

Teoria de Linguagens – *overview*

(Prof. Marco Rodrigo Costa)



Conceitos básicos

- Alfabeto (Σ): conjunto finito de símbolos
- Símbolo (ou caractere): entidade abstrata, não definida formalmente, como números, letras e outros
- Palavra (ou cadeia, ou sentença) (w): sequência finita de símbolos do alfabeto justapostos
 - Palavra vazia (ϵ): palavra sem símbolo
 - Σ^* : conjunto de todas as palavras possíveis sobre Σ
 - Σ^+ : conjunto de todas as palavras possíveis sobre Σ excetuando-se a palavra vazia $\Rightarrow \Sigma^+ = \Sigma^* - \epsilon$

Linguagem Formal

- Linguagem formal: é um conjunto de palavras sobre um alfabeto
 - Exemplo: considere o alfabeto $\Sigma = \{a, b\}$. São exemplos de linguagens sobre Σ :
 - Conjunto vazio: $= \{ \}$
 - Conjunto formado pela palavra vazia $= \{ \varepsilon \}$
 - Conjunto de palíndromos (palavras que têm a mesma leitura da esquerda para a direita e da direita para a esquerda): $\{\varepsilon, a, b, aa, bb, aba, aaaa, \dots\} \Rightarrow$ linguagem infinita

Gramática

- Gramática: é uma quádrupla ordenada $G = (V, T, P, S)$, onde:
 - V é um conjunto finito de símbolos **variáveis** ou **não-terminais**
 - T é um conjunto finito de símbolos **terminais** disjunto de V
 - P é um conjunto finito de pares, denominados **regras de produção**, onde a primeira componente é palavra de $(V \cup T)^+$ e a segunda componente é palavra de $(V \cup T)^*$
 - S é um elemento de V denominado **variável inicial**
- Uma regra de produção (α, β) é representada por $\alpha \rightarrow \beta$
- As regras de produção definem as condições de geração das palavras da linguagem \Rightarrow gramática é formalismo de geração
- $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n \equiv \alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- **Exemplo aplicado:** Mini Pascal

Gramática...

- Derivação: seja $G = (V, T, P, S)$, uma gramática. Uma derivação é um par da relação denotada por \Rightarrow com domínio em $(V \cup T)^+$ e contra-domínio em $(V \cup T)^*$. Logo, uma derivação (α, β) é representada por $\alpha \Rightarrow \beta$
 - A derivação é a substituição de uma subpalavra de acordo com uma regra de produção

Gramática...

- Linguagem gerada: seja $G = (V, T, P, S)$, uma gramática. A linguagem gerada pela gramática G , denotada por $L(G)$, é composta por todas as palavras de símbolos terminais deriváveis a partir do símbolo inicial S , ou seja:
$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$
- Gramáticas equivalentes: duas gramáticas G_1 e G_2 são ditas equivalentes se, e somente se, $L(G_1) = L(G_2)$.

Gramática...

- Convenções: utiliza-se, para:
 - Símbolos variáveis: A, B, \dots, S, T
 - Símbolos terminais: a, b, \dots, s, t
 - Palavras de símbolos terminais: u, v, w, x, y, z
 - Palavras de símbolos terminais ou variáveis: α, β, \dots
- Exemplos (gramática, derivação e linguagem gerada):
 - $V = \{S, A, B\}, T = \{a, b\}, P = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$
 - <Exemplo 22, pgs. 24 e 25, ref. Menezes>

Sistema de Estados Finitos

- Sistema de Estados Finitos: é um modelo matemático de sistema com entradas e saídas discretas
- Pode assumir um número finito e pré-definido de estados
- Cada estado resume somente as informações do passado necessárias para determinar as ações para a próxima entrada
- Exemplo clássico: elevador. Não memoriza as requisições anteriores. Cada “estado” sumariza as informações “andar corrente” e “direção de movimento”. As entradas são as requisições pendentes
- Exemplos: autômatos finitos (determinísticos ou não), autômatos com pilha e máquinas de Turing

Linguagens Formais – Contextos

- Teoria de linguagens
 - Projetos de Linguagens de Programação
 - Reconhecedores e tradutores de Linguagens de Programação. Especificamente, por exemplo, nas fases de análise léxica e sintática
- Implementação de linguagens
 - Empregada em técnicas de desenvolvimento de compiladores
- Problemas computacionais
 - Útil no estudo de viabilidade computacional de soluções de problemas
- Conceitos de linguagens de programação
 - Útil para melhor programação e projetos de linguagens

Exemplo aplicado da definição de uma LP –
Syntax of Mini-Pascal (Welsh & McKeag, 1980) –
<https://www.cs.helsinki.fi/u/vihavain/k10/okk/minipascal/minipascalsyntax.html>

Syntax in recursive descent order

<program> ::= program <identifier> ; <block> .

<block> ::= <variable declaration part>

<procedure declaration part>

<statement part>

<variable declaration part> ::= <empty> | var <variable declaration> ; { <variable declaration> ; }

<variable declaration> ::= <identifier> { , <identifier> } : <type>

<type> ::= <simple type> | <array type>

<array type> ::= array [<index range>] of <simple type>

<index range> ::= <integer constant> .. <integer constant>

<simple type> ::= <type identifier>

<type identifier> ::= <identifier>

...

[Voltar](#)

Exemplo aplicado da definição de uma LP –
Syntax of Mini-Pascal (Welsh & McKeag, 1980) –

<https://www.cs.helsinki.fi/u/vihavain/k10/okk/minipascal/minipascalsyntax.html>

(continuação)

Lexical grammar

<constant> ::= <integer constant> | <character constant> | <constant identifier>
<constant identifier> ::= <identifier>
<identifier> ::= <letter> { <letter or digit> }
<letter or digit> ::= <letter> | <digit>
<integer constant> ::= <digit> { <digit> }
<character constant> ::= < any character other than ' >' | '''
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B |
C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<special symbol> ::= + | - | * | = | < > | < | > | <= | >= | (|) | [|] | := | . | , | ; | : | .. | div | or | and |
not | if | then | else | of | while | do | begin | end | read | write | var | array |
procedure | program
<predefined identifier> ::= integer | Boolean | true | false

[Voltar](#)

Introdução

- **Linguagem de Programação:**
 - é uma Linguagem Formal
 - conjunto de palavras sobre um alfabeto
 - “Qualquer notação para descrição de algoritmos e estruturas de dados”. (Pratt)
 - “Uma linguagem que tem por objetivo expressar programas de computador e é capaz de expressar qualquer programa de computador”. (MacLennan)

Introdução...

- **Requisitos obrigatórios para uma linguagem ser considerada LP:**
 - Ser universal
 - Poder ser implementada em um computador
- **Requisitos desejáveis para uma linguagem poder ser considerada LP:**
 - Ser natural
 - Possuir implementação eficiente

Introdução...

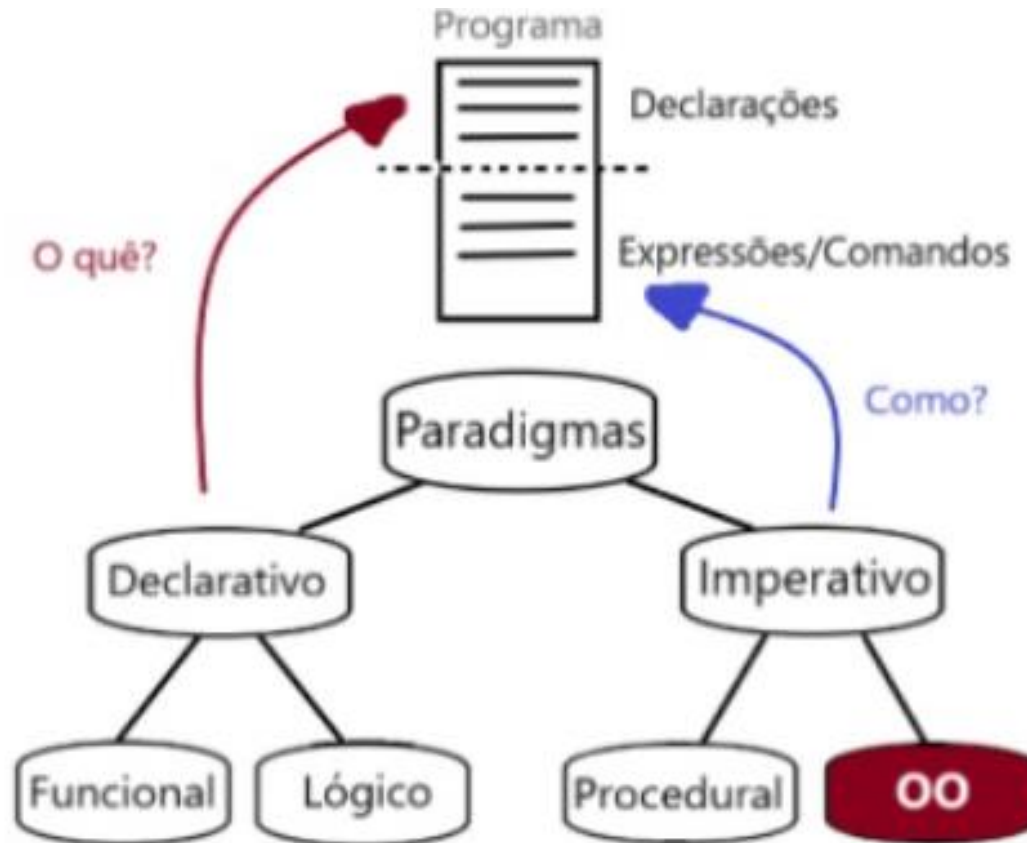
- Sintaxe e Semântica:
 - **Sintaxe** \Leftrightarrow **forma**: maneira como expressões, comandos e declarações são combinados para formar programas
 - **Semântica** \Leftrightarrow **significado**: como os programas se comportarão quando executados nos computadores

Introdução...

- Paradigmas
 - Conjunto de entidades reunidas por afinidade sob determinadas regras de uso dos recursos disponíveis
- **Paradigmas de LPs:**
 - Conjunto modelo dos conceitos individuais das LPs agrupados segundo “afinidade” dos projetos e estilos de programação das LPs: declarativo (funcional e lógico) e imperativo (procedural e OO)

Introdução...

- Paradigmas de LPs:



Introdução...

- Por que estudar LPs?
 - Melhora a compreensão da LP utilizada
 - Aumenta o vocabulário de construções de programação úteis
 - Permite uma melhor escolha da LP
 - Facilita o aprendizado de uma nova LP
 - Facilita o projeto de uma nova LP

Introdução...

- Linguagens de Programação ocupam uma posição central em ciência da computação, pois se relacionam com:
 - Bancos de Dados
 - Recuperação de Informações
 - Interação Humano-Computador
 - Sistemas Operacionais
 - Arquitetura de Computadores
 - Redes de Computadores
 - Inteligência Artificial

Valores

- O que é um valor?

Qualquer entidade que existe durante uma computação, ou seja, qualquer coisa que pode ser avaliada, armazenada, passada como parâmetro para uma função, etc.

- Por que estudar valores?

Dados são a matéria-prima da computação.

Valores...

- Como estudar valores?
 - Agrupa-los em tipos.
- Tipos: especificação da classe de valores que podem ser associados à variável, bem como das operações que podem ser usadas para criar, acessar e modificar estes valores.
- LPs, em geral, proveem um conjunto de tipos primitivos e mecanismos para estruturar tipos compostos. Algumas oferecem a opção de tipos recursivos.

Valores...

- Tipos primitivos (tipos embutidos)
 - Valores atômicos
 - Indicam a área de aplicação da LP
 - LPs costumam dar nomes diferentes a seus tipos primitivos
 - Tipos primitivos mais comuns:
 - Lógico = $\{false, true\}$
 - Inteiro = $\{..., -2, -1, 0, +1, +2, ...\}$
 - Real = $\{..., -1.0, ..., 0, ..., +1.0, ...\}$
 - Caractere = $\{..., 'a', 'b', ..., 'z', ...\}$
 - Obs: os tipos inteiro, real e caractere são definidos pela implementação da LP.

Valores...

- A representação subjacente fica invisível ao programador \Rightarrow maior legibilidade e portabilidade dos programas
- Algumas LPs permitem a definição de um novo tipo primitivo, através da enumeração de seus valores \Rightarrow *tipos enumerados*

Valores...

- Exemplo 1: PASCAL

```
type dia = (domingo, segunda, terca, quarta, quinta, sexta, sabado);  
var hoje, amanha, aniversario: dia;
```

```
hoje := segunda;  
amanha := succ(hoje);  
aniversario := amanha;
```

- Exemplo 2: C

```
enum dia_semana (domingo = 1, segunda, terca, quarta, quinta, sexta,  
sabado);
```

- Algumas LPs também permitem a definição de um subconjunto de um tipo já existente, através da definição de um intervalo.
- Exemplo: **type** indice = 1..100;

Valores...

- Tipos compostos (tipos estruturados)
 - São compostos a partir de tipos simples
 - Grande variedade: tuplas, registros, variantes, uniões, arranjos, strings, listas, árvores, arquivos, relações, etc
 - Mecanismos básicos de estruturação:

A) Produto cartesiano

- O produto cartesiano de n conjuntos S_1, S_2, \dots, S_n , denotado por $S_1 \times S_2 \times \dots \times S_n$, é um conjunto cujos elementos são n -tuplas ordenadas (s_1, s_2, \dots, s_n) , onde $s_i \in S_i$.
- Nomes simbólicos: tuplas, registros, estruturas.
- $\#(S_1 \times S_2 \times \dots \times S_n) = \#S_1 \times \dots \times \#S_n$

Valores...

- Exemplo 1: Registro em Pascal

```
type pessoa = record
    nome: string[20];
    idade: integer;
    altura: real;
end;
```

- Exemplo 2: Tupla em ML

```
type pessoa = string * int * real OU
```

```
type pessoa = {nome: string, idade: int, altura: real }
```

- Caso especial: $n = 0$

- Não é um conjunto vazio, mas um conjunto contendo uma tupla sem elementos.
- Ex.: **unit** em ML e **void** em Algol-68 e C.

Valores...

B) União discriminada

- É um mecanismo de estruturação que especifica que uma escolha será feita dentre diferentes estruturas alternativas. Cada estrutura alternativa é chamada *variante*.
- É diferente da união de conjuntos
Seja $T = \{a, b\}$
 $T \cup T = \{a, b\} = T$
 $T + T = \{\text{esq } a, \text{esq } b, \text{dir } a, \text{dir } b\} \neq T$
- Constitui a base de registros variantes, uniões, construções de ML e tipos algébricos de Miranda
- $\#(S_1 + S_2 + \dots + S_n) = \#S_1 + \dots + \#S_n$

Valores...

- Exemplo: Pascal

type item = **record**

 preco: real;

case disponivel: boolean **of**

 true: (quantidade: integer; local: string);

 false: (mes_entrega:1..12)

end;



Tags(discriminantes) são valores e podem gerar insegurança

⇒ Permite acessar um campo mesmo que ele não exista ⇒
erro de execução

⇒ Atribuição de um novo valor ao tag possui o efeito colateral
de destruir um campo e criar um novo com valor indefinido. Logo,
sugere-se utilizar comando **case**.

Valores...

C) Mapeamento

- É uma função de um conjunto finito de valores S em valores de um tipo T
- Um arranjo representa um mapeamento finito
- Muitas LPs possuem arranjos multidimensionais
- O conjunto do índice deve ser discreto
- $\#(S \rightarrow T) = (\#T)^{\#S}$
- Exemplo (Pascal): `type matriz = array [1..10, 0..20] of real;`
- Abstrações de funções representam um outro tipo de mapeamento
- Abstração de função \neq função matemática
 - Abstração de função implementa uma função através de um algoritmo
 - Ela pode acessar e alterar valores de variáveis não-locais
- Exemplo (Pascal):
`function quadrado (x: real): real
begin quadrado := x * x; end;`

Valores...

D) Sequência

- Uma sequência consiste em um número arbitrário de ocorrências de itens de dados de um determinado tipo T
- Propriedade: deixa em aberto o número de ocorrências de um componente
- Exemplos: strings (que também podem ser implementados como arranjos de caracteres) e arquivos sequenciais
- Não há um consenso com relação à classificação de strings: é um valor primitivo ou composto?
- Operações usuais: concatenação, seleção do primeiro elemento, fatia (substring), ordenação lexicográfica, etc

Valores...

E) Conjunto potência

- É o tipo de variáveis cujo valor pode ser qualquer subconjunto de um conjunto de elementos de um determinado tipo T, o qual é chamado tipo base
- Representam conjuntos \Rightarrow operações típicas de conjuntos
- $\#(\wp T) = 2^{\#T}$
- Exemplo (Pascal):

```
type ingrediente = (feijao, arroz, alface,  
                    cenoura, couve, cebola);
```

```
var salada, sobras: set of ingrediente;
```

```
    sobras := [alface];
```

```
    salada := [cenoura..cebola];
```

```
    if not feijao in sobras
```

```
        then salada := salada + sobras;
```


Valores...

F) Recursão

- Um tipo de dados recursivo T pode ter componentes que pertençam ao próprio tipo T
- Permite definir agregados cujo tamanho pode crescer arbitrariamente e cuja estrutura pode ter complexidade arbitrária
⇒ pode ser implementado por ponteiros
- Algumas LPs permitem a definição de tipos recursivos diretamente
- A cardinalidade de um tipo recursivo é infinita
- Exemplo (Pascal):

```
type ref_arvore = ^nodo_arvore;  
    nodo_arvore = record  
        info: char;  
        esq, dir: ref_arvore;  
    end;
```

Valores...

- Sistema de tipos
 - Tipagem estática e dinâmica
 - Para evitar operações sem sentido, uma implementação de LP deve realizar uma verificação de tipos sobre os operandos
 - Quando realizar essa verificação?

 - Linguagem tipada estaticamente: toda variável e parâmetro possui um tipo fixo determinado pelo programador \Rightarrow verificação em tempo de compilação

Valores...

- Linguagem tipada dinamicamente: somente os valores possuem um tipo fixo, ou seja, variáveis e parâmetros podem assumir valores de tipos diferentes, durante a execução \Rightarrow verificação em tempo de execução
- Tipagem dinâmica X estática
 - A tipagem dinâmica torna mais lenta a execução de um programa
 - A tipagem estática é mais segura
 - A tipagem dinâmica é mais flexível. Exemplo:

Valores...

- Exemplo:

```
procedimento leliteral (var: item);  
  inicio  
    ler um string;  
    se o string representa um literal inteiro  
    entao item := valor numerico do string  
    senao item := proprio string  
  fim
```

Obs: o código acima permitiria, por exemplo, a leitura de um mês como 2 ou FEV.

Valores...

– Equivalência de Tipos

- **Equivalência estrutural:** duas variáveis têm tipos compatíveis se possuem a mesma estrutura
- **Equivalência de nomes:** duas variáveis têm tipos compatíveis se possuem o mesmo nome de tipo, definido pelo usuário ou primitivo, ou se aparecem na mesma declaração
- Equivalência estrutural X equivalência de nomes
 - Equivalência de nomes se aproxima mais ao conceito de tipos abstratos de dados
 - Equivalência de nomes é mais fácil de implementar

Valores...

- Exemplo:

```
type t = array [1..20] of integer;  
var a, b: array [1..20] of integer;  
    c   : array [1..20] of integer;  
    d   : t;  
    e, f : record a: integer; b: t; end;
```

Pela equivalência estrutural, a, b, c, d, e.b e f.b têm tipos compatíveis.

Pela equivalência de nomes, a e b, d, e.b e f.b têm tipos compatíveis, mas a e c não!

Valores...

– Princípio da Completeza de Tipo

- “Nenhuma operação deve ser arbitrariamente restringida sobre os tipos de valores envolvidos”
- Justificativa: restrições tendem a reduzir o poder de expressão de uma LP
- Utiliza-se os termos valores de primeira classe e de segunda classe para diferenciar valores que podem ser utilizados de todas as formas possíveis (avaliados, atribuídos, passados como argumentos, etc), de valores que sofrem alguma restrição

Valores...

- Expressões
 - São frases de programa que podem ser avaliadas a fim de fornecer um valor
 - A) Literal
 - Forma mais simples de expressão
 - Representa um valor fixo
 - Ex. em Pascal: 1234, 1.5, 'b'
 - B) Agregado
 - Expressão que constrói um valor composto a partir dos valores de seus componentes
 - Ex. 1 (ML): $(a * 2.0, b / 2.0)$
 - Ex. 2 (Ada): `anonovo := (y => ano + 1, m => jan, d => 1);`
 - Pascal oferece agregados apenas para conjuntos, mas para arranjos e registros a linguagem exige que se faça atribuição a cada um de seus componentes

Valores...

- C) Chamada de função
 - Calcula um resultado através da aplicação de uma abstração de função a um argumento
 - Ex. 1 (ML): (if cond then sin else cos) (x)
 - Ex. 2 (C): f (x)
 - Um operador também pode ser visto como uma função
 - Várias LPs reconhecem essa semelhança entre operadores e funções e, portanto, permitem a definição de operadores da mesma forma utilizada para se definir funções

Valores...

- D) Expressão condicional
 - Possui várias subexpressões, dentre as quais somente uma é escolhida para ser avaliada
 - Nem toda LP oferece esse recurso
 - Ex. (C): $(a > b) ? a : c$
 - Ex. (ML): `if a > b then a else b`
- E) Acesso a variáveis e constantes
 - Produz os valores das variáveis e/ou constantes denotados pelos respectivos identificadores

4- Armazenamento

- Variáveis
 - Variável: objeto que contém valor
 - Modelam objetos do mundo real que possuem um estado
 - Em geral, variáveis são utilizadas através de atribuições e têm tempo de vida curto. Contraexemplo: arquivos
 - Variáveis atualizáveis \neq variáveis da matemática.
Ex.: Na matemática não é possível realizar $x := x + 1$
 - Seja ***n*** uma variável. “O valor de ***n***” quer dizer “o conteúdo da célula denotada por ***n***”
 - Modelo abstrato de armazenamento
 - Uma memória é uma coleção de células
 - Cada célula tem um status corrente: alocada ou livre
 - Cada célula alocada tem um conteúdo corrente: valor armazenável ou indefinido

4- Armazenamento...

- Variáveis Compostas
 - Uma variável de tipo composto consiste de componentes (que são variáveis). O conteúdo desses componentes pode ser inspecionado e atualizado seletivamente

Atualização total x atualização seletiva

- Atualização seletiva = atualização de um componente de variável
- Nem toda LP oferece esse recurso. Ex.: ML

4- Armazenamento...

Variáveis arranjo

- Motivação: “quando e como um conjunto de índices de um arranjo é determinado?”
 1. Arranjo estático = em tempo de compilação
 2. Arranjo dinâmico = no momento da criação da variável. Ex. (Algol-68): `[m:n] int vetor`
 - Obs.: quando esta declaração é encontrada em tempo de execução, o vetor é alocado conforme os valores correntes de m e n; preserva este tamanho até seu desaparecimento quando da saída do seu escopo
 3. Arranjo flexível = quando são feitas atribuições à variável (durante a execução). Ex. (Algol-68): `flex [1:0] int a`
 - Obs.: O objeto referenciado por a é declarado como um vetor contendo, inicialmente, nenhum inteiro. A instrução `a := (2, 3, 49)` altera os limites para `[1:3]` e atribui valores a todos os seus elementos.

4- Armazenamento...

- Armazenáveis
 - Valores que podem ser armazenados em células simples, que não podem ser atualizados seletivamente. Ex.: PASCAL \Rightarrow valores primitivos, conjuntos, apontadores
- Tempo de vida
 - É o intervalo de tempo entre a criação e a destruição de uma variável
 - Relaciona-se à economia de memória \Rightarrow uma variável só precisa ocupar memória se estiver “viva”

4- Armazenamento...

Variáveis locais e globais

- Variável local = declarada dentro de um bloco para uso somente dentro do bloco
- Variável global = pode ser referenciada dentro de um bloco, mas não foi declarada localmente
- Uma ativação de um bloco é um intervalo de tempo durante o qual o bloco está sendo executado

4- Armazenamento...

- Ex.: considere a seguinte estrutura de programa (linguagem hipotética)

Bloco A

Bloco B

Bloco C

...

fim C

Bloco D

...

fim D

... C, D,... C, D,...

fim B

...B...

fim A

Considere a seguinte sequência de chamadas, como disposto no pseudocódigo, originalmente a partir de A: qual o tempo de vida das variáveis deste programa?

4- Armazenamento...

- Em geral, uma variável local não retém o seu conteúdo em ativações sucessivas do bloco onde foi declarada. Contra-exemplo: variável estática, pois é local mas seu tempo de vida é igual ao tempo de execução do programa inteiro

- Exemplo: C – Qual será a saída do programa abaixo?

```
int a = 1;
void f ( ) {
    int b = 1; // inicializado a cada chamada de f
    static int c = 1; // inicializado uma única vez
    cout<<"a = "<<a++
        <<"b = "<<b++
        <<"c = "<<c++<<"\n";
    c = c +2;
}
void main ( ) { while (a < 4) f ( ); }
```

4- Armazenamento...

Variáveis de heap

- Podem ser criadas e destruídas a qualquer momento e, portanto, o tempo de vida não segue um padrão
- São criadas por comandos
- São anônimas
- São acessadas através de apontadores
- Alocador = cria uma variável de heap que é acessada indiretamente, geralmente por um ponteiro
- Liberador = destrói uma variável de heap

4- Armazenamento...

- Quase todas as LPs imperativas fornecem apontadores, apesar de constituir um conceito de baixo nível, ao invés de dar suporte a tipos recursivos diretamente
- Sejam px e py dois apontadores para uma lista. A atribuição $px := py$ faz com que as duas variáveis compartilhem a lista
- E se a LP implementar o tipo lista diretamente? Como interpretar o tipo de atribuição acima?
 1. Colocar em px uma referência para a lista referenciada por py:
 - Esta interpretação envolve compartilhamento
 - Pode ser inconsistente com a forma de atribuição de registros ou arranjos
 - Fácil de ser implementada
 2. Colocar em px uma cópia completa da lista referenciada por py:
 - É uma interpretação mais natural
 - Pode ser mais consistente com a forma de atribuição de registros ou arranjos
 - A cópia de listas é de implementação custosa

4- Armazenamento...

Variáveis persistentes

- Variável transiente = o tempo de vida está limitado à ativação do programa que a criou. Ex.: variáveis locais e de heap
- Variável persistente = o tempo de vida transcende uma ativação de um programa particular. Ex.: arquivos
- Arquivos são variáveis compostas que, em geral, são utilizadas para conter grande quantidade de dados com tempo de vida longo
- Muitas LPs restringem a forma de utilização e atualização de arquivos, por questões de eficiência. Por exemplo, proíbem a atribuição de um arquivo completo
- O Princípio da Completeza de Tipos sugere que todos os tipos da LP sejam permitidos para variáveis transientes e persistentes. Assim, não haveria tipos, comandos ou procedimentos especiais para realizar operações de entrada-saída, poupando o programador de converter dados de um tipo de dados persistente para transiente e vice-versa

4- Armazenamento...

Referências “penduradas” (dangling)

- Referência pendurada é um ponteiro que aponta para uma área de memória que foi desalocada
- Principais situações em que pode ocorrer:
 - Quando uma referência a uma variável local é atribuída a uma variável com tempo de vida maior. Exemplo (LP hipotética)

```
var r: ^Integer ;  
procedure P;  
  var v: Integer;  
  begin  
    r := &v;  
  end;  
begin  
  P;  
  r^ := 1  
End
```
 - Quando uma LP possui um liberador, este poderá transformar todas as referências à variável liberada em referências penduradas

4- Armazenamento...

– Algumas soluções:

- Não permitir que uma referência a uma variável local seja atribuída a uma variável com tempo de vida maior.

Desvantagem: pode requerer verificações em tempo de execução

- Tratar todas as variáveis como variáveis de heap \Rightarrow ao fim de uma ativação de bloco as variáveis declaradas dentro dele continuam a existir enquanto houver alguma referência a elas

Desvantagem: alocação de memória é menos eficiente

4- Armazenamento...

- Comandos
 - São frases de programa que serão executadas a fim de atualizar variáveis
 - Caracterizam as LPs imperativas
 - Tipos fundamentais de comandos:
 - A. Saltos
 - Salto (*skip*) ou comando vazio (*dummy*) é a forma mais simples de comando, pois nada realiza
 - Exemplo: **if** *E* **then** *C* **else** *skip*;

4- Armazenamento...

B. Atribuições

- Em geral, possuem a forma $V := E$, onde V é uma acesso a variável, cujo valor passará a ser E
- O que significa acesso a uma variável? Em outras palavras, qual o significado de n , nos comandos `read (n);` `n := n + 1;` `write (n);` ?
 - Pode ser uma referência a uma variável em alguns contextos e o conteúdo atual da variável em outros contextos; ou
 - Pode ser sempre uma referência a uma variável, mas em alguns contextos há uma operação implícita de de-referenciação. Em algumas LPs, a de-referenciação deve ser explícita. Exemplo (ML): `n := !n + 1;`
- Alguns tipos de atribuições
 - Múltipla: $V_1 := \dots := V_n := E$
 - Simultânea: $V_1, \dots, V_n := E_1, \dots, E_n$
 - Combinada com operador binário: $V += E$

4- Armazenamento...

C. Chamadas de procedimento

- Aplica uma abstração de procedimento a alguns argumentos
- Em geral tem a forma $P(AP_1, \dots, AP_n)$, onde P determina a abstração de procedimento a ser aplicada e os parâmetros reais AP_1, \dots, AP_n determinam os argumentos a serem passados
- Um argumento pode ser um valor (resultado de uma expressão) ou o acesso a uma variável

4- Armazenamento...

D. Comandos sequenciais

- Fluxo de controle mais comum
- Em geral têm a forma $C_1; C_2; \dots ; C_n$, onde o comando C_1 é executado antes de C_2 , e assim por diante
- Está disponível em toda LP imperativa

4- Armazenamento...

E. Comandos colaterais

- Comandos são executados sem uma ordem definida
- Em geral têm a forma C_1, C_2, \dots, C_n
- Comandos colaterais são não-determinísticos:
 - Uma computação é determinística se for possível prever com precisão a sequência de passos que será executada. Caso contrário, ela é não determinística
 - Exemplo: $x := 5, x := x + 1;$
- Mas um comando colateral pode ser *efetivamente determinístico* se nenhum subcomando utilizar uma variável atualizada por outro

4- Armazenamento...

F. Comandos condicionais

- Possui vários subcomandos, dentre os quais somente um é escolhido para ser executado
- O comando *if* é o mais simples e pode ser encontrado em toda LP imperativa
- O comando *if* não-determinístico pode existir e ser útil, por exemplo e principalmente, para programação concorrente
- Existem também comandos condicionais com escolha baseada em valores diferentes de lógicos

4- Armazenamento...

G. Comandos iterativos

- Um comando iterativo (*loop*) tem um subcomando (corpo do *loop*) que será executado repetidamente e, em geral, algum tipo de sentença que determina quando a iteração deverá parar
- Comandos iterativos podem ser:
 - Indefinidos: não se conhece o número de iterações. Exemplo: `while`
 - Definidos: é possível definir o número de iterações. Caracteriza-se pelo uso de variável de controle. O tratamento de variáveis de controle varia de linguagem para linguagem. Exemplo: `for`
- Questionamentos:
 1. Qual o valor da variável de controle após o término da repetição?
 2. Qual o valor da variável de controle após um salto para fora da repetição?
 3. O que acontece quando o corpo do *loop* possui uma atribuição para a variável de controle?

4- Armazenamento...

- Expressões com efeitos colaterais
 - São expressões que ao serem avaliadas causam o efeito colateral de atualizar variáveis

Expressões de comandos

- Permitem utilizar o poder da atribuição e da iteração no cálculo de resultados de funções

4- Armazenamento...

- Exemplo (LP hipotética)

```
var p: real;  
    i: integer;
```

```
begin  
  p := a[n];  
  for i := n-1 downto 0 do  
    p := p * x + a[i];  
  yield p  
end
```

- Em Pascal, por exemplo, o corpo de uma abstração de função é uma expressão de comando
- Qualquer tipo de expressão de comando introduz a possibilidade de efeitos colaterais

4- Armazenamento...

- Exemplo: uma função que lê um caractere de um arquivo e retorna o caractere lido, tem um efeito colateral sobre o arquivo

E	if getchar (f) = 'F' then
R	sexo := feminino
R	else if getchar (f) = 'M' then
O	sexo := masculino

4- Armazenamento...

Linguagens orientadas para expressões

- Uma linguagem orientada para expressões é uma linguagem imperativa que elimina todas as distinções entre expressões e comandos. A avaliação de expressões retorna um valor e, em geral, possui o efeito colateral de atualizar o valor da variável
- Por exemplo, é possível escrever $V' := (V := E)$
- Exemplos de LPs: Algol-68 e C
- Vantagem: maior simplicidade e uniformidade
- Desvantagem: encoraja o uso de efeitos colaterais, o que pode gerar um estilo de programação crítico.

Exemplo: `while (ch := getchar (f)) <> '*' do write (ch)`

5- Amarrações

- Programas envolvem entidades, como subprogramas, variáveis e comandos
- Entidades têm propriedades ou atributos. Por exemplo, uma variável tem um nome, escopo, tempo de vida, valor, tipo, uma área de memória onde seu valor é guardado
- Amarração = especificação de alguns dos atributos de uma entidade
- Amarração estática = estabelecida antes da execução do programa e não pode ser mudada depois
- Amarração dinâmica = estabelecida durante a execução do programa e pode ser mudada, de acordo com as regras da LP

5- Amarrações...

- Amarrações e ambientes
 - Muitas LPs permitem que um dado identificador / possa ser declarado em várias partes do programa, possivelmente denotando entidades diferentes \Rightarrow a interpretação de uma expressão ou comando contendo uma ocorrência de / dependerá do contexto
 - Uma declaração produz uma associação ou amarração entre o identificador declarado e a entidade que ele irá denotar
 - Um ambiente é um conjunto de amarrações. Cada expressão e comando é interpretado em um ambiente particular

5- Amarrações...

- Denotáveis
 - Denotáveis = entidades que podem ser denotadas por identificadores
 - Exemplo: em Pascal, os denotáveis e os tipos de declarações em que podem ser amarrados são:
 - Valores primitivos e strings – definições de constantes
 - Referências a variáveis – declarações de variáveis
 - Abstrações de procedimentos e funções – definições de procedimentos e funções
 - Tipos – definições de tipos

5- Amarrações...

- Escopo
 - Trecho do programa onde uma declaração é efetiva, ou seja, um identificador é conhecido e pode ser usado

Estrutura de bloco

- Bloco = qualquer sentença de programa que delimita o escopo de quaisquer declarações que ele possa conter
- Estrutura de bloco = relacionamento textual entre blocos

5- Amarrações...

- Tipos de estruturas de bloco:
 - A. Monolítica: o programa inteiro é um único bloco



O escopo de toda declaração é o programa inteiro



Todas as declarações devem estar agrupadas em um único ponto do programa



Todas as entidades declaradas devem ter identificadores distintos

5- Amarrações...

- B. Nivelada (“flat”): o programa é particionado em blocos distintos



Uma variável pode ser declarada dentro de uma sub rotina, sendo, portanto, local à sub rotina

- C. Aninhada (“nest”): cada bloco pode estar aninhado dentro de outro bloco



Um bloco pode ser posicionado onde for conveniente e identificadores podem ser declarados dentro dele

5- Amarrações...

Escopo e visibilidade

- Ocorrência de amarração = ocorrência de um identificador no ponto em que é declarado
- Ocorrência de aplicação = ocorrência de identificador que denota um entidade
- Quando um programa contém mais de um bloco, um mesmo identificador pode ser declarado em blocos diferentes, denotando entidades diferentes
- O que acontece quando um mesmo identificador é declarado em dois blocos aninhados? A declaração mais externa é dita ser invisível ou escondida para a declaração mais interna

5- Amarrações...

Amarração estática e dinâmica

- Amarração estática ou escopo estático: o corpo da função é avaliado no ambiente de definição da função



Pode-se determinar em tempo de compilação a qual ocorrência de amarração corresponde uma dada ocorrência de aplicação de um identificador

- Amarração dinâmica ou escopo dinâmico: o corpo da função é avaliado no ambiente de chamada da função



Deve-se determinar em tempo de execução a qual ocorrência de amarração corresponde uma dada ocorrência de aplicação de um identificador

5- Amarrações...

- A amarração dinâmica não combina com tipagem estática
- Regras de escopo dinâmicas são fáceis de implementar, mas apresentam desvantagens dos pontos de vista da disciplina de programação e eficiência de implementação

- Exemplo 1:

```
const a = 2;  
function aumentado (d: Integer);  
begin  
    aumentado := d * a;  
end;
```

```
procedure ...;  
const a = 3;  
begin  
    ... aumentado(h) ...  
end;
```

```
begin  
    ... aumentado (h) ...  
end
```


5- Amarrações...

- Exemplo 2:
program dinamico;

var r: real;

procedure show; begin write (r : 5 : 3) end;

procedure small;
var r: real
begin
r := 0.125; show
end;

begin
r := 0.25;
show; small; writeln;
end.

5- Amarrações...

- Declarações
 - Uma declaração é uma frase de programa utilizada para elaborar amarrações
 - Declarações podem ser explícitas ou implícitas. Por exemplo, em Fortran, toda variável não declarada cujo nome comece com a letra I denota um inteiro
 - Objetivos de declarações:
 - Escolha da representação de armazenamento – no caso da declaração fornecer tipo de dado
 - Gerenciamento de memória – a partir do tempo de vida de objetos
 - Definição de operações que podem ser realizadas
 - Permitir verificação estática de tipos

5- Amarrações...

Definições

- Definição = uma declaração simples cujo único efeito é produzir amarrações
- Exemplo 1: Em Pascal, uma definição de constante amarra um identificador a um valor determinado em tempo de compilação
- Exemplo 2: Em ML, uma definição de valor amarra um valor a um identificador, possivelmente em tempo de execução
- Exemplo 3: Em Pascal, uma definição de procedimento amarra um identificador a uma abstração de procedimento e uma definição de função amarra um identificador a uma abstração de função

5- Amarrações...

Declarações de tipos

- Definição de tipo = serve somente para amarrar um identificador a um tipo existente. Encontrado em LPs com equivalência de tipos estrutural
- Declaração de novo tipo = cria um novo tipo distinto. Adequado para equivalência de tipos de nome
- Exemplo: Pascal

type pessoa = record

nome: string[30];

idade: integer;

altura: real;

end;

A elaboração dessa definição de tipo cria um novo tipo string anônimo

5- Amarrações...

Declarações de variáveis

- Definição de variável – serve somente para amarrar um identificador a uma variável já existente
- Declaração de variável – cria uma nova variável distinta
- Exemplo: ML

```
val count = ref 0 // cria uma nova variável inteira  
                // com valor inicial 0
```

```
val pop = populacao sub estado  
                // amarra pop a uma variável já  
                // existente
```

- Exemplo: C++

```
int x = 2;          // cria uma nova variável inteira 'x'  
int &y = x;          // define 'y' como novo nome para 'x'
```

5- Amarrações...

Declarações colaterais

- Podem ser escritas na forma *D1 **and** D2*. O efeito é elaborar as subdeclarações *D1* e *D2* independentemente e combinar as amarrações produzidas. Nenhuma das subdeclarações pode usar um identificador declarado em outra subdeclaração
- Não são muito comuns
- Exemplo: ML
 - val pi = 3.14159**
 - and sin = fn (x: real) ⇒...**
 - and cos = fn (x: real) ⇒...**

5- Amarrações...

Declarações sequenciais

- Podem ser escritas na forma $D1 ; D2$. O efeito é elaborar a subdeclaração $D1$ seguida de $D2$, permitindo que as amarrações produzidas por $D1$ possam ser utilizadas em $D2$
- É o tipo mais comum

Declarações recursivas

- Utiliza amarrações que ela mesma produz
- Nem toda LP a suporta. As LPs mais modernas a suportam, mas, em geral, restringem as declarações recursivas a definições de tipos, procedimentos e funções
- Exemplo: Pascal – uma sequência de definições de tipos, procedimentos e funções é sempre tratada automaticamente como recursiva. Definições de constantes e declarações de variáveis são sempre tratadas como não recursivas

5- Amarrações...

- Blocos

- Comandos de bloco

- Comando de bloco = um comando contendo uma declaração que produz amarrações que serão usadas somente para executar o comando

LET pi = 3.14 IN write (pi);

- Apesar de um comando de bloco se comportar como qualquer outro comando, em Pascal não é permitido utilizá-lo com a mesma liberdade com que se usa comandos simples \Rightarrow o programador não pode colocar as declarações espalhadas pelo código, onde seriam necessárias

5- Amarrações...

Expressões de bloco

- Expressão de bloco = uma expressão contendo uma declaração que produz amarrações que serão usadas somente para avaliar a expressão

LET x: int = 3 IN x + 1;

Princípio de qualificação

- Princípio de qualificação = é possível incluir um bloco em qualquer classe sintática, desde que as sentenças desta classe especifiquem algum tipo de computação

5- Amarrações...

Declarações de bloco

- Declaração de bloco = um bloco contendo uma declaração local que produz amarrações que serão usadas somente para elaborar a declaração de bloco

LET c: const int = 8 IN V: array[1..c] of int;

- O uso de uma declaração de bloco permite manter pequena a interface do módulo, limitando o número de amarrações visíveis externamente ao módulo
- Essa construção dá suporte para o conceito de encapsulamento: um módulo grande pode declarar várias entidades, mas somente algumas delas podem ser exportadas

6- Abstração

- Abstração é o processo de identificar as qualidades ou propriedades importantes do fenômeno que está sendo modelado
- Em programação, a abstração faz referência à distinção entre O QUE uma parte do programa faz e COMO ela é implementada
- Exemplo: as construções de uma LP são abstrações do código de máquina
- A abstração é importante para a construção de programas grandes. Por exemplo, o programador pode introduzir vários níveis de abstração, através da implementação de procedimentos

6- Abstração...

- Tipos de abstrações
 - Uma abstração é uma entidade que incorpora (personifica) uma computação
 - Exemplo: uma abstração de função incorpora uma expressão a ser avaliada

Abstrações de função

- Uma abstração de função incorpora uma expressão a ser avaliada e, quando chamada, produz um valor como resultado
- Visão do usuário: uma chamada da função irá mapear os argumentos a um resultado
- Visão do programador: uma chamada da função irá avaliar o corpo da função, tendo os parâmetros formais amarrados aos argumentos correspondentes

6- Abstração...

Def. de função em Pascal	Def. de função em ML
<pre>function pot (x: Real; n: Integer): Real; begin if (n = 1) then pot := x; else pot := x * pot (x, n-1); end;</pre>	<pre>fun pot (x: real, n: int) = if n = 1 then x else x * pot (x, n-1)</pre>

- Críticas à notação de Pascal:
 - O corpo da função sempre possui comandos
 - O retorno da função é definido através de uma atribuição a uma pseudo-variável \Rightarrow é possível chegar ao fim do corpo de uma função, sem executar uma atribuição
 - O identificador de função denota duas entidades diferentes em um mesmo escopo
 - O corpo da função é, sintaticamente, um comando, mas, semanticamente, é um tipo de expressão

6- Abstração...

- Qual o significado de uma definição de função do tipo `function I (FP1, ..., FPn) is E?`



O identificador `I` é amarrado a uma determinada abstração de função, que é uma entidade que possui a propriedade de retornar um resultado sempre que for chamada com argumentos apropriados

- É perfeitamente possível para uma LP separar os conceitos de abstração e amarração. Por exemplo, em ML, podemos escrever: `fn (x:real) ⇒ x * x * x;`

6- Abstração...

Abstrações de procedimento

- Uma abstração de procedimento incorpora um comando a ser executado e, quando chamado, irá atualizar o valor de variáveis
- Visão de usuário: uma chamada de procedimento irá atualizar variáveis, de uma maneira que será influenciada por seus argumentos
- Visão do programador: uma chamada de procedimento irá executar o corpo do procedimento, tendo os parâmetros formais amarrados aos argumentos correspondentes

6- Abstração...

- Qual o significado de uma definição de procedimento do tipo `procedure I (FP1, ..., FPn) is C?`



O identificador `I` é amarrado a uma determinada abstração de procedimento, que é uma entidade que possui a propriedade de atualizar variáveis sempre que for chamada com argumentos apropriados

Princípio da Abstração

- Princípio da abstração = é possível construir abstrações sobre qualquer classe sintática, desde que as sentenças dessa classe especifiquem algum tipo de computação

6- Abstração...

- Uma abstração de seletor é uma abstração sobre um acesso a variável. Em outras palavras, uma abstração de seletor possui um corpo, que é um acesso a variável, e uma chamada de seletor é um acesso a variável que retorna uma referência a uma variável através da avaliação do corpo de uma abstração de seletor
- Exemplo: supondo que Pascal permitisse a definição de abstrações de seletor, então poderíamos definir o seguinte:
selector primeiro (var f: fila): integer is
...{retorna uma referência ao primeiro item em f}

E escrever as seguintes chamadas:

```
l := primeiro (filaA);  
primeiro (filaA) := primeiro (filaA) – 1;
```

6- Abstração...

- Como muitas linguagens imperativas, Pascal possui seletores pré-definidos, mas não oferece recursos para aumentar este repertório
- O Princípio da Abstração ajuda a identificar formas de tornar a linguagem mais expressiva e mais regular

6- Abstração...

- Parâmetros
 - Parâmetro formal: um identificador utilizado dentro de uma função para denotar um argumento
 - Parâmetro real: uma expressão (ou outra sentença) que produz um argumento
 - Argumento: um valor que pode ser passado para uma abstração. Cada LP possui o seu próprio conjunto de valores que podem ser passados como argumentos
 - Mecanismo de passagem de parâmetro: método de associação entre parâmetros formais e reais. Há vários tipos

6- Abstração...

Mecanismos de cópia

- Um mecanismo de cópia permite que valores sejam copiados para dentro ou para fora de uma abstração, quando chamada
- Os parâmetros formais denotam variáveis locais \Rightarrow criação na entrada da abstração e destruição na saída
- Passagem de valor: na entrada da abstração, uma variável local X é criada e recebe como valor inicial o valor do argumento. Qualquer alteração de X não afeta outra variável não local
- Passagem de resultado: o argumento deverá ser (uma referência a) uma variável. Uma variável X é criada, sem valor inicial. Na saída da abstração, o valor final de X é atribuído à variável que é o argumento
- Passagem de valor-resultado: combinação dos dois mecanismos anteriores

6- Abstração...

- Exemplo: Ada

```
procedure Inclui (i: in ITEM; f: in out FILA) is  
  ...{retorna uma fila com mais um item}  
end;
```

```
procedure Retira (i: out ITEM; f: in out FILA) is  
  ...{retorna um item e a fila com menos um item}  
end;
```

- Desvantagem: a cópia de valores compostos pode ter custo alto

6- Abstração...

Mecanismos de definição

- Um mecanismo de definição permite que um parâmetro formal X seja amarrado diretamente ao argumento
- Parâmetro constante: o argumento é um valor (de primeira classe). X é amarrado ao valor do argumento durante a ativação da abstração chamada
- Parâmetro variável (ou de referência): o argumento é uma referência a uma variável \Rightarrow qualquer utilização de X é, na verdade, uma utilização indireta do argumento
- Parâmetro procedimental: o argumento é uma abstração de procedimento \Rightarrow qualquer chamada a X é, na verdade, uma chamada indireta ao argumento (que é um procedimento)
- Parâmetro funcional: o argumento é uma abstração de função \Rightarrow qualquer chamada a X é, na verdade, uma chamada indireta ao argumento (que é uma função)

6- Abstração...

- A passagem de subprogramas como parâmetros tem a vantagem de permitir que um mesmo subprograma possa executar funções diferentes. Mas pode ser obscuro

- Exemplo

```
program param (input, output)
```

```
  procedure b (function h (n: integer): integer);
  begin
    writeln(h(2));
  end;
```

```
  procedure c;
    var m: integer;
```

```
  function f(n: integer): integer;
  begin
    f := m + n;
  end;
```

```
  begin
    m := 0;
    b (f);
  end;
```

```
begin
  c;
end.
```

6- Abstração...

- Exemplo: que valor será impresso por esse programa, supondo: a) passagem de valor-resultado, e; b) passagem de variável?

```
program doido (input, output)  
var A: integer;
```

```
procedure sei_la (var x: integer);  
begin  
  x := 2; A := 0;  
end;
```

```
begin  
  A := 1;  
  sei_la (A);  
  writeln(A);  
end.
```


6- Abstração...

- Exemplo:

```
procedure confuso (var m, n: integer);  
begin  
  n := 1;  
  n := m + n;  
end;
```

Supondo $i = 4$, qual será o valor de i após a chamada **confuso** (i, i)?

- Mecanismos de definição X mecanismos de cópia
 - O mecanismo de definição possui uma semântica mais simples
 - O mecanismo de definição, em geral, é mais eficiente
 - Uma desvantagem dos parâmetros variáveis é a possibilidade de utilização de apelidos (“aliasing”), o que torna o programa mais difícil de entender
 - Em algumas situações, os parâmetros constantes e variável possibilitam um poder de expressão similar aos dos mecanismos de cópia. Assim, a **escolha** é uma **decisão importante** para o **projetista** da linguagem

6- Abstração...

O Princípio da Correspondência

- Pode-se perceber uma correspondência entre alguns mecanismos de passagem de parâmetro e alguns tipos de declarações. A diferença é que uma declaração especifica o identificador e a entidade a qual será amarrada. Uma especificação de parâmetro formal especifica somente identificador (às vezes, o seu tipo) e o argumento vem de alguma outra parte do programa (parâmetro real). Exemplo:

const x = E;	procedure P (const X: T); ...; ... P(E)
---------------------	--

6- Abstração...

- Por questões de simplicidade e regularidade, um projetista de LP pode optar por eliminar todas as diferenças entre declarações e mecanismos de passagem de parâmetros
- Princípio da correspondência: para cada forma de declaração, existe um mecanismo de passagem de parâmetro e vice-versa

6- Abstração...

- Ordem de avaliação
 - Quando uma abstração é chamada, em que momento cada parâmetro real é avaliado?
 - Avaliação prévia (ou de ordem aplicativa) – no ponto da chamada. O parâmetro real é avaliado uma única vez e o seu valor substitui cada ocorrência do parâmetro formal
 - Avaliação de ordem normal – no momento em que o argumento for realmente utilizado. O parâmetro formal é substituído pelo parâmetro real
 - Avaliação tardia – o argumento será avaliado somente na primeira vez que for utilizado

6- Abstração...

- O resultado pode variar dependendo da ordem de avaliação
- Exemplo:

```
fun cand (b1: bool, b2: bool) =  
  if b1 then b2 else false
```

Supondo $n = 0$, qual será o resultado da chamada **cand ($n > 0$, $t/n > 0.5$)**?

- Função *strict* (rigorosa): uma chamada à função só pode ser avaliada se todos os seus argumentos puderem ser avaliados
- Função *nonstrict*: uma função é dita *nonstrict* com relação a um argumento n , se uma chamada à função puder ser avaliada, mesmo que o n -ésimo argumento não possa ser avaliado
- A avaliação de ordem normal é claramente ineficiente, quando faz o mesmo argumento ser avaliado várias vezes

6- Abstração...

- Propriedade de Church-Rosser = se uma expressão pode ser avaliada, então ela pode ser avaliada através da utilização consistente da avaliação de ordem normal. Se uma avaliação pode ser realizada em ordens diferentes (misturando ordem normal e aplicativa), então todas as ordens de avaliação produzirão o mesmo resultado
- Qualquer LP que permita efeitos colaterais não possui a propriedade de Church-Rosser
- Algol-60 permite que o programador faça uma escolha entre a avaliação prévia (passagem de valor) e avaliação de ordem normal (passagem de nome)
- Basicamente, no mecanismo de passagem de nome:
 - Cada ocorrência do parâmetro é considerada como sendo textualmente substituída pelo argumento. Isso pode levar a complicações
 - A passagem de nome é poderosa, porque permite passar funções e procedimentos, além das variáveis simples e estruturadas
 - A passagem de nome pode levar a programas que são difíceis de ler

7- Encapsulamento

- Sistemas de programação de grande porte são constituídos de módulos
- Um módulo é qualquer unidade de programa que possua um nome e que possa ser implementada como uma entidade independente
- Um módulo bem projetado tem um único objetivo e possui uma interface pequena com outros módulos \Rightarrow é reutilizável
- A chave para modularidade é a abstração
- Diz-se que um módulo encapsula seus componentes. Esses componentes podem ser tipos, constantes, variáveis, procedimentos, funções, etc
- *Componentes exportáveis* de um módulo são os componentes que são visíveis externamente ao módulo. Os componentes escondidos (ocultos) são usados somente para auxiliar a implementação dos componentes exportáveis

7- Encapsulamento...

- Pacotes

- Pacotes simples

- Um pacote é especificado através de uma lista de informações declarativas. Em geral, essas informações podem ser qualquer denotável da LP, tais como tipos, constantes, variáveis, procedimentos, funções e outros pacotes
 - Um pacote pode ser visto como um conjunto encapsulado de amarrações
 - Exemplo: Ada

- `package I is D end I;`**

- `package conversao_metrica is`**
`pol_cm: constant Float := 2.54;`
`pe_cm: constant Float := 30.48;`
`jarda_cm: constant Float := 91.44;`
`milha_km: constant Float := 1.609;`
`end conversao_metrica;`

7- Encapsulamento...

Ocultamento de informação (*information hiding*)

- Em geral, um pacote contém declarações de componentes exportáveis e escondidos. A distinção, em Ada, é feita através da divisão de um pacote em duas partes:
 - declaração do pacote – declara somente os componentes exportáveis
 - corpo do pacote – contém declarações de todos os componentes escondidos, além dos corpos de procedimentos e funções exportáveis
- A declaração do pacote contém somente as informações necessárias ao usuário para que ele possa utilizar o pacote, dando condições ao compilador de realizar verificações de tipo
- Exemplo: Ada

package trig is

function sin (x: in float) return float;

function cos (x: in float) return float;

end trig;

package body trig is

pi: constant float := 3.1416;

function sin (x: in float) return float;

-- comandos para calcular o seno de x

function cos (x: in float) return float;

-- comandos para calcular o coseno de x

end trig;

7- Encapsulamento...

- Tipos abstratos
 - Considere as seguintes definições de tipos:
type racional = int * int;
type data = int * int;

Qual é o tipo da expressão (1, 12)? É possível comparar um valor do tipo **data** com um valor do tipo **racional**?
 - Quando representamos um tipo através de um outro tipo, algumas dificuldades podem ocorrer:
 - o tipo da representação pode possuir valores que não correspondem a qualquer valor do tipo desejado
 - comparações com resultados incorretos
 - os valores de um tipo podem ser confundidos com valores do outro tipo, salvo se for declarado um novo tipo
 - Um tipo abstrato é um tipo definido através de um grupo de operações
 - Tipicamente, o programador escolhe uma representação para os valores do tipo abstrato e implementa as operações em termos da representação escolhida. A representação é escondida; o módulo exporta somente o próprio tipo abstrato e suas operações

7- Encapsulamento...

- Exemplo: (Ada)

```
package tipo_turma is
  type id_aluno is integer;
  type Turma is limited private;
  procedure inclui_aluno (t: in out Turma; aluno: in id_aluno);
  procedure cria_turma (t: in out Turma; prof: in integer; sala: in integer);
private
  tam_max: constant integer := 65;
  type Turma is
    record
      sala: integer;
      professor: integer;
      tam_classe: integer range 0..tam_max := 0;
      lista_classe: array (1..tam_max) of id_aluno;
    end record;
end tipo_turma;

package body tipo_turma is
  procedure inclui_aluno (t: in out Turma; aluno: in id_aluno) is
    -- código para inclusão de um aluno na turma
  procedure cria_turma (t: in out Turma; prof: in integer; sala: in integer) is
    -- código para criar uma nova turma
end tipo_turma;
```

7- Encapsulamento...

- Com um tipo abstrato, não interessa se um dado valor do tipo possui várias representações possíveis, pois as representações estão escondidas do usuário. O que é importante é que somente propriedades desejadas dos valores são observáveis, usando as operações associadas ao tipo abstrato
- Uma representação de tipo abstrato sempre pode ser alterada, sem haver a necessidade de realizar alguma alteração externamente ao módulo
- Definir um tipo abstrato exige mais esforço
- É comum um tipo abstrato fornecer operações construtoras, para compor valores do tipo abstrato e operações destrutoras, para decompor tais valores
- Tipos abstratos são similares aos tipos pré-definidos.
Ex: turmaA: Turma;

7- Encapsulamento...

- Objetos e classes

- Objetos simples

- O termo objeto é freqüentemente utilizado para uma variável escondida em um módulo (ou o próprio módulo), sendo que esse módulo contém operações exportáveis sobre esta variável. A vantagem disso é que alterações na representação da variável não provocam alterações externas ao módulo. Em geral, a variável é uma estrutura de dados
 - Objeto tem tempo de vida, que em geral, é definido da mesma forma que para variáveis locais

7- Encapsulamento...

Classes de objetos

- Como definir classes de objetos similares?
- Exemplo: (Ada)

```
generic package tipo_turma is
  type id_aluno is integer;
  procedure inclui_aluno (aluno: in id_aluno);
  procedure cria_turma (prof: in integer; sala: in integer);
end tipo_turma;
```

```
package body tipo_turma is
  tam_max: constant integer := 65;
  type Turma is
    record
      sala: integer;
      professor: integer;
      tam_classe: integer range 0..tam_max := 0;
      lista_classe: array (1..tam_max) of id_aluno;
    end record;
```

```
procedure inclui_aluno (aluno: in id_aluno) is
  -- código para inclusão de um aluno na turma
procedure cria_turma (prof: in integer; sala: in integer) is
  -- código para criar uma nova turma
begin
  ...; -- inicializa a turma
end tipo_turma;
```

7- Encapsulamento...

A elaboração do pacote genérico anterior, simplesmente amarra ***tipo_turma*** a uma classe de objetos. Para criar objetos, deve instanciar o pacote genérico:

package turmaA is new tipo_turma;

Assim pode-se acessar esse objeto ***turmaA***:

turmaA.cria_turma(10,222);
turmaA.inclui_aluno(13892);

- Qual a diferença entre um tipo abstrato e uma classe de objetos?
 - Tipos abstratos – operações (funções e procedimentos) possuem um parâmetro a mais \Rightarrow Na chamada de uma operação de um tipo abstrato, todos os argumentos estão explícitos
 - Classes de objetos – instâncias diferentes definem procedimentos distintos, sendo que cada um deles acessa um objeto de dados distinto. Procedimentos e funções têm como que um parâmetro implícito
 - Tipos abstratos são semelhantes aos tipos pré-definidos \Rightarrow a definição de um novo tipo abstrato amplia a variedade de tipos fornecidos ao programador

7- Encapsulamento...

- Genéricos

- Abstrações genéricas

- Uma abstração genérica é uma abstração sobre uma declaração. Em outras palavras, uma abstração genérica possui um corpo que é uma declaração e uma instanciação genérica é uma declaração que produzirá amarrações através da elaboração do corpo de uma abstração genérica
 - Exemplo: (Ada) – pacotes genéricos também podem ter parâmetros

```
generic  
type Elem is private;  
tam: in positive;  
  
package tipo_pilha is  
type Pilha is private;  
procedure empilha (i: in Elem; s: in out pilha);  
procedure desempilha (i: out Elem; s: in out pilha);  
private  
type Pilha is  
record  
  dados: array (1.. tam) of Elem;  
  topo: integer range 0..tam := 0;  
end record;  
end tipo_pilha;
```

```
package body tipo_pilha is  
  procedure empilha (i: in Elem; s: in out pilha);  
    -- código para inclusão de elemento na pilha  
  procedure desempilha (i: out Elem; s: in out pilha);  
    -- código para retirar elemento da pilha  
end tipo_pilha;
```

Essa definição de tipo genérico pode ser instanciada para produzir uma pilha de inteiros ou de turmas:

```
package Pilha_Inteiros is new tipo_pilha (Elem => integer, 100);  
package Pilha_Turmas is new tipo_pilha (Elem => Turma, 60);
```


8- Sistema de Tipos

- O projeto de um verificador de tipos de uma LP é baseado nas informações sobre as construções sintáticas da linguagem, a noção de tipos e as regras de atribuição de tipos associadas às construções da linguagem
- Um sistema de tipo é uma coleção de regras para atribuição de tipos a várias partes de um programa
- Um verificador de tipos implementa um sistema de tipo

8- Sistema de Tipos...

- Monomorfismo
 - Um sistema de tipo é dito ser monomórfico, se toda constante, variável, resultado de função e parâmetro formal deve ser declarado com um tipo específico
 - Vantagem: verificação de tipo é simples e direta
 - Desvantagem: dificulta a criação de programas reutilizáveis, já que muitos algoritmos e estruturas de dados são genéricos
 - Nenhuma LP é totalmente monomórfica. Por exemplo, algumas funções e procedimentos pré-definidos da linguagem Pascal têm tipos que não podem ser expressos no sistema de tipos da linguagem:
 - Procedimento *write* – o efeito de uma chamada de procedimento *write(E)* irá depender do tipo de *E*. Na verdade, o identificador *write* denota simultaneamente vários procedimentos distintos \Rightarrow sobrecarga
 - Função *eof* – o tipo da função é *File(τ)* \rightarrow *Lógico*, sendo τ qualquer tipo \Rightarrow polimorfismo
 - A herança também pode ser encontrada de uma forma bem simples: um tipo intervalo herda todas as operações de seu pai



Pascal é inconsistente: todas as abstrações explicitamente definidas pelo programador são monomórficas, mas muitas abstrações pré-definidas são sobrecarregadas ou polimórficas

8- Sistema de Tipos...

- Sobrecarga
 - Capacidade de amarrar a um único identificador ou operador várias entidades (amarrações) simultaneamente
 - Também conhecido por *polimorfismo ad-hoc*
 - Em geral, sobrecarga só é aceitável onde cada chamada de função não é ambígua
 - Considere um identificador ou operador I que denote uma função f_1 do tipo $S_1 \rightarrow T_1$ e uma função f_2 do tipo $S_2 \rightarrow T_2$. Há dois tipos de sobrecarga:
 - *Sobrecarga independente de contexto*: requer que S_1 e S_2 sejam distintos. Para identificar a função a ser chamada, basta saber o tipo do parâmetro real

8- Sistema de Tipos...

- *Sobrecarga dependente de contexto*: requer que S_1 e S_2 sejam distintos ou que T_1 e T_2 sejam distintos. Se S_1 e S_2 são distintos, a função a ser chamada pode ser identificada pelo tipo do parâmetro. Caso contrário, se T_1 e T_2 são distintos, deve-se considerar o contexto para poder identificar a função a ser chamada. Com esse tipo de sobrecarga, torna-se possível formular expressões cujas chamadas de funções sejam ambíguas. Caberá à LP proibir expressões ambíguas.
 - Exemplo: suponhamos que o operador $/$ denote três funções distintas:
 - » Inteiro x Inteiro \rightarrow Inteiro
 - » Real x Real \rightarrow Real
 - » Inteiro x Inteiro \rightarrow Real

Considerando n uma variável do tipo Inteiro e x , do tipo Real, qual função será chamada nas expressões abaixo?

$$x := 7 / 2 \Rightarrow x = 3.5$$

$$n := 7 / 2 \Rightarrow n = 3$$

$$n := (7 / 2) / (5 / 2) \Rightarrow n = 3 / 2 = 1$$

$$x := (7 / 2) / (5 / 2) \Rightarrow x = 3 / 2 = 1.5 \text{ OU } \Rightarrow x = 3.5 / 2.5 = 1.4$$

8- Sistema de Tipos...

- Polimorfismo
 - Relaciona-se a abstrações que operam de maneira uniforme sobre valores de diferentes tipos
 - Também conhecido por *polimorfismo paramétrico*
 - ML foi a primeira LP a fornecer um sistema de tipo realmente polimórfico

8- Sistema de Tipos...

Abstrações polimórficas

- Ao invés de definir tipos específicos para o tipo de uma função, utiliza-se variáveis de tipo
- Exemplo: ML

fun segundo (x: σ , y: τ) = y

A função **segundo** é do tipo $\sigma \times \tau \rightarrow \tau$, sendo que σ e τ representam qualquer tipo. Assim, a chamada de função **segundo (13, true)** retorna **true**. A chamada de função **segundo ("José", "Silva")** retorna **"Silva"**. Mas a chamada de função **segundo (10, 9, 1990)** é inválida

8- Sistema de Tipos...

Tipos parametrizados

- Um tipo parametrizado é um tipo que tem outro(s) tipo(s) como parâmetro(s)
- Exemplo: os tipos **file**, **set** e **array** do Pascal
- Em uma LP monomórfica, somente existem tipos parametrizados pré-definidos. Por exemplo, em Pascal, o programador **não** pode escrever:

```
type par (  $\tau$  ) = record primeiro, segundo:  $\tau$  end;  
    ParInt = par (integer);  
    ParReal = par (real);
```

- Exemplo: ML

```
type  $\tau$  par =  $\tau$  *  $\tau$ 
```

Podemos utilizar essa definição de tipo para criar um par de inteiros **int par** ou um par de reais **real par**, por exemplo

8- Sistema de Tipos...

Politipos

- Um politipo é um tipo que contém uma ou mais variáveis de tipo. Por exemplo, $\sigma \times \tau \rightarrow \tau$, σ e τ são variáveis de tipo, ou seja, representam um tipo desconhecido
- Um politipo deriva uma família completa de tipos, obtida através da substituição sistemática de um tipo real por cada variável de tipo
- Um tipo que não possui variáveis de tipo é chamado de monotipo. Em linguagens monomórficas, todos os tipos são monotipos
- Um politipo também é um tipo. Em geral, o conjunto de valores de um politipo é a interseção de todos os tipos que podem ser derivados a partir dele
- Exemplo 1: considerando o politipo Lista (τ), a lista vazia é a interseção de todos os tipos Lista
- Exemplo 2: O tipo $\tau \rightarrow \tau$ é a interseção de todos os tipos Inteiro \rightarrow Inteiro, String \rightarrow String, Boolean \rightarrow Boolean, etc

8- Sistema de Tipos...

- Inferência de tipo
 - Inferência de tipo \Rightarrow o tipo de uma entidade declarada é inferido, ao invés de ser explicitamente determinado

Inferência de tipo monomórfico

- Em ML, podemos escrever:
fun par(n) = (n mod 2 = 0)

Supondo que o operador **mod** tenha tipo **integer x integer \rightarrow integer**, torna-se possível inferir que **n** tem que ser do tipo **integer** e que o tipo do retorno da função é “lógico” \Rightarrow o tipo da função **par** é **integer \rightarrow boolean**

- ML adota uma atitude *laissez-faire* com relação a tipos: o programador pode determinar ou não o tipo de uma entidade
- Desvantagem: um pequeno erro de programação pode confundir o compilador, que irá produzir mensagens de erro obscuras, ou inferir um tipo diferente daquele desejado pelo programador



é boa prática de programação determinar tipos explicitamente, mesmo que isso seja redundante

8- Sistema de Tipos...

Inferência de tipo polimórfico

- Nem sempre é possível que uma inferência de tipo produza um monotipo. Em outras palavras, às vezes será produzido um politipo

- Exemplo (ML):

fun length (l) =

case l of

nil \Rightarrow 0

| cons (h, t) \Rightarrow 1 + length (t)

O resultado dessa função é **integer** e **l** é do tipo **Lista (τ)**. Assim, **length** é do tipo **Lista (τ) \rightarrow integer**

8- Sistema de Tipos...

- Coerção
 - Coerção é um mapeamento implícito de valores de um tipo para valores de um tipo diferente. Ela é executada automaticamente, sempre que necessário
 - Considere um contexto no qual se espera um operando do tipo T , mas é fornecido um operando do tipo T' (não equivalente a T). A LP pode permitir uma coerção nesse contexto, desde que a linguagem defina um mapeamento único entre o tipo T' e o tipo T
 - LPs modernas tendem a minimizar ou eliminar a coerção, pois ela não funciona bem com sobrecarga e polimorfismo. Por exemplo, no caso de Ada, o mapeamento é feito explicitamente *Float(x)*

8- Sistema de Tipos...

- Subtipos e herança
 - Relacionada à capacidade de permitir que subtipos herdem operações de seus supertipos
 - Também conhecido por *polimorfismo de inclusão*
 - Pascal reconhece uma forma restrita de subtipo, ao permitir a definição de intervalos de qualquer tipo primitivo discreto T
 - Cada LP reconhece alguns subconjuntos de tipos como subtipos, mas não subconjuntos arbitrários. Por exemplo, nenhuma LP permite declarar uma variável que varie somente sobre os números inteiros primos

8- Sistema de Tipos...

- Uma condição necessária para S ser um subtipo de T é que $S \subseteq T$. Assim, um valor de S pode ser usado com segurança onde um valor do tipo T é esperado
- Seja U um tipo que não seja um subtipo de T , mas que tenha elementos comuns com T . Então, um valor do tipo de U pode ser usado onde se espera um valor do tipo de T , desde que o verificador de tipo, em tempo de execução, reconheça que o valor pertence também a T
- Associado a cada tipo T , há um subconjunto de operações que podem ser aplicadas a valores do tipo T . Essas operações também poderão ser aplicadas a valores de qualquer tipo S que seja subtipo de T . Podemos dizer então que S herda todas as operações associadas a T
- O termo herança vem da Programação Orientada a Objetos

9- Sequenciadores

- Um sequenciador é uma construção que altera o fluxo de controle normal (uma única entrada e uma única saída)



Permite a programação de fluxos de controle mais genéricos

- Exemplos
 - Saltos (*jumps*)
 - Escapes (*escapes*)
 - Exceções (*exceptions*)

9- Sequenciadores...

- Saltos (*Jumps*)
 - Comandos simples, seqüenciais, condicionais e iterativos possuem uma única entrada e uma única saída



Qualquer comando formado pela composição de comandos simples, seqüenciais, condicionais e iterativos também possui uma única entrada e uma única saída

- Um salto (*jump*) é uma transferência explícita de controle de um ponto do programa para outro
- A forma mais comum de saltos é: **goto L**, sendo que **L** é um label que denota um ponto do programa. Para amarrar um label a um ponto do programa escreve-se algo da forma: **L: C**

9- Sequenciadores...

– Exemplo

Formas de controle básicas	Seqüência de execução	Com <i>jump</i>
Composição (Seqüencial)	S0 S1 S2 S3	S0 goto L1 L2: S2 goto L3 L1: S1 goto L2 L3: S3
Alternativa	S0 S1 S3 <u>ou</u> S0 S2 S3	S0 if A = 0 then goto L1 S1 goto L2 L1: S2 L2: S3
Iteração	S0 S2 <u>ou</u> S0 S1 S2 <u>ou</u> S0 S1 S1 S2...	S0 L1: if A = 0 then goto L2 S1 goto L1 L2: S2

9- Sequenciadores...

- O uso irrestrito de saltos permite que um comando possua várias entradas e saídas
 - Vantagens
 - fácil de usar (em programas pequenos)
 - completamente genérico
 - eficiente
 - Desvantagem: falta de legibilidade (“programas espaguete”)

9- Sequenciadores...

- Muitas LPs impõem restrições à utilização de saltos: em geral, o escopo de um label é o bloco mais próximo que contém sua ocorrência de amarração
- Pascal impõe as seguintes restrições a saltos: um comando **goto** pode ser usado dentro de um bloco ou para saltar de um bloco interno para o externo, mas não pode ser usado para saltar de um bloco externo para um interno. Em particular, apesar de permitir saltar para fora de um procedimento, não é permitido saltar para dentro

< VISUALIZAR EXEMPLO NO QUADRO >

9- Sequenciadores...

- Sequenciadores alteram significativamente a semântica de uma LP. Por exemplo, o comando sequencial C1; C2; é executado através da execução de C1, seguida da execução de C2. Mas se houver um sequenciador em C1, não se sabe se C2 será executado
- Problema relacionado ao salto para fora de um bloco: eliminar variáveis locais ao bloco; abandonar uma ativação de procedimento
- E se o procedimento for recursivo? Qual(is) das ativações deverá(ao) ser abandonada(s)?



Labels não podem denotar simplesmente um ponto do programa, mas um ponto do programa de uma determinada ativação do procedimento no qual se encontra

- Saltos, que parecem ser simples, introduzem uma complexidade indesejável à semântica de uma LP

9- Sequenciadores...

- Escapes (*Escapes*)
 - Um escape é um sequenciador que termina a execução de um comando textualmente fechado. Em outras palavras, é uma transferência direta de controle para a saída do comando



Permite a programação de fluxos de controle com uma única entrada e múltiplas saídas

9- Sequenciadores...

— Exemplo: C

```
while (x <= 100) {  
    if (x < 0) {  
        cout << "Erro: valor negativo";  
        break;  
    }  
    ... cin >> x;  
}
```

```
do {  
    cin >> x;  
    if (x < 0) {  
        cout << "Erro: valor negativo";  
        continue;  
    }  
    ...  
} while (x <= 100);
```

9- Sequenciadores...

- Exemplo: Ada

Por default, **exit** finaliza o loop mais próximo (interno). No caso de querer terminar um loop mais externo, é necessário dar um nome a ele

< VISUALIZAR EXEMPLO NO QUADRO >

- Um tipo importante de escape é o **return**. Em geral, LPs impedem que escapes transfiram controle para fora de procedimentos. Isso evita que escapes sejam capazes de terminar ativações de procedimentos
- O tipo escape mais drástico é o **halt**, que termina o programa inteiro

9- Sequenciadores...

- Exceções (*Exceptions*)
 - O que acontece quando uma operação aritmética causa um *overflow*? E uma divisão por zero? E uma operação de entrada/saída não puder ser realizada?
 - Observação: quando essas **condições excepcionais** ocorrem, o programa não pode continuar normalmente
 - Alternativa 1: o programa interrompe sua execução \Rightarrow inflexibilidade que compromete a modularidade e a falta de robustez
 - Alternativa 2: o controle é transferido para um tratador de exceções, uma parte do programa que especifica o que deve ser feito em condições excepcionais \Rightarrow robustez

9- Sequenciadores...

- Uma técnica comum para tratar condições de exceção é dar a cada abstração P , um código de resultado, que pode ser uma variável global, um parâmetro de resultado ou um resultado de função. Quando chamado, P seta seu código de resultado para indicar se tudo deu certo ou, caso contrário, qual condição excepcional foi detectada
 - Desvantagens:
 - o programa pode ficar confuso, devido ao número de testes
 - o programador pode omitir um teste de resultado

9- Sequenciadores...

- Um outra técnica é associar um tratador a cada exceção específica
- Uma exceção é uma indicação (sinal) de que surgiu uma condição excepcional
- PL/I foi a primeira LP a incorporar alguma forma geral de tratamento de exceções → muito confuso
- Ada incorporou esse conceito na linguagem com mais sucesso:
 - um tratador para qualquer exceção e é associado a um comando C. Se C (direta ou indiretamente) “dispara” e, a execução de C é abandonada e o controle é transferido para o tratador
- Pode-se associar diferentes tratadores para a mesma exceção de diferentes comandos
- Pode-se associar diferentes tratadores para diferentes exceções de um mesmo comando

9- Sequenciadores...

– Exemplo em Java:

```
try {  
    // código a ser executado  
}  
catch (ClasseDeExceção instânciaDaExceção) {  
    // tratamento da exceção  
}  
finally {  
    // código a ser executado mesmo que uma exceção seja  
    lançada  
}
```

9- Sequenciadores...

```
01. public class ExemploDeExcecao {
02.     public static void main(String[] args) {
03.         String var = "ABC";
04.         try {
05.             Integer i = new Integer(var);
06.             System.out.println("A variável i vale " + i);
07.         } catch (NumberFormatException nfe) {
08.             System.out.println("Não é possível atribuir a
09. string " + var + " a um Objeto Inteiro.\n" + "A seguinte
10. mensagem foi retornada:\n\n" + nfe.getMessage());
11.         }
12.     }
13. }
```

- O código acima apresentará algo como:
Não é possível atribuir a string ABC a um Objeto Inteiro.
A seguinte mensagem foi retornada:
For input string: "ABC"
- Perceba que a linha *System.out.println("A variável i vale " + i)* não foi executada, pois houve um erro na linha anterior. Portanto, apenas a mensagem de tratamento do erro *NumberFormatException* foi impressa na tela.

Conceitos e Paradigmas de
Linguagens de Programação

Considerações Finais

Professor Marco Rodrigo Costa

“Qual Linguagem de Programação devo utilizar?”

- Critérios **errados**
 - Fanatismo (“... é brilhante!”)
 - Preconceito (“... é um lixo!”)
 - Inércia (“... é difícil de aprender!”)
 - Medo de mudar (“... conhecemos melhor esta!”)
 - Modismo (“... todo mundo está usando!”)
 - Pressão comercial/mercado (“... Empresa ‘X’ utiliza!”)
 - Conformismo/Segurança (“... Ninguém foi demitido...”)

“Qual Linguagem de Programação devo utilizar?”...

- Critérios **adequados** (devem ser técnicos e financeiros)
 - Escala → programação de grande porte...?
 - Modular → visão do usuário e programador...?
 - Reusável → efetivo reuso de código...?
 - Portável → sofre grandes alterações em diferentes plataformas...?
 - Nível → maior ou menor abstração (bits, pointers...)...?
 - Confiável → erros são detectados e corrigidos o quanto antes..?

“Qual Linguagem de Programação devo utilizar?”...

- Critérios **adequados** (devem ser técnicos e financeiros)...
 - Eficiente → roda rápido...?
 - Legível → sintaxe favorável...?
 - Modelagem dos dados → tipos e operações favorecem as entidades implementadas...?
 - Modelagem dos processos → construções favorecem processos concorrentes, sequenciais...?
 - Disponibilidade de compiladores e outras ferramentas
 - Familiaridade → tempo de aprendizado compensa...?

Tendências em LPs...

- Proliferação de Linguagens de Programação
 - Variações de sintaxe e semântica
 - Concorrência de projetistas e implementadores
- Ampliação de Paradigmas de Programação
 - Soluções de novos tipos de problemas
- Maior nível de abstração
- Variedade de comunicação com os programas (*inputs e outputs*)