Linguagens de Programação
Conceitos e Técnicas

Amarrações

Prof.

Conceituação

- Amarração (ou *binding*) é uma associação entre entidades de programação, tais como entre uma variável e seu valor ou entre um identificador e um tipo
- Enfoque na amarração de identificadores a entidades

Tempos de Amarração

Identificador ou Símbolo	Entidade	Tempo de Amarração
*	Operação de multiplicação	projeto da LP
int	Intervalo de inteiros	projeto da LP (JAVA) implementação do compilador (C)
variável	Tipo	compilação (C) execução (Python)
função	Código correspondente da função	ligação
variável global	Variável em memória	carga do programa
variável local	Variável em memória	execução

- Amarração Estática X Dinâmica

Identificadores

- Identificadores são cadeias de caracteres definidas pelos programadores para servirem de referência a entidades de computação
- Objetivam aumentar a legibilidade, redigibilidade e modificabilidade
- LPs podem ser *case sensitive* e limitar o número máximo de caracteres
  - Não é adequado diferenciar identificadores pelo formato maiúsculo x minúsculo
    - casa X CASA

Identificadores

- Alguns identificadores podem ter significado especial para a LP
  - Palavras Reservadas
    - PYTHON print
    - C while
  - Palavras Pré-definidas
    - Python str

      str = 'loucura'
  - Palavra Chave
    - FORTRAN

      INTEGER REAL

      REAL INTEGER

Ambientes de Amarração

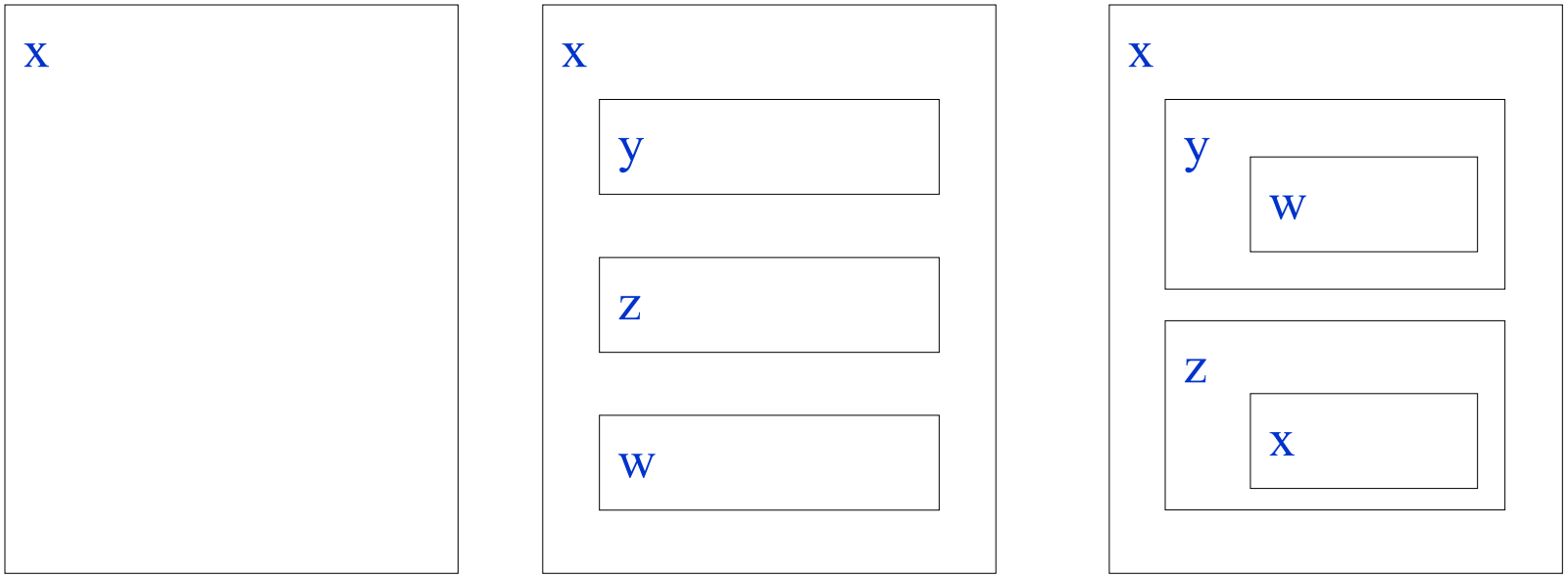
- Amarração de Identificador a Duas Entidades Distintas no Mesmo Ambiente

```
int a = 13;
void f() {
   int b = a;
   int a = 2;
   b = b + a;
}
```

Escopo

- Estático
  - definição do subprograma
  - tempo de compilação
  - texto do programa
- Dinâmico
  - chamada do subprograma
  - tempo de execução
  - fluxo de controle do programa

Escopo Estático



Bloco Monolítico | Blocos Não Aninhados | Blocos Aninhados

Escopo Estático

- Ocultamento de Entidade em Blocos Aninhados

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

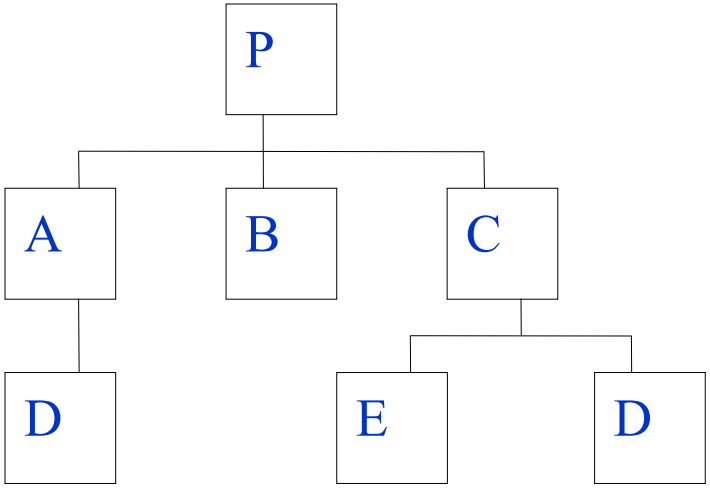
Escopo Estático

- Referência Seletiva em ADA

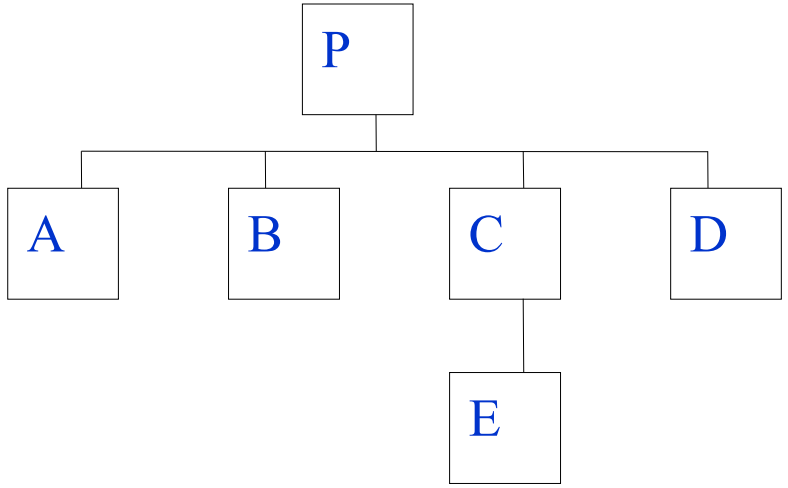
```
procedure A is
    x : INTEGER;
    procedure B is
        y : INTEGER;
        procedure C is
            x : INTEGER;
        begin
            x := A.x;
        end C;
    begin
        null;
    end B;
begin
    null;
end A;
```

Escopo Estático

- Problemas com Estrutura Aninhada



a



b

Escopo Estático

Estrutura de Blocos de C

```
int x = 10;
int y = 15;
int f() {
        return y – x;
}
void g() {
    int x = 20;
    x = x  + f();
}
```

```
void main() {
    g();
}
```

Escopo Estático

Perl

```
$x = 0;
sub f {
    return $x;
}
sub g {
    my $x = 1;  # outra $x
    return f();
}
print g()."\n";
```

Escopo Dinâmico

Perl

```
$x = 0;
sub f {
    return $x;
}
sub g {
    local $x = 1;   # outra $x
    return f();
}
print g()."\n";
```

Adendo

Perl

```
$x = 0;
sub f {
    return $x;
}
sub g {
    $x = 1;   # altera $x
    return f();
}
print g()."\n";
```

Escopo Dinâmico

- Problemas
  - Eficiência
  - Legibilidade
  - Acesso
  - Confiabilidade
- Não usado por maioria das LPs

Definições e Declarações

- Definições produzem amarrações entre identificadores e entidades criadas na própria definição
- Declarações produzem amarrações entre identificadores e entidades já criadas ou que ainda o serão

Definições e Declarações

- Localização de Definições de Variáveis em C++

```
void f() {
   int a = 1;
   a = a + 3;
   int b = 0;
   b = b + a;
}
```

Definição de Constantes

- Em C

```
const float pi = 3.14;  // nao pode ser modificado
void f(int i) {
  const int tam = 3*i;
  ...
}
```

- Em JAVA

```
final int const1 = 9;
final int const2 = (int)(Math.random()*20);
```

Definição de Constantes

- Em Ada

```
pi : constant float := 3.1415;
```

- Em Haskell

```
pi = 3.14
```

- Em Ruby

```
Pi = 3.14  # Definição sintática - Começa com letra maiúscula
Pi = 3.1415  # Ruby permite mudar valor de constante
             # Apenas produz warning
```

Declaração de Constantes

- Em C

```
#define pi 3.1415
```

  - Não aloca memória
  - Não tem escopo de visibilidade

Definições e Declarações de Tipos

Definições de Tipos em C

```
struct data {
    int d, m, a;
};
```

```
union angulo {
    int graus;
    float rad;
};
```

```
enum dia_util {
        seg, ter, qua,
    qui, sex
};
```

Declarações de Tipos em C

```
struct data;
typedef union angulo curvatura;
typedef struct data aniversario;
```

Definições e Declarações de Tipos

Definições de Tipos em ADA

```
type Cor is (Branca, Vermelha, Verde, Azul, Marrom, Preta);
type Idade is range 0 .. 130;
type Tabela is array(1 .. 10) of Integer;
```

Declarações Tipos em ADA

```
type CorParede is Cor;
type TempoVida is Idade;
```

Definições e Declarações de Variáveis

Definições de Variáveis em C

```
int k;          // global eh inicializada automaticamente com zero
union angulo ang;
struct data d;
int *p, i, j, k, v[10];
```

Definições com Inicialização

```
for (int i = 0; i < 100; i++) { } // i nao eh mais acessivel
char virgula = ',';
float f, g = 3.59;
int j, k, l = 0, m=23;
```

Definições e Declarações de Variáveis

- Definições com Inicialização Dinâmica

```
void f(int x) {
   int i;
   int j = 3;
   i = x + 2;
   int k = i * j * x;
}
```

- Definições com Inicialização em Variáveis Compostas

```
int v[3] = { 1, 2, 3 };
```

Definições e Declarações de Variáveis

- Declaração de Variáveis em C

```
extern int a;
```

- Declaração de Variáveis em C++

```
int r = 10;
int &j = r;
j++;
```

Definições e Declarações de Variáveis

- Definição de Variáveis em ADA

```
idade : integer;
x, y : float;
z : float := 2.0;
```

- Definição de Variáveis em Lua

```
idade = 1
x, y = 5, 3
num, str = 2, "ola"
```

Definições e Declarações de Variáveis

- Definição de Variáveis em Python

```
idade = 10
x, y = 5, 10
```

- Definição de Variáveis em Ruby

```
local = 34
@instancia = 42     # sintática
@@classe = 57       # sintática
```

Definições e Declarações de Variáveis

- Declaração de Variáveis em Python
  - global

```
v = 1
def f():
    print v
```

```
v = 1
def f():
    print v
    v = 10     # nao pode alterar localmente
#[UnboundLocalError]
```

```
v = 1
def f():
    global v
    print v
    v = 10
```

Definições e Declarações de Subprogramas

- Definição de Subprogramas em C

```
int soma (int a, int b) {
   return a + b;
}
```

- Declaração de Subprogramas em C

```
int incr (int);
void f(void) {
   int k = incr(10);
}
int incr (int x) {
   x++;
   return x;
}
```

Definições e Declarações de Subprogramas

- Definição de Subprogramas em ADA

```
procedure soma (a, b: in integer; c: out integer) is
begin
  c := a + b;
end soma;

function soma (a, b: in integer) return integer is
begin
  return a + b;
end soma;
```

Definições e Declarações de Subprogramas

- Definição de Subprogramas em Lua

```
function soma (a, b)
  return a + b
end

soma = function(a,b)
  return a+b
end

function afastaD(x,d)
  return x+d, x-d
end
```

Definições e Declarações de Subprogramas

- Definição de Subprogramas em Ruby

```
def soma ( a, b)
  return a + b
end

def afastaD ( x, d)
  return x+d, x-d
end
```

Definições e Declarações de Subprogramas

- Definição de Subprogramas em Python

```
def soma (a, b):
  return a + b

def afastaD(x,d):
  return (x+d, x-d)
```

Definições Compostas Seqüenciais

- Definições Seqüenciais em C

```
struct funcionario {
   char nome [30];
   int matricula;
   float salario;
};
struct empresa {
   funcionario listafunc [1000];
   int numfunc;
   float faturamento;
};
int m = 3;
int n = m;
```

Definições Compostas Seqüenciais

- Definições Seqüenciais em ML

```
val par = fn (n: int) => (n mod 2 = 0)
val negacao = fn (t: bool) => if t then false else true
val impar = negacao o par
val jogo = if x < y then par else impar
```

- Operador de Composição
  - Em Haskell

```
h = f . g
```

Definições Compostas Recursivas

- Definição Recursiva de Função em C

```
float potencia (float x, int n) {
   if (n == 0) {
      return 1.0;
   } else if (n < 0) {
      return 1.0/ potencia (x, -n);
   } else {
      return x * potencia (x, n - 1);
   }
}
```

- Tipo Recursivo em C

```
struct lista {
   int elemento;
   struct lista * proxima;
};
```

Definições Compostas Recursivas

- Erro em Definição de Função strcmp em C

```
int strcmp (char *p, char *q) {
    return !strcmp (p, q);
}
```

- Explicitação de Recursividade em Função ML

```
val rec mdc = fn ( m:int, n: int) = >
   if m > n then mdc (m – n, n)
   else if m < n then mdc (m, n – m)
   else m
```

Definições Compostas Recursivas

- Definições Mutuamente Recursivas em C

```
void segunda (int);  // se faltar assume: int segunda();
void primeira (int n) {
   if (n < 0) return;
   segunda (n – 1);
}
void segunda (int n) {
   if (n < 0) return;
   primeira (n – 1);
}
```

Definições Compostas Recursivas

- Estruturas Mutuamente Recursivas (C)

```
struct pai;
struct filho {
   char * nome;
   int idade;
   struct pai * pai;
};
struct pai {
   char * nome;
   int cpf;
   struct filho * filho;
};
```

```
int main(){
  struct pai p;
  struct filho f;
  p.filho = &f;
  f.pai = &p;
}
```

Definições Compostas Recursivas

- Estruturas Mutuamente Recursivas (Java)

```
public class A {
   public B b;
   public A() {
      System.out.println("A");
   }
}
public class B {
   public A a;
   public B() {
      System.out.println("B");
   }
}
```

```
public class C {
   public static void main (String[] args) {
      A a = new A();
      B b = new B();
      b.a = a;
      a.b = b;
   }
}
```

Definições Compostas Recursivas

- Atenção com Estruturas Cíclicas

```
public class A {
   public B b;
   public A() {
      b = new B();
      System.out.println("A");
   }
}
public class C {
   public static void main (String[] args) {
  A a = new A();
  }
}
```

```
public class B {
   public A a;
   public B() {
      a = new A();
      System.out.println("B");
   }
}
```

Definições Compostas Recursivas

- Estruturas Mutuamente Recursivas (Ruby)

```
class Autor
  attr_accessor :nome, :livros
  def initialize(nome)
    @nome = nome
    @livros = []
  end
end
```

```
class Livro
  attr_accessor :titulo, :autores
  def initialize(titulo)
    @titulo = titulo
    @autores = []
  end
end
```