referenciamento, fazendo sombra à variável original (isso é escopamento dinâmico). A função utiliza a nova variável em sua execução.

Na terceira invocação, quando o contexto da nova variável já se extinguiu, a função volta a usar a variável original.

Se não houvesse o escopamento dinâmico opcional, o mesmo efeito poderia ter sido obtido com uma variável global que precisaria ter sido alterada duas vezes (para o novo valor primeiro, e para o valor anterior depois). O escopamento dinâmico propicia agilidade nesse sentido. **Nos Lisps em geral, muitas variáveis de "configuração" do sistema são declaradas com escopamento dinâmico (opcional).**

O escopamento dinâmico se manifesta em linguagens de programação com escopamento estático de formas inusitadas. Michael Fogus, em seu livro **Functional JavaScript**, chama a atenção para o escopamento dinâmico associado à palavra reservada `this`.

`this` em JavaScript é um nome vinculado a um objeto. Qual objeto? Digite os seguintes comandos num interpretador (isto é, num REPL) JavaScript (por exemplo, o console do Google Chrome):

```
function capturaThisGlobal(a, b) { return this; }

capturaThisGlobal(3, 4);
//=> algum objeto global, provavelmente Window

capturaThisGlobal.apply('clp', [3, 4]);
//=> 'clp'
```

Na primeira linha, é definida uma função chamada `capturaThisGlobal`, com dois parâmetros, `a` e `b` (que não são usados na função), e que resulta no objeto que `this` aponta.

Na sua primeira invocação, a função fornece um objeto `Window`.

Quando a função é invocada através do seu método `apply` (todas as funções em JavaScript possuem esse método), que conta com um parâmetro inicial para receber um objeto a ser atribuído a `this` e um parâmetro para receber os argumentos da função reunidos em um array, observa-se o escopamento dinâmico do nome `this`.

9. Onde começa o escopo?

Há pelo menos duas formas de definição do escopo de um vínculo a ser adotadas por uma linguagem de programação:

- o vínculo vale da declaração do nome até o fim do bloco (em geral, o fim da sub-rotina)
- o vínculo vale desde o início do bloco, mesmo antes da declaração, até o seu fim.

Isso muda o sentido do seguinte programa em pseudocódigo:

```
var i = 3

function f()  // declara f
     print i
     var i = 10
     print i

f()  // executa f
```

Na primeira forma, os valores impressos seriam 3 e 10; na segunda forma, seriam 10 e 10.

A primeira forma é mais comum em linguagens de programação. Um exemplo de linguagem que adota a segunda forma é C#.

10. O que é uma variável?

O que é uma variável em linguagens de programação?

- Um nome?
- Um valor?
- Um endereço?
- Um vínculo?
- Um tipo?

Resposta: a combinação de tudo isso, mas cada linguagem de programação tem sua forma peculiar de fazê-lo. Em linguagens que incentivam a programação com dados imutáveis (Haskell, Ocaml, Clojure), variáveis são chamadas de *valores*, e pensadas como um puro vínculo nome-valor. Embora na implementação subjacente uma célula de memória esteja envolvida (isto é, um endereço), o programador pode se permitir deixar de pensar em endereços. A programação deixa de ser uma disciplina de “variáveis” e passa a ser uma disciplina de “valores”. De todo jeito, sempre serão **nomes vinculados**.