




Laboratório de Engenharia de Software

Assertivas

Arndt von Staa
LES/DI/PUC-Rio
Março 2015

Especificação



Laboratório de Engenharia de Software

- Objetivos dessa aula
 - Estabelecer uma notação rigorosa para assertivas
 - A notação visa apoiar
 - a argumentação da corretude de programas
 - a implementação de assertivas executáveis, possivelmente usando geração de código
- Justificativa
 - assertivas tornam mais formais as especificações do código
 - são a base para a argumentação (ou prova) da corretude de programas
 - assertivas executáveis são a base para o desenvolvimento de programas auto verificáveis (*self-checking*)
 - assertivas reduzem significativamente o retrabalho inútil
- Texto
 - Pezzè, M.; Young, M.; *Teste e Análise de Software*; Porto Alegre, RS: Bookman; 2008, capítulo 7
 - Staa, A.v.; *Programação Modular*; Rio de Janeiro: Campus/Elsevier; 2000, capítulo 13

Mar 2015Arndt von Staa © LES/DI/PUC-Rio2

Assertivas, exemplos



- Exemplos

- $\text{sqrt}(x) =: y \mid 1 - \varepsilon < y^2/x < 1 + \varepsilon$
- $\text{sqrt}(x)$ produz y tal que $1 - \varepsilon < y^2/x < 1 + \varepsilon$
 - em que, por convenção, $\varepsilon = 10^{-n}$, onde n é o número de dígitos significativos desejado

Assertivas, exemplos



- Exemplos


- em uma *lista* duplamente encadeada:

$\forall pElem \in lista : ? pElem \rightarrow pAnt \neq \text{NULL} \Rightarrow$
 $pElem \rightarrow pAnt \rightarrow pProx == pElem$

ParaTodos $pElem$ **pertencentes_a** *lista* **vale**
se $pElem \rightarrow pAnt \neq \text{NULL}$ **entao**
 $pElem \rightarrow pAnt \rightarrow pProx == pElem$

Laboratório de Engenharia de Software

Terminologia




- Em engenharia de software existem, infelizmente, vários nomes para a mesma coisa, ex.:
 - métodos, funções, subrotinas, procedures: é tudo a mesma coisa
- Assertivas e contratos são a mesma coisa
 - o termo contrato foi usado por Meyer ao definir a linguagem de programação orientada a objetos Eiffel em 1986
 - o uso de contratos levou à definição do termo *design by contract*, ou projeto dirigido por contratos, ou programação dirigida por contratos e outras
 - a base disso é verificação formal usando a lógica de Hoare (1969)
 - essa foi inspirada pelas *program annotations* descritas por Floyd (1967)
- Em várias linguagens de programação podem-se avaliar as assertivas em tempo de execução usando funções que usualmente possuem a assinatura: `bool assert (bool expression)`
- Por hábito, nascido em 1973, vou continuar a usar o termo *assertiva*

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
5

Laboratório de Engenharia de Software

O que são assertivas?




- Assertivas são **expressões condicionais** envolvendo dados e estados manipulados
 - condições: `<`, `<=`, `==`, `!=`, `>=`, `>`
 - predicados, similares a:
 - `ExisteX(conjunto)`
 - `EhTipoX(dado)`
 - expressões que resultem em um valor booleano

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
6

Laboratório de Engenharia de Software

O que são assertivas?




- Assertivas podem ser definidas em **níveis de abstração**
 - **funções** ou métodos
 - devem estar satisfeitas em **determinados pontos** do corpo da função
 - usuais são as assertivas de **entrada** e as assertivas de **saída**
 - devem estar satisfeitas ao entrar e ao retornar de funções
 - **pré** e **pós condições**
 - **classes** e módulos
 - assertivas **invariantes da classe**
 - as invariantes da classe envolvem somente atributos de um único objeto e, caso existam, os da classe (**static**)
 - assertivas **estruturais** podem envolver atributos de vários objetos e de diferentes classes
 - **programas**
 - devem estar satisfeitas para os **dados de interface**
 - arquivos
 - dados persistentes
 - mensagens

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
7

Laboratório de Engenharia de Software

O que são assertivas?




- Assertivas são uma forma de formalizar especificações
 - são uma das formas de **técnica formal leve** (*lightweight formal method*)
 - são possivelmente incompletas
 - devido à dificuldade de expressá-las
 - devido à falta de atenção do redator
 - são possivelmente incorretas
 - uso incorreto da teoria matemática
 - erros ao redigir as expressões lógicas
- Apesar das potenciais deficiências do ponto de vista formal, assertivas têm-se mostrado eficazes e eficientes como técnicas de redução e observação de defeitos em tempo de desenvolvimento

Agerholm, S.; Larsen, P.G.; "A Lightweight Approach to Formal Methods"; *Proceedings FM-Trends 98 - International Workshop on Current Trends in Applied Formal Methods*; Berlin: Springer; 1999; pags 168-183

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
8

Laboratório de Engenharia de Software

Assertivas como prevenção de defeitos




- O uso de assertivas torna a especificação "suficientemente formal"
 - a forma de raciocinar ao redigir código é diferente da forma de raciocinar ao redigir assertivas, induz assim uma redundância de raciocínio ao desenvolver
 - essa redundância pode atenuar os problemas relacionados com a revisão pelo próprio autor
 - a redundância de raciocínio é uma forma de verificação simultânea com o desenvolvimento
 - contribui para uma significativa redução da densidade de defeitos inicial
- Logo: assertivas aumentam a eficácia de revisões e inspeções

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
9

Laboratório de Engenharia de Software

Assertivas como prevenção de defeitos




- O contínuo ajuste, enquanto se redige o código, das assertivas e do correspondente código de modo que formem um todo coerente, induz uma argumentação da correteza do código
 - note que argumentação não é uma prova formal
- O esforço adicional requerido para redigir e coevoluir as assertivas é em torno de 10%
 - modo de medir: percentual de linhas de código contendo assertivas
- Logo: temos uma redução significativa do custo total:
 - menos retrabalho inútil
 - menos defeitos remanescentes → menores riscos
 - melhor documentação facilita a manutenção

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
10

Laboratório de Engenharia de Software

Exemplo real – técnicas formais leves




Número de falhas observadas por assertivas durante testes	22
Número de falhas observadas por outros meios durante testes	5
Tempo médio para remoção de falhas, assertivas	30'
Tempo médio para remoção de falhas, outros meios	6h
Número de falhas na aceitação, assertivas	2
Número de falhas na aceitação, outros meios	0
Número de falhas nos 2 meses iniciais, leves	2
Número de falhas nos 2 meses iniciais, graves	0
Número de falhas após 2 meses de uso	0

Magalhães, J.; Recovery Oriented Software; Tese de Doutorado; PUC-Rio; 2009

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
11

Laboratório de Engenharia de Software

Assertivas **executáveis**



- São instrumentos de **detecção de erros**, i.e. **falhas**
 - permitem a criação de programas auto-verificantes ("self-checking")
- Reduzem o esforço para **diagnosticar** falhas
 - a instrumentação pode ser inserida de modo que a **latência de observação** dos erros seja pequena
 - o relato de falha (log) pode conter os valores dos dados críticos no momento da detecção
- Viabilizam a geração automática de casos de teste
 - podem servir de **oráculos dinâmicos**
 - viabilizam o teste baseado na geração de dados aleatórios
- São necessárias para desenvolver artefatos **robustos**
 - observam erros, evitando danos substanciais
 - as instrumentações mais eficazes para isso são assertivas

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
12

Exemplos de assertivas executáveis



- Mover o índice de elemento corrente **numMove** elementos em direção à origem (**numMove < 0**), ao final de uma lista (**numMove > 0**), ou nada (**numMove == 0**)

```
int LST_List :: MoveCurrElement( int numMove )
{
    // Assertiva de entrada
    #ifdef _DEBUG
        EXC_ASSERT( inxCurrElem >= -1 ) ;
        EXC_ASSERT( inxCurrElem == -1 ? pCorr == NULL ; true ) ;
        EXC_ASSERT( inxCurrElem == 0 ? pPrev == NULL ; true ) ;
        EXC_ASSERT( inxCurrElem < numElem ) ;
        EXC_ASSERT( inxCurrElem == numElem - 1 ? pProx == NULL :
            true ) ;

        int inxCurrElemEntrada = inxCurrElem ;
    #endif
}
```

índice corrente pode ser < 0
sse for anterior à origem da lista

obviamente deveria existir uma relação
envolvendo **pCorr** e **inxCurrElem**

Necessário para mais tarde
saber o valor ao entrar

Outro problema: pCorr e pPrev pProx possuem
qualificadores diferentes

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

13

Exemplos de assertivas executáveis



```
// Assertiva de saída
#ifdef _DEBUG
    EXC_ASSERT( inxCurrElem >= -1 ) ;
    EXC_ASSERT( inxCurrElem == -1 ? pCorr == NULL : true ) ;
    EXC_ASSERT( inxCurrElem == 0 ? pPrev == NULL : true ) ;
    EXC_ASSERT( inxCurrElem < numElem ) ;
    EXC_ASSERT( inxCurrElem == numElem - 1 ? pProx == NULL :
        true ) ;

    EXC_ASSERT( inxCurrElem >= max( -1 ,
        inxCurrElemEntrada + numMove ) ;
    EXC_ASSERT( inxCurrElem <= min( numElem - 1 ,
        inxCurrElemEntrada + numMove ) ;
#endif
```

Efeito da operação

Observação: Uma parte da assertiva de entrada é igual a uma parte da de saída, logo essa parte da assertiva é uma **invariante estrutural**

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

14

Condição



- Condição é uma expressão cujo resultado é um **booleano**
 - pode somente assumir os valores **true** ou **false**
 - qualquer expressão válida na linguagem de programação utilizada e que avalia para um booleano pode ser uma condição
- Exemplos
 - `i != 0 j < 100 3 < pi`
 - `i`, `i + 1` e `i--` não valem, pois não são booleanos, mesmo que em algumas linguagens o inteiro 0 seja **false** e os demais inteiros **true**
 - `pArvore->pRaiz != NULL`
 - `pArvore->pRaiz` não vale, mesmo que algumas linguagens permitam usar `if(pArvore->pRaiz)` para verificar se o ponteiro é nulo ou não
 - `alpha1 = arcsin(sin (alpha)) / alpha`
 - `(0.999999 < alpha1) && (alpha1 < 1.000001)` } controle usando tolerância

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

15

Conjunção, expressão “e”



- 1) $condição_1 \ \&\& \ condição_2 \ \&\& \ ... \ \&\& \ condição_n$
 - 2) $condição_1 \wedge condição_2 \wedge ... \wedge condição_n$
 - 3) $condição_1, condição_2, ..., condição_n$
 - 4) $condição_1$
 $condição_2$
 $...$
 $condição_n$
- as quatro expressões são equivalentes
 - para que a expressão seja **verdadeira**, **todas** as condições 1, 2, ... n devem ser **verdadeiras**
 - evite a forma 2, pois confunde com o operador “^” (ou exclusivo bit a bit) de C / C++

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

16

Disjunção, expressão "ou"



1) $condição_1 || condição_2 || \dots || condição_n$

2) $condição_1 \vee condição_2 \vee \dots \vee condição_n$

- as duas formas são equivalentes
- para que a expressão seja verdadeira **uma ou mais** das condições 1, 2, ... n devem ser **verdadeiras**
- evite a forma 2, por analogia à restrição do uso de " \wedge " nas expressões conjuntivas
 - além do mais \vee confunde com a letra 'v' que, segundo analisadores léxicos, é um nome

Disjunção exclusiva, expressão "xor"



1) $condição_1 \text{ xor } condição_2 \text{ xor } \dots \text{ xor } condição_n$

- para que a expressão seja verdadeira **exatamente uma** das condições 1, 2, ... n deve ser verdadeira

- **exatamente:** uma e somente uma
- **xor:** *exclusive or*

Negação



- ! *Condição*
 - se *Condição* for **verdadeira**, a expressão será **falsa**
 - se *Condição* for **falsa**, a expressão será **verdadeira**
- Encontram-se por vezes os símbolos: ' \sim ' ou ' \neg ' para designar negação

Implicação, se



- 1) **se** *premissa* **então** *consequente*
- 2) ? *premissa* \Rightarrow *consequente*
 - a expressão ? $\omega \Rightarrow \omega'$ simplifica a análise sintática simples,
 - a expressão $\omega \Rightarrow \omega'$ somente é reconhecida depois de encontrar o " \Rightarrow "
 - se *premissa* for **verdadeira** e a *consequente* também for **verdadeira**, a expressão será **verdadeira**
 - se *premissa* for **verdadeira** e a *consequente* for **falsa**, a expressão será **falsa**
 - se *premissa* for **falsa**, a expressão será **verdadeira**
 - na realidade se *premissa* for **falsa** a expressão passa a ser irrelevante – a partir de uma premissa falsa qualquer conclusão vale
- evite **else**, redija a implicação com a premissa negada, exemplo
!(premissa) \Rightarrow consequente
 - similar a um "guarded command" (comando com guarda) em programação
- evite aninhamentos de **ifs**, redija a expressão condicional completa
- evite o uso da precedência usual das linguagens de programação, redija a expressão condicional parentetizada

Laboratório de Engenharia de Software

Se e somente se

1. *condição_a* sse *condição_b*
2. ? *condição_a* \Leftrightarrow *condição_b*

- Para a expressão ser verdadeira precisa-se mostrar que:
 - se *condição_a* == V então *condição_b* == V
 - se *condição_a* == F então *condição_b* == F
 - se *condição_b* == V então *condição_a* == V (recíproca)
 - se *condição_b* == F então *condição_a* == F

LES

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
21

Laboratório de Engenharia de Software

Conjuntos

- { A1 , A2 , ... , An }
 - em que A1, A2, ... , An são elementos, objetos, ou referências a objetos

LES

Mar 2015
Arndt von Staa © LES/DI/PUC-Rio
22

Conjuntos



- **Nome** OU **Nome{ refEstrutura }**
 - é o conjunto de todos os elementos da estrutura **Nome**,
 - caso exista a chave **refEstrutura**
 - identificada, ou referenciada, por **refEstrutura**
 - sugestão:
 - no caso C referencie sempre a cabeça da estrutura
 - no caso OO procure ancorar toda a estrutura em um objeto cabeça
 - o tipo da estrutura deve estar implícito no nome.
 - se não estiver pode-se usar **Tipo :: Nome{ refEstrutura }**
- Alternativas dependentes da linguagem usada
 - **Estrutura{ refEstrutura }** /* notação similar a C */
 - **objetoEstrutura** /* notação similar a OO */
 - referenciam elementos (objetos) da classe cabeça da estrutura

Conjuntos



- *{ regra de formação do conjunto }*
 - a regra é uma expressão,
 - ex. uma expressão envolvendo um quantificador
 - ex. uma gramática, que indica a lei de formação dos elementos do conjunto
 - exemplos
 - `{ \forall pElem \in Lista{ pLista } | pElem->pEsq == NULL }`
 - definido por*

`< Inteiros > ::= $digito 0 - 9 [$digito] ;` *produção de uma gramática*
`{ $\forall i | i \in$ Inteiros }` *todos os elementos pertencentes à gramática*

< x > : x é um não terminal, ou uma gramática (i.e. o não terminal origem das produções)

Quantificadores



- ParaTodos , ou \forall
- Pertence , ou \in
- Vale , ou :
- TalQue , ou |
- Exemplo:
 - $\forall pElem \in Lista\{ pLista \} : \text{condição1} , \text{condição2} , \dots$
 - para todos os elementos *pElem* pertencentes à lista *pLista* valem as condições: *condição1* , *condição2* , ...
 - use *pElem* caso se trate de um ponteiro para o elemento ou *refElem* caso se trate de uma referência explícita
 - obs.: em Java todos os objetos são **implicitamente** referenciados, neste caso use *elem*
 - $\forall i \mid (0 \leq i < n) : \text{condição1} , \text{condição2} , \dots$
 - para todos os *i* tal que *i* esteja no intervalo [0 .. *n*) valem as condições: *condição1* , *condição2* , ...

Quantificadores



- Exemplo:
 - $\forall pElem \in Lista\{ pLista \} \mid \text{condiçãoA} , \text{condiçãoB} : \text{condição1} , \text{condição2} , \dots$
 - $\forall pElem \in Lista\{ pLista \} \mid pElem \rightarrow pEsq \neq NULL : pElem \rightarrow pEsq \rightarrow pDir == pElem$
 - $\forall pElem \in Lista\{ pLista \} \mid pElem \rightarrow pDir \neq NULL : pElem \rightarrow pDir \rightarrow pEsq == pElem$

Quantificadores



- Existe ou \exists
- TalQue ou $|$
- Para estes valem ou $:$
- Exemplo
 - $\exists pElem \in Lista\{ pLista \} \mid condi\c{c}{a}o1 \ \&\& \ condi\c{c}{a}o2$
 - existe **pelo menos um** elemento $pElem$, pertencente à lista $pLista$, tal que as condições $condi\c{c}{a}o1$ e $condi\c{c}{a}o2$ sejam verdadeiras
- Problema com **Existe**: pode existir nenhum, pode existir exatamente um, podem existir vários
 - precisa-se considerar cada um desses três casos

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

27

Cardinalidade de conjunto



- $|| \textit{conjunto} ||$ cardinalidade – número de elementos do conjunto
- $\textit{card}(\textit{conjunto})$ cardinalidade em notação que não confunde com o operador *ou lógico* “ $||$ ”
- **vazio** ou ϕ conjunto vazio
- $? \textit{card}(\{ \forall pElem \in Lista\{ pLista \} \mid pElem \rightarrow pEsq == NULL \}) \neq 1 \Rightarrow \textit{erro}$
- $? \textit{card}(\{ \forall pNo \in Arvore\{ pArvore \} \mid pNo == pArvore \rightarrow pNoCorrente \}) \neq 1 \Rightarrow \textit{erro}$

o que querem dizer as expressões acima?

Mar 2015

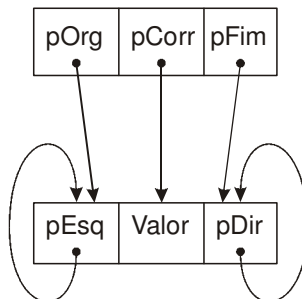
Arndt von Staa © LES/DI/PUC-Rio

28

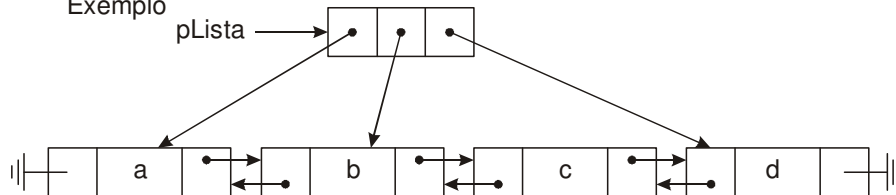
Exemplo: lista com cabeça



Modelo



Exemplo



Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

29

Exemplo assertiva estrutural: lista com cabeça



Controle da cabeça da lista quando a lista está vazia:

? pLista != NULL

fim

? ((pLista->pOrg == NULL) || (pLista->pFim == NULL) ||
(pLista->pCorr == NULL)) => pLista->numElem == 0

? pLista->numElem == 0 => (pLista->pOrg == NULL) &&
(pLista->pFim == NULL) && (pLista->pCorr == NULL)

fim

numElem é um dado redundante e que serve somente para a verificação da corretude da lista

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

30

Exemplo assertiva estrutural: lista com cabeça



Controle da cabeça de lista quando a lista não está vazia

```
? pLista->numElem > 0 => ( pLista->pOrg != NULL ) &&  
    ( pLista->pFim != NULL ) && ( pLista->pCorr != NULL )  
? pLista->pCorr != NULL =>  $\exists$  pElem  $\in$  Lista( pLista ) |  
    pLista->pCorr == pElem poderia existir mais do que um?  
? pLista->pOrg != NULL => pLista->pOrg->pEsq == NULL  
? pLista->pFim != NULL => pLista->pFim->pDir == NULL
```

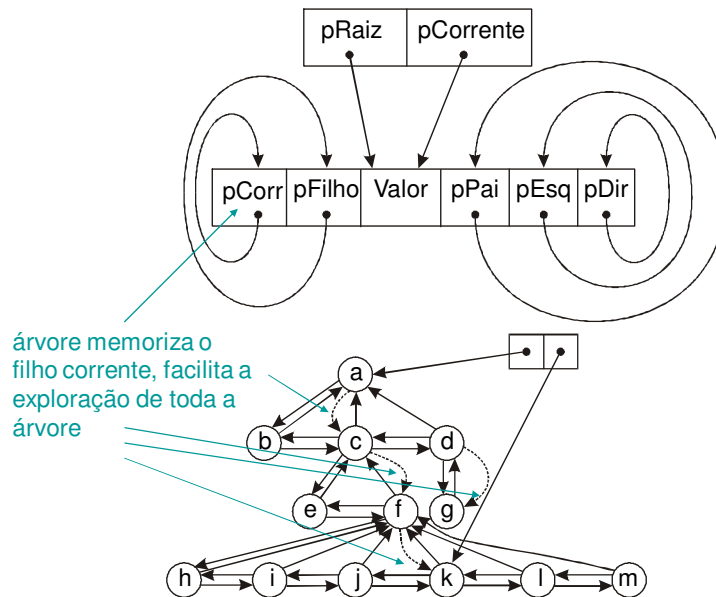
Exemplo assertiva estrutural: lista com cabeça



Verificação do corpo da lista

```
?  $\forall$  pElem  $\in$  Lista{ pLista } : /* calculado percorrendo a lista */  
    ? pElem->pEsq == NULL => pLista->pOrg == pElem  
    ? pElem->pEsq != NULL => pElem->pEsq->pDir == pElem  
    ? pElem->pDir == NULL => pLista->pFim == pElem  
    ? pElem->pDir != NULL => pElem->pDir->pEsq == pElem
```


Exemplo assertiva estrutural: árvore n-ária 1 / 4



Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

33

Exemplo assertiva estrutural: árvore 2 / 4



Cabeça da árvore é $pArv \neq \text{NULL}$

- Se a árvore estiver vazia: $pArv \rightarrow pRaiz == \text{NULL}$ e $pArv \rightarrow pCorrente == \text{NULL}$
- Se a árvore não estiver vazia:
 - $pArv \rightarrow pRaiz$ aponta para o nó raiz da árvore
 - $pArv \rightarrow pCorrente$ aponta para exatamente um dos nós da árvore

Outra redação:

$pArv \rightarrow \text{numNos} == 0 \Rightarrow pArv \rightarrow pRaiz == \text{NULL} \ \&\& \ pArv \rightarrow pCorrente == \text{NULL}$

$pArv \rightarrow \text{numNos} \neq 0 \Rightarrow pArv \rightarrow pRaiz \neq \text{NULL} \ \&\& \ pArv \rightarrow pRaiz \rightarrow pPai == \text{NULL} \ \&\&$

$\exists pNo \in \text{Arvore}\{ pArv \} \mid pNo == pArv \rightarrow pCorrente$

$pArv \rightarrow \text{numNos}$ é um atributo necessário para as assertivas, mas não necessariamente em produção

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

34

Exemplo assertiva estrutural: árvore 3 / 4



Corpo da árvore:

```
∀ pNo ∈ Arvore{ pArv } :  
  ? pNo->pPai == NULL => pArv->pRaiz == pNo &&  
    pNo->pEsq == NULL &&  
    pNo->pDir == NULL  
  ? pNo->pPai != NULL => ∃ pElem ∈ Lista{ pNo->pPai->pFilho } |  
    pElem == pNo  
  ? pNo->pFilho == NULL => pNo->pCorr == NULL  
  ? pNo->pFilho != NULL =>  
    ( pNo->pFilho->pEsq == NULL ) &&  
    ( ∀ pFil ∈ Lista{ pNo->pFilho } : pFil->pPai == pNo ) &&  
    ( ∃ pFil ∈ Lista{ pNo->pFilho } | pNo->pCorr == pFil )
```

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

35

Exemplo assertiva estrutural: árvore 4 / 4



```
∀ pNo ∈ Arvore{ pArv } :  
  /* continuação */  
  ? pNo->pEsq == NULL && pNo->pPai == NULL =>  
    pArv->pRaiz == pNo  
  ? pNo->pEsq == NULL && pNo->pPai != NULL =>  
    pNo->pPai->pFilho == pNo  
  ? pNo->pEsq != NULL => pNo->pEsq->pDir == pNo  
  ? pNo->pDir == NULL => true  
  ? pNo->pDir != NULL => pNo->pDir->pEsq == pNo  
  ? pNo->pDir != NULL => pNo->pPai == pNo->pDir->pPai
```

- sempre assegure que sejam consideradas todas as partições da expressão usada como premissa
- no caso de dúvida crie uma tabela de decisão para fins de verificação

Mar 2015

Arndt von Staa © LES/DI/PUC-Rio

36

Variáveis e atribuição lógicas



- **Seja** Variável_a = Variável_b
ou
- **Seja** Variável_a = Expressão
 - variáveis lógicas que existem somente para fins de processamento de condições, exemplo:
 - *pArv->numNos* – esta variável não faz parte do modelo da árvore, entretanto poderia fazer e é facilmente calculável ao efetuar operações sobre árvores
 - o tipo da variável pode ser qualquer,
 - se não for booleano, a variável precisará ser utilizada em uma condição
 - se for booleano, a variável pode ser a própria condição
 - variáveis e atribuições lógicas não devem ser necessárias para o processamento produtivo, dessa forma podem corresponder a código compilado condicionalmente

Funções lógicas



- nome_da_função (lista_de_parâmetros) ::= expressão
 - ::= leia-se “definida como”
 - a expressão deve envolver todos os parâmetros
 - a função não deve gerar efeitos colaterais
 - alterações de variáveis lógicas somente podem ser realizadas por atribuições lógicas específicas
 - funções lógicas não devem ser necessárias para o processamento produtivo, dessa forma podem corresponder a código compilado condicionalmente
 - Exemplo

bissexto(a) ::= a div 400 ||

(a div 4 && !(a div 100))

a div b: retorna true se *a* for divisível por *b*, ex. (*a*>0 && *b*>0 ? *a*%*b*==0 : false)

No **calendário Gregoriano** (1582): são bissextos os anos divisíveis por 400, dos restantes não são bissextos os divisíveis por 100, dos restantes agora são bissextos os divisíveis por 4. Segundo a conta o ano “médio” terá 365 + 97 / 400 dias = 365,2425 dias. O ano Tropical medido é aproximadamente de 365,24219 dias, assim o calendário Gregoriano produz um excesso de um dia a cada 3300 anos.

Controle de tempo, proposta



- Em tempo de execução o controle de tempo requer uma função que retorne o relógio corrente
 - C/C++ a expressão `clock() / CLOCKS_PER_SEC` retorna tempo decorrido desde o início da execução em segundos, com uma fração de possivelmente centésimos de segundo dependendo da máquina, do compilador e do modo de otimização. Problema: falta de precisão e portabilidade
- Com essa expressão pode-se marcar o início e o final da execução e verificar se o tempo decorrido está dentro das restrições esperadas. Problemas:
 - latência / precisão-do-relógio precisa ser grande (>100?)
 - multiprogramação Heisenbug
- Assertiva temporal que mede o tempo consumido por P
`latency(P(argumentos)) < tempo limite`



FIM