Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

A Linguagem de Programação Ita

por

Marco Túlio de Oliveira Valente
Roberto da Silva Bigonha

RT 005/96

Caixa Postal, 702

30.161 - Belo Horizonte - MG

Fevereiro de 1996

Sumário

A Linguagem Ita

Make it as simple as possible,
but not simpler.
Albert Einstein

1 Introdução

Ita[1] é uma linguagem de programação inspirada em C[1], tendo como objetivo de lhe introduzir conceitos de programação orientada por objetos, como herança, encapsulamento de dados e polimorfismo, adotando-se a filosofia de programação de Eiffel[2, 3], porém no estilo de C++[4]. Esses conceitos incentivam a produção de *software* com alto grau de reusabilidade. Além de reusabilidade, incentiva-se também a produção de *software* correto, isto é, de acordo com sua especificação, através de um estilo de programação por contrato. Dentre os recursos de Ita que procuram suportar programação por contrato estão pré e pós-condições, invariantes e tratamento de exceções. Objetos em Ita são tratados uniformemente através de uma semântica de referência.

Nesse trabalho, descrevem-se essencialmente as extensões incluídas em Ita com relação a C, não sendo abordados tópicos que já estão descritos no manual de referência dessa última [1]. Uma descrição completa da gramática da linguagem é mostrada no apêndice A

2 Estrutura Léxica

2.1 Comentários

Além dos comentários normais `/* ..... */`, dispõe-se de comentários iniciados por um `//` e que prosseguem até o final da linha.

2.2 Palavras Chave

Foram acrescentadas 25 novas palavras chave:

[1]Uma referência ao Pico do Itacolomi (em tupi-guarani, pedra com filhote) de 1797 metros, localizado na Serra do Espinhaço, em Ouro Preto, MG.

```
abstract   friend     persistent  retry
check      in         pre         self
class      init       pos         set
deferred   initial    private
delete     invariant  public
exception  is         rescue
finish     new        result
```

2.3 Constantes

Além de constantes inteiras, reais, caracteres e strings, lta possui constantes conjunto, usadas para se referir a conjuntos de inteiros não negativos.

Exemplo:

```
{0,1,2,3}, {2,5..10,13}, {}
```

3 Tipo Conjunto

Além dos tipos derivados vetor, ponteiro, estrutura e união, lta possui o tipo derivado `set`, usado para declarar conjuntos de inteiros não negativos. Sobre operandos desse tipo, aplicam-se os seguintes operadores: `in` (pertinência), `+` (união), `-` (diferença), `*` (interseção) e `/` (diferença simétrica)[2].

Exemplo:

```
set x= {0,2,4,6,8,10}, y;
y= x + {1,3};
```

4 Classes

Classes são estruturas usadas para implementar Tipos Abstratos de Dados (TAD). Uma classe define um tipo, usado para declarar instâncias da classe (objetos).

Uma classe possui um cabeçalho e um corpo. O cabeçalho de uma classe contém o seu nome e, opcionalmente, uma indicação de que ela é uma classe abstrata (seção 11), uma lista de parâmetros genéricos (seção 4.3), o nome de sua superclasse (seção 9) e de

[2] $x/y = (x - y) + (y - x)$

suas classes *friends* (seção 4.2). No corpo de uma classe declaram-se os seus membros, que podem ser *membros-de-dados (atributos)* ou *membros-funções (métodos)* e, opcionalmente, o invariante da classe e/ou um bloco `retry` para a mesma (seção 8).

A implementação dos métodos de uma classe pode ocorrer na própria definição da classe (implementação interna) ou fora de sua definição (implementação externa). Nesse último caso, a definição da classe contém apenas um protótipo para o método. Implementação externa é necessária no caso de classes mutuamente recursivas. Quando a implementação é externa, o operador de qualificação `::` indica entrada no escopo da classe à qual o método pertence.

Exemplo:

```
class A {                    // Membros privados:
    int a1;                      // atributo (membro-de-dado)
    int f1 (...) { ..... }       // definicao de metodo (membro-funcao)
    .....
  public:  .....             // Membros publicos:
    int a2;                      // atributo (membro-de-dado)
    int f2 (...);                // prototipo de metodo (membro-funcao)
}

int A::f2 (...) { ..... }  // implementacao externa de metodo
```

O escopo de um membro, isto é, a região do programa em que ele tem efeito, é determinado pelo especificador de acesso em vigor durante a sua definição. Existem dois especificadores de acesso: `public` e `private`. Um especificador vigora até a definição do próximo. O especificador de acesso inicial em uma classe é `private`.

Membros privados, isto é, declarados sob especificador `private`, são visíveis apenas no corpo da classe; membros públicos (`public`) possuem o mesmo escopo do nome da classe. No caso de atributos públicos, no entanto, o acesso é apenas para leitura.

Uma classe somente tem acesso a identificadores globais que sejam tipos, isto é, identificadores de outras classes ou identificadores introduzidos em *typedefs*.

4.1 Instanciação de Classes

Objetos são instâncias de classes, consistindo em uma área de memória própria, que armazena o valor corrente de seus atributos. São acessados através de referências. Por exemplo,

uma declaração como a seguinte

```
A a;
```

introduz `a` como uma referência para objetos da classe `A`, sem, no entanto, criar nenhum objeto correspondente. Toda referência possui um *tipo estático*, com o qual foi declarada, e um *tipo dinâmico*, que é o tipo do objeto correntemente referenciado por ela (seção 10).

Durante a execução de um método, o objeto corrente é acessado através da pseudo-referência `self`.

A efetiva criação e destruição de objetos é feita através dos operadores `new` e `delete`. O operador `new` aloca memória para um objeto de determinada classe e retorna uma referência para o mesmo. A área alocada *não* é inicializada com valores *defaults*. O operador `delete` libera a memória associada a uma referência.

Uma classe pode possuir dois tipos de métodos especiais: *inicializadores* (`init`) e *finalizadores* (`finish`). Métodos inicializadores são chamados implicitamente pelo operador `new`, após esse ter criado o objeto. Opcionalmente, uma lista de parâmetros de chamada para o método inicializador pode fazer parte da expressão `new`. Métodos inicializadores são usados para inicializar a área de memória de um objeto e garantir a validade de seu invariante (seção 8). Como não existe inicialização com valores *defaults*, toda classe com invariante definido deve possuir pelo menos um inicializador.

Os métodos finalizadores são chamados pelo operador `delete` antes que ele libere a área de memória de um objeto. Métodos finalizadores não possuem parâmetros formais. Por esse motivo, não existe lista de parâmetros de chamada no operador `delete`.

Exemplo:

```
class A {
    .....
  public:
    int x;
    init (int, float)  { ..... }    // inicializadora
    finish ()  { ..... }            // finalizadora
    void f (...)  { ..... }
}
.....
A a1= new A (10, 3.14);
a1.f (...);
delete a1; .....
```

Métodos inicializadores e finalizadores estão sujeitos às mesmas restrições de escopo de quaisquer outros membros de uma classe.

4.2 Classes *Friends*

Se uma classe `A` diz, em seu cabeçalho, que outra classe `B` é uma *friend* sua, então membros privados de `A` são acessíveis em `B`. No caso de atributos, o acesso é apenas para leitura. A exceção são os atributos herdados por subclasses *friends*. Somente nesse caso permite-se acesso também para escrita.

Exemplo:

```
class A friend B, C {
  int x;  .....
}

class B { .....
  int g (void) {
    A a= new A;
    int y= a.x;    // Acesso a atributo privado de A
    .....
  }  .....
}

class C: A {       // C subclasse friend de A
  int g (void) {
    x= 0;          // Acesso para escrita
  }
}
```

Subclasses *friends* podem redefinir não apenas membros públicos da superclasse, mas também membros privados (seção 9.1).

A relação entre classes *friends* é reflexiva, mas não é simétrica nem transitiva.

4.3 Classes Genéricas

Classes genéricas são classes que possuem tipos como parâmetros. Tais parâmetros (tipos) são chamados de *parâmetros genéricos (tipos genéricos)*. Tipos genéricos são usados, no escopo de classes genéricas, para declarar atributos, valores de retorno, parâmetros formais ou variáveis automáticas de funções. Em todos esses casos, o que se define efetivamente é uma referência para o parâmetro (tipo) genérico.

Não é possível criar objetos de tipo genérico, isto é, o operador `new` não se aplica a referências de tipo genérico.

Parâmetros genéricos podem ser de duas formas:

Irrestrito: quando aceita-se qualquer tipo como parâmetro genérico. Nesse caso, apenas operações aplicáveis a qualquer referência podem ser aplicadas sobre referências desse tipo, isto é, operações de atribuição, duplicação e comparação (seção 5.1).

Restrito: quando o parâmetro genérico é restringido por um tipo, chamado *tipo restringente*; possíveis parâmetros de chamada deverão ser subtipos do tipo restringente. Com isso, todas as operações do tipo restringente poderão ser aplicadas a referências genéricas.

Exemplo:

```
class A <T> {
  .....
  T a1;
  T f1 (T x) { ..... }
  .....
}
.....
A <X> a1;
A <A<Y>> a2= new A<A<Y>> (.....);
B <C1> b1= new B <C1> (.....);
.....
```

```
class B <T: C> {
  .....
  T b1;
  T f1 (T x) { ..... }
  .....
}
```

No escopo de uma classe genérica, referências recursivas a ela somente são válidas se os seus argumentos correspondem exatamente aos parâmetros genéricos da própria classe.

Não é possível fazer uso de uma classe genérica antes de sua definição, isto é, não existe anúncio (*forward*) de classes genéricas.

5 Semântica de Referência

5.1 Operações Pré-definidas

Sobre qualquer referência podem ser realizadas as operações descritas nas tabelas abaixo (supondo `p1` e `p2` dois tipos (classes) `P1` e `P2`, respectivamente)[3].

ATRIBUIÇÃO		
	Semântica de Referência	Semântica de Valor
Sintaxe	`p1= p2`	`void copy (P1 p1, P2 p2)`
Restrição	`P2` subtipo de `P1`	`p1 != NULL`; `P2` subtipo de `P1`
Semântica	atribuição de referências. `p1` passa a ter o mesmo tipo dinâmico de `p2`.	copia-se a parte `P1` do objeto referenciado por `p2` para o objeto referenciado por `p1`. O tipo dinâmico de `p1` fica inalterado.

DUPLICAÇÃO	
Sintaxe	`void clone (P1 p1, P2 p2)`
Restrição	`P2` subtipo de `P1`
Semântica	cria-se um novo objeto conforme o tipo dinâmico de `p2`; faz-se `p1` referenciá-lo; copia-se os atributos de `p2` para o objeto criado.

COMPARAÇÃO		
	Semântica de Referência	Semântica de Valor
Sintaxe	`p1 == p2` ou `p1 != p2`	`char equal (P1 p1, P2 p2)`
Restrição	não há	`P2` subtipo de `P1`
Semântica	comparação de referências	comparação dos objetos referenciados por `p1` e `p2`

5.2 Passagem de Parâmetros

Assim como em C, toda passagem de parâmetros é *por valor*. No caso de parâmetros do tipo classe, passa-se, por valor, uma referência para um objeto.

[3]Na descrição dessas operações, usa-se o conceito de subtipo, descrito na seção 9. Caso não deseje consultar essa seção de imediato, considere apenas que um tipo é sempre subtipo dele mesmo.

Exemplo:

```
class A {
  public:
    int x;
    init (int x2) { x= x2; }          // inicializadora
    void set_x (int x2) { x= x2; }    // atualiza "x"
}

void f (A a) { a.set_x (1); }         // parametro formal tem
.....                                 // tipo que e uma classe
A a = new A (0);                      // a.x = 0
f (a);
printf ("%d\n", a.x);                 // a.x = 1
```

6 Funções Polivalentes

Funções em Ita com parâmetros formais de tipo classe ou de tipo genérico são chamadas de *funções polivalentes*[4]. Um parâmetro formal de tipo classe pode receber parâmetros de chamada dessa classe e de suas subclasses. O mesmo ocorre com parâmetros genéricos de funções membro de classes genéricas, que também podem receber parâmetros de chamada de qualquer tipo ou de uma hierarquia de tipos, dependendo do tipo do parâmetro (irrestrito ou restrito) e da instanciação da classe.

Logo, tais funções, apesar de possuírem uma única implementação, podem ser empregadas com objetos de diferentes tipos.

Exemplo:

```
void f (A a, B b, ...) { ..... }
```

7 Funções Polissêmicas

Polissemia[5] em Ita designa a capacidade de uma função possuir várias implementações, sendo todas elas denotadas pelo mesmo nome. A determinação da implementação a ser

[4]Polivalente, pelo Aurélio, é um adjetivo que refere-se ao que oferece diversas possibilidades de emprego ou aplicação; que é eficaz em vários casos diferentes.

[5]Polissemia, segundo o Dicionário Aurélio, refere-se a uma palavra com muitas significações.

executada como resultado de uma chamada é feita avaliando-se, nessa ordem:

- O número e tipo dos parâmetros das diversas funções;
- A forma do objeto denotado pela referência sobre a qual aplica-se a função;
- O estado do objeto denotado pela referência sobre a qual aplica-se a função.

Descreve-se abaixo o primeiro e terceiro casos de polissemia. O segundo caso é tratado na seção 10.1.

7.1 Polissemia por Número e Tipo dos Parâmetros

Diz-se que uma função, seja ela um método ou não, é polissêmica pelo número e tipo dos seus parâmetros, quando existe outra de mesmo nome em seu escopo e da qual ela se distingue pela sua *identificação* [6].

A identificação de uma função (ou chamada de função) é o par

$$(\text{nome-função}, \text{assinatura})$$

onde *assinatura* é a tupla formada pelos tipos de seus parâmetros formais (ou parâmetros reais). Por exemplo, a identificação do primeiro método `f` mostrado acima é o par $(f, (int))$.

Uma identificação

$$(f, (T_1, T_2, \ldots, T_m))$$

conforma-se com outra identificação

$$(g, (U_1, U_2, \ldots, U_n))$$

se $f = g$ e $(T_1, T_2, \ldots, T_m)$ conforma-se com $(U_1, U_2, \ldots, U_n)$.

Diz-se que $(T_1, T_2, \ldots, T_m)$ conforma-se com $(U_1, U_2, \ldots, U_n)$ se $m = n$ e T_i é igual a U_i ou T_i é compatível para atribuição com U_i ou T_i é um subtipo de U_i, para $i \leq m$

A identificação de uma função deve ser única em seu escopo, isto é, não deve existir outra função cuja identificação se conforme com a sua (unicidade da definição).

Exemplo:

[6] Essa forma de polissemia em outras linguagens é chamada de *sobrecarga*.

```
class A {
  int f (int) { ..... }          // funcoes polissemicas
  int f (int, float) { ..... }
}
```

A identificação de uma chamada de função deve se conformar com a identificação da função a ser executada. Por exemplo, uma chamada da forma `f (10, 2)` executará a segunda função `f` acima, pois $(f, (int, int))$ (identificação da chamada) conforma-se com $(f, (int, float)$ (identificação da função).

7.2 Polissemia por Estado

Funções polissêmicas por estado são métodos de mesma assinatura, mas com implementações diferentes em função do estado da classe.

O especificador de tipo `state` é usado para definir estados de objetos de uma classe, isto é, um determinado conjunto de valores de seus atributos. Um estado deve ser sempre inicializado com uma expressão booleana que, quando avaliada, informará se um objeto da classe encontra-se ou não nesse estado.

A cláusula `at` introduz o estado de execução de uma função polissêmica por estado.

Exemplo:

```
class A {
   int temp, umidade;
   state frio= temp < 15;        // estados
   state umido= umidade > 70;
   int f (int, float) at frio { ..... }   // funcoes polissemicas
   int f (int, float) at umido { ..... }
 }   .....
```

A escolha da implementação a ser executada em uma chamada de função polissêmica por estado é feita avaliando as cláusulas `at` na ordem em que aparecem na definição da classe. A primeira implementação cuja cláusula `at` avalie em verdade é executada. Não existindo tal cláusula, nenhuma implementação é executada e ativa-se uma exceção com motivo `STATE_FAILURE`.

8 Asserções

Asserções são expressões booleanas avaliadas em tempo de execução e que descrevem propriedades de trechos de código. Asserções podem ser de três tipos:

Invariante: asserção associada a classes e que deve ser válida para todas as instâncias das mesmas, isto é, no início e no fim da execução de cada um de seus métodos.

Pré-condição: asserção associada a métodos ou funções comuns e que deve ser válida no início da execução dos mesmos.

Pós-condição: asserção associada a métodos ou funções comuns e que deve ser válida ao término da execução dos mesmos.

Exemplo:

```
class A {
  int a;
  .....
  int f (int x)
  pre (x > 0)                                        // pre-condicao
  pos ( (result > 0) && (a == initial (a) + 1)) // pos-condicao
  { ..... }
  .....
  invariant: (a >= 0)       // invariante
}
```

Quando uma asserção é violada, isto é, sua avaliação resulta em falso, produz-se uma exceção (seção 8.1).

A expressão booleana de uma asserção pode conter funções. Considera-se, no entanto, uma má técnica o uso de funções com efeitos colaterais em asserções.

Métodos inicializadores e finalizadores não podem possuir nem pré nem pós-condições. No entanto, o invariante da classe é avaliado após a execução de um método inicializador e antes da execução de um finalizador.

Se um método não possui pré-condição, assume-se que a sua pré-condição seja `true` (a não ser que o método esteja sendo redefinido; nesse caso ele herda a pré-condição do pai (seção 9.1)). O mesmo ocorre com a pós-condição.

Em uma pós-condição pode-se usar duas expressões especiais:

- `initial (exp)`: valor da expressão `exp` na chamada da função.
- `result`: valor a ser retornado pela função.

O invariante de uma classe somente é avaliado nos "estados estáveis" da mesma, isto é, imediatamente antes e depois de uma chamada remota de um método da classe e após uma função inicializadora. Classes sem invariante possuem implicitamente um invariante `true`.

Mostra-se na tabela abaixo as diversas situações em que asserções de uma classe `C` são avaliadas:

Situação	Fórmula de Correção
Operador `new` com inicializadora c sobre C	$\{T\}\ new\ \ body_c\ \{\text{INV}_C\}$
Operador `delete` sem finalizadora sobre C	$\{\text{INV}_C\}\ delete\ \{T\}$
Operador `delete` com finalizadora d sobre C	$\{\text{INV}_C\}\ body_d\ \ delete\ \{T\}$
Chamada local (sem qualificação) de um método m de C	$\{\text{pre}_m\}\ body_m\ \{\text{pos}_m\}$
Chamada remota (com qualificação) de um método m de C	$\{\text{INV}_C \wedge \text{pre}_m\}\ body_m\ \{\text{INV}_C \wedge \text{pos}_m\}$[7]

Onde: T é o valor lógico `true`; INV_C é o invariante da classe C; $body_f$ é o resultado da execução do corpo da função f; pre_f e pos_f são, respectivamente, as pré e pós-condições da função f.

Pode-se incluir uma asserção em qualquer parte de um programa através da palavra reservada `check`.

```
check NEGATIVE (x >= 0);
```

No exemplo, `NEGATIVE` é uma constante inteira que define um código para a exceção a ser ativada no caso de violação da asserção `(x >= 0)`. Esse código é usado para identificar a exceção durante a execução de seu tratador (seção 8.1).

[7] Como operações lógicas são avaliadas com *curto-circuito*, é importante notar que essa expressão garante que o invariante é sempre avaliado antes das pré e pós-condições de uma função.

8.1 Tratamento de Exceção

Quando uma asserção é violada (sua avaliação resulta em falso), produz-se uma *exceção*. A ativação de uma exceção transfere o controle para o bloco `rescue` associado à função na qual a exceção ocorreu. O bloco `rescue` associado a uma função é o que segue o seu código ou, se esse não existir e a função for um método, o bloco `rescue` da classe do método. Classes e funções comuns sem bloco `rescue` possuem implicitamente um bloco `rescue` vazio. O bloco `rescue` de uma classe ou membro-função nunca é herdado[8].

Exemplo:

```
class A {
  .....
  public:
    void f(); .....
    pre (...) pos (...) { ..... }
    rescue {  ..... retry; ..... }    // bloco rescue de f
}
rescue { ..... retry; ..... }         // bloco rescue da classe
.....
```

Quando uma pré-condição é violada, ativa-se uma exceção na função chamada e não na função chamadora.

A execução de um bloco `rescue` pode terminar de duas formas:

- *Chegando-se ao seu fim:* nesse caso, a função falha; o controle retorna à função chamadora, com a exceção levantada. Isso implica que a execução é desviada do ponto de retorno para o bloco *rescue* da função chamadora. Se a exceção se propagar até a função `main ()`, o programa será abortado.

- *Executando-se um* `retry`*:* nesse caso, volta-se a executar o corpo da função, reavaliando-se sua pré-condição e mantendo-se os valores correntes de suas variáveis automáticas; a exceção é desativada, embora possa ser ativada mais uma vez nessa nova execução.

[8] A justificativa para essa restrição é que o bloco `rescue` pode referenciar variáveis locais da função à qual ele está associado.

Durante a execução de um bloco `rescue`, o levantamento de novas exceções fica inibido.

Em um bloco `rescue`, a expressão `exception`, do tipo inteiro, retorna um código para a exceção que está sendo tratada. Existem os seguintes códigos pré-definidos:

Código	Motivo da Exceção
`PRE_INVARIANT`	Violação de invariante na entrada de método
`POS_INVARIANT`	Violação de invariante na saída de método
`PRECONDITION`	Violação de pré-condição de método
`POSCONDITION`	Violação de pós-condição de método
`ROUTINE_FAILURE`	Falha em rotina chamada (retorno com exceção levantada)
`VOID_CALL`	Tentativa de chamada de método em objeto do tipo `NULL`
`TYPE_FAILURE`	Falha em *type guard* (seção 10.2)
`STATE_FAILURE`	Falha em execução de função polissêmica
`DEFERRED_VIOLATION`	Tentativa de execução de método não efetivado

Podem existir ainda códigos definidos pelo programador e associados a exceções levantadas através de um `check`.

9 Herança

Herança é um mecanismo para construir classes através da especialização de classes já existentes. Sendo `A` a classe já existente e `B` a classe que está sendo construída, diz-se que `B` é uma *subclasse* ou uma *classe derivada* de `A`. Ao contrário, `A` é a *superclasse* ou *classe base* de `B`. Dessa definição decorre que a herança é simples, isto é, que uma subclasse possui apenas uma superclasse.

Exemplo:

```
class A {                        class B: A {
  int a1;                          int b1;
  int f1 (int x) { ..... }         int g1 (int x ) { ..... }
public:                          public:
  int a2;                          int b2;
  init (int x) { a2= x; }          init (int x, int y): A (y) { b2= x; }
  int f2 (int) { ..... }           int g2 (int) { ..... }
} .....                          } .....
```

Assim como sua superclasse, uma subclasse define também um tipo, que, em lta, é dito ser um *subtipo* do tipo da superclasse. Um objeto de um subtipo pode ser usado onde seu supertipo for esperado. Para simplificar algumas regras, considera-se que um tipo é sempre um subtipo e um supertipo de si mesmo.

Uma subclasse herda *todos* os membros públicos ou privados de sua superclasse, com exceção dos métodos inicializadores e finalizadores. No entanto, apenas os membros públicos são acessíveis. Uma subclasse pode ainda definir seus próprios membros, assim como redefinir os que foram herdados. No exemplo acima, `B` possui os atributos `a1` (não acessível), `a2`, `b1` e `b2` e os métodos `f1` (não acessível), `f2`, `g1` e `g2`.

Opcionalmente, uma função inicializadora de uma subclasse pode especificar que a inicializadora da superclasse deve ser executada antes dela própria. Deve-se, nesse caso, informar a lista de parâmetros de chamada da função. Como exemplo, tem-se a função `init` da subclasse `B` acima.

O invariante de uma superclasse é automaticamente combinado ao invariante de suas subclasses através de um `and` lógico. Por exemplo, se uma classe `A` tem invariante INV_A e uma subclasse `B` invariante INV_B, o invariante efetivamente calculado para a classe `B` é $\text{INV}_A \wedge \text{INV}_B$.

9.1 Redefinição de Membros

Membros públicos de uma classe podem ser redefinidos em suas subclasses. Permite-se a redefinição de membros privados apenas quando a subclasse for *friend* da superclasse. Entende-se por redefinição a alteração do tipo de um atributo, da expressão booleana de um estado ou do corpo de um método, com ou sem alteração de sua assinatura.

Exemplo:

```
class A {                  class B: A {
  .....                      .....
  public:                    public:
    A x;                       B x;               // redefinicao atributo
    int y;                     state s= y < 15;   // redefinicao estado
    state s= y < 20;           B f (P p) { .... } // redefinicao metodo
    A f (Q q) { ..... }                           // P supertipo de Q
}                          }
```

Atributos podem ser redefinidos para um subtipo do tipo original. Usando-se o conceito de identificação (seção 7.1), a redefinição de um método

$$R\ f\ (Q_1\ q_1, Q_2\ q_2, \ldots, Q_m\ q_m)$$

de uma classe `A` para um método

$$S\ f\ (P_1\ p_1, P_2\ p_2, \ldots, P_m\ p_m)$$

em uma subclasse `B` é válida se:

1. `S` for um subtipo de `R` e P_i for um supertipo de Q_i, para $i \leq m$ (regra da contravariância).

2. Não existe outra identificação de `f` em `A` que conforme-se com a identificação de `f` em `B` (unicidade da redefinição).

Na redefinição de um método, a pré-condição do método original aplica-se automaticamente ao novo método, sem possibilidade de ser redefinida. Já a pós-condição pode ser redefinida, mas a pós-condição original é automaticamente combinada com a nova através de um `and` lógico. Por exemplo, se um método `f` de `A` com pré e pós-condições $(\text{pre}_{f_A}, \text{pos}_{f_A})$ for redefinido em uma subclasse `B` com pós-condição pos_{f_B}, as pré e pós-condições efetivamente calculadas para esse método serão $(\text{pre}_{f_A}, \text{pos}_{f_A} \wedge \text{pos}_{f_B})$.

Funções membro com redefinições ao longo de uma hierarquia de classes são chamadas de funções polissêmicas por forma (seção 10.1).

9.1.1 Acesso a Métodos da Superclasse

No escopo da redefinição de um método, o acesso ao método original da superclasse é feito através da pseudo-referência `super`.

Exemplo:

```
class A { .....                 class B: A { .....
  int f (void) { ..... }          int f (void) {    // redefincao
  .....                             super.f ();     // executa A::f
}                                   .....
                                  } .....
                                }
```

10 Polimorfismo

Referências em Ita são polimórficas[9], isto é, podem denotar em tempo de execução objetos de uma classe e de todas suas subclasses. Daí surgem os conceitos de *tipo estático* e *tipo dinâmico* de referências. O tipo estático é o tipo com o qual ela foi declarada. Já o tipo dinâmico é seu tipo efetivo em tempo de execução e é sempre um subtipo do estático.

Como afirmado na seção 5.1, o tipo dinâmico de uma referência é alterado por uma operação de atribuição ou pela passagem de parâmetro.

Exemplo:

```
P1 p1;              // tipo estatico de p1: P1
P2 p2= new P2;      // tipo estatico de p2: P2
p1= p2;             // tipo dinamico de p1: P2
p1.f();             // executa f() de P2
```

10.1 Polissemia por Forma

Funções polissêmicas por forma são métodos com implementações diferentes (redefinições) em uma hierarquia de classes[10]. Nesse tipo de polissemia, o tipo dinâmico da referência sobre a qual aplica-se a função, isto é, a forma dessa referência, é que determina qual método será executado. Diz-se também que funções polissêmicas por forma são chamadas usando-se associação dinâmica (*dynamic binding*).

10.2 Verificação Dinâmica de Tipos

Duas construções foram incorporadas à linguagem para manipular tipos dinâmicos:

- Teste-de-tipo (*type test*): expressão que testa se o tipo dinâmico de uma referência é igual ou derivado de um dado tipo. Um teste-de-tipo têm a forma `(a is A)`, onde `a` é uma referência e `A` um tipo classe.

- Guarda-de-tipo (*type guard*): asserção que visa garantir que uma referência possui um certo tipo dinâmico. Uma guarda-de-tipo tem a forma `((A) a)`, onde `A` é um tipo classe e `a` uma referência. Se `a` tem tipo dinâmico `A` (ou derivado de `A`), a expressão

[9]Polimorfismo, segundo o Dicionário Aurélio, é a propriedade daquilo que se apresenta sobre várias formas ou que é sujeito a variar de forma.

[10]Em C++, tais funções são chamadas de *funções virtuais*.

toda é considerada como sendo desse tipo. Caso contrário, ativa-se uma exceção com motivo `TYPE_FAILURE`.

Note que guardas-de-tipo possuem a mesma sintaxe de um *type casting* de C, isto é, a semântica de *type casting* envolvendo referências é definida por uma guarda-de-tipo.

11 Classes Abstratas

Classes abstratas não podem possuir instâncias, isto é, o operador `new` não aplica-se a essas classes. Devido a essa característica, classes abstratas podem possuir métodos cuja implementação somente estará disponível em suas subclasses. Para tais métodos, chamados de métodos postergados (*deferred*), definem-se apenas seus protótipos, incluindo pré e pós-condições.

Classes abstratas são distinguidas pela palavra reservada `abstract` e métodos postergados pela palavra `deferred`.

Exemplo:

```
abstract class A { .....        // classe abstrata
  deferred void f (int x)       // metodo postergado
  pre (x > 0) pos (...);        // inclui pre e pos-condicoes
  .....
}
```

Um programa que defina uma subclasse de uma classe abstrata e não efetive a implementação dos métodos postergados herdados deverá também declarar essa subclasse como sendo uma classe abstrata. Caso contrário, a tentativa de execução de um método não implementado ativará uma exceção com código `DEFERRED_VIOLATION`.

12 Inicialização

Assim como em C, variáveis externas e estáticas podem ser inicializadas somente com expressões constantes. No caso de variáveis automáticas, o inicializador pode ser qualquer expressão, incluindo chamadas de função.

13 Estrutura de Programa

Um programa é composto por uma série de unidades de tradução. Cada unidade de tradução (contida em um arquivo) consiste em uma seqüência de declarações externas. Essas podem ser de três tipos: definição de função, definição de classe ou outra declaração. Uma definição de função ou de classe somente pode ocorrer no primeiro nível léxico. A execução de um programa inicia-se pela ativação da função de nome `main ()`.

Cada declaração possui um escopo, isto é, uma região do programa onde ela tem efeito. O escopo de uma declaração é determinado pelo especificador de acesso em vigor durante sua definição. Os especificadores de acesso são os mesmos usados para definir o escopo de membros de uma classe: `public` (declarações com escopo de programa) e `private` (declarações com escopo de arquivo). Um especificador de acesso vigora até que seja encontrado um novo. O especificador de acesso inicial é `private`.

Exemplo:

```
// arq1.ita
  .....
public:
  .....
private:
  .....
```

A palavra `static`, quando fora de um bloco, dá escopo de arquivo à declaração que lhe segue, mesmo que ela pertença a uma região com especificador de acesso público.

A existência de um especificador de acesso para cada declaração permite que o compilador gere automaticamente um arquivo de cabeçalho (extensão `.ih`) para cada arquivo Ita (extensão `.ita`).

Referências

[1] Kernighan, K. and Ritchie, D.M. *The C Programming Language (second edition).* Prentice-Hall, 1988.

[2] Meyer, Bertrand *Object Oriented Software Construction*, Prentice-Hall, 1988.

[3] Meyer, Bertrand *Eiffel: The Language*, Prentice-Hall, 1992.

[4] Stroustrup, B. *The C++ Programming Language (second edition)*. Prentice-Hall, 1991.

[5] Wirth, N. What can we do about the unnecessary diversity of notation for syntatic definitions? *Communications of ACM* 20, (11):822-823, November 1977.

A A Gramática de Ita

Mostra-se nesse apêndice a gramática completa de Ita. Basicamente, essa gramática é uma extensão da gramática ANSI de C [1] com as construções propostas por Ita. Na descrição da gramática usa-se uma variante da BNF proposta em [5]. Nessa variação, terminais são mostrados entre aspas duplas (ex: `"if"`), repetição é expressa por chaves (ex: $\{a\}$ equivale a $\epsilon|a|aa|aaa|\cdots$) e opcionalidade por colchetes (ex: $[a]$ equivale a $\epsilon|a$).

A.1 Definições Externas

```
translation_unit= [ access_specifier ":" ] external_definition
                { [ access_specifier ":" ] external_definition }

external_definition= function_definition
          | class_definition
          | declaration

function_definition= effective_function
          | deferred_function

effective_function= function_signature function_body

deferred_function= "deferred" function_signature ";"

function_signature= [declaration_specifier] function_declarator
            [function_header]

function_declarator= ["::"] declarator

function_header= [init_constructor] [state_definition]
                 [assertion_definition]

function_body= compound_statement [rescue_block]

init_constructor= init_head "(" [argument_expression] ")"

init_head= ":" type_identifier

state_definition= "at" identifier

assertion_definition= [pre_condition] [pos_condition]
```

```
pre_condition= "pre" "(" expression ")"

pos_condition= "pos" "(" expression ")"

rescue_block= "rescue" compound_statement
```

A.2 Declarações

```
declaration= declaration_specifier [ init_declarator_list ] ";"

declaration_specifier= storage_class
          | storage_class declaration_specifier
          | type_specifier
          | type_specifier declaration_specifier
          | type_qualifier
          | type_qualifier declaration_specifier

type_qualifier= "const" | "volatile"

type_specifier= basic_type
          | type_declarator
          | struct_or_union_specifier
          | enum_specifier

basic_type= "int"    | "char"   | "short"    | "long" | "float"
          | "double" | "signed" | "unsigned" | "void"

type_declarator= type_identifier [ "<" generic_argument_list ">" ]

generic_argument_list= generic_declarator { "," generic_declarator }

generic_declarator= type_declarator
          | basic_type

storage_class= "auto" | "register" | "static" | "extern"
          | "typedef" | "inline"   | "persistent"

struct_or_union_specifier= struct_or_union
              "{" struct_declaration_list "}"
          | struct_or_union identifier
```

```
            [ "{" struct_declaration_list "}" ]

struct_or_union= "struct" | "union"

struct_declaration_list= struct_declaration { struct_declaration }

struct_declaration= specifier_qualifier_list struct_declarator_list ";"

specifier_qualifier_list= type_specifier
          | type_specifier specifier_qualifier_list
          | type_qualifier
          | type_qualifier specifier_qualifier_list

struct_declarator_list= struct_declarator { "," struct_declarator }

struct_declarator= declarator [bit_field_size]
          | bit_field_size

bit_field_size= ":" constant_expression

enum_specifier= "enum" "{" enumerator_list "}"
          | "enum" identifier [ "{" enumerator_list "}" ]

enumerator_list= enumerator { "," enumerator }

enumerator= identifier [ "=" constant_expression ]

init_declarator_list= init_declarator { "," init_declarator }

init_declarator= declarator [ "=" initializer ]

initializer= assignment_expression
          | "{" initializer_list [","] "}"

initializer_list= initializer { "," initializer }

declarator= [pointer] direct_declarator

direct_declarator= identifier
          | "init"
          | "finish"
          | "(" declarator ")"
          | direct_declarator "[" [constant] "]"
```

```
          | direct_declarator "(" [parameter_type_list] ")"

pointer= "*" [type_qualifier_list] { "*" [type_qualifier_list] }

type_qualifier_list= type_qualifier { type_qualifier }

parameter_type_list= parameter_list [ ',' "..." ]

parameter_list= parameter_declaration { "," parameter_declaration }

parameter_declaration= declaration_specifier
          | declaration_specifier declarator
          | declaration_specifier pointer

type_name= type_specifier [pointer]

type_identifier= identifier
```

A.3 Classes

```
class_definition= class_head class_body
          | "class" identifier ";"

class_head= ["abstract"] class_head_definition

class_head_definition= "class" identifier class_head_specifier
          | "class" type_identifier class_head_specifier

class_head_specifier= [generic_parameter_spec] [super_class_spec]
            [friend_class_spec]

generic_parameter_spec= "<" generic_parameter_list ">"

generic_parameter_list= generic_declaration { "," generic_declaration }

generic_declaration= identifier [ ":" type_identifier ]

super_class_spec= ":" type_declarator

friend_class_spec= "friend" friend_class_list
```

```
friend_class_list= type_identifier { "," type_identifier }

class_body= "{" class_body_definition "}" [rescue_block]

class_body_definition= [ access_specifier ":" ] member_list
                       [invariant_spec]

member_list= member_declaration
            { [ access_specifier ":" ] member_declaration }

member_declaration= declaration_specifier member_declarator_list ";"
           | function_definition
           | "state"  identifier "=" assignment_expression ";"

member_declarator_list= declarator { "," declarator }

access_specifier= "public" | "private"

invariant_spec= "invariant" ":" "(" expression ")"
```

A.4 Comandos

```
statement= labeled_statement
         | expression_statement
         | compound_statement
         | selection_statement
         | iteration_statement
         | jump_statement
         | retry_statement
         | check_statement

labeled_statement= identifier ":" statement
         | "case" constant_expression ":" statement
         | "default" ":" statement

expression_statement= [expression] ";"

compound_statement= "{" compound_statement_body "}"

compound_statement_body= [declaration_list] [statement_list]
```

```
declaration_list= declaration { declaration }

statement_list= statement { statement }

selection_statement= "if" "(" expression ")" statement
           [ "else" statement ]
         | "switch" "(" expression ")" statement

iteration_statement= "while" "(" expression ")" statement
         | "do" statement "while" "(" expression ")" ";"
         | "for" "(" [expression] ";" [expression] ";" [expression] ")"
           statement

jump_statement= "goto" identifier ";"
         | "continue" ";"
         | "break" ";"
         | "return" [expression] ";"

retry_statement= "retry" ";"

check_statement= "check" integer_constant "(" expression ")" ";"
```

A.5 Expressões

```
expression= assignment_expression { "," assignment_expression }

constant_expression= conditional_expression

assignment_expression= conditional_expression
          | unary_expression assignment_operator assignment_expression
          | new_expression

assignment_operator= "="   | "*=" | "/=" | "%=" | "+=" | "-=" | "<<="
          | ">>=" | "&=" | "^=" | "|="

new_expression= "new" type_declarator [ "(" [argument_expression] ")" ]


conditional_expression= logical_OR_expression
          | logical_OR_expression "?" expression ":"
            conditional_expression
          | logical_OR_expression TK_IS TK_TYPE_NAME
```

```
          | logical_OR_expression TK_IN logical_OR_expression

logical_OR_expression= logical_AND_expression
          | logical_OR_expression "||" logical_AND_expression

logical_AND_expression= inclusive_OR_expression
          | logical_AND_expression "&&" inclusive_OR_expression

inclusive_OR_expression= exclusive_OR_expression
          | inclusive_OR_expression "|" exclusive_OR_expression

exclusive_OR_expression= AND_expression
          | exclusive_OR_expression "^" AND_expression

AND_expression= equality_expression
          | AND_expression "&" equality_expression

equality_expression= relational_expression
          | equality_expression "==" relational_expression
          | equality_expression "!=" relational_expression

relational_expression= shift_expression
          | relational_expression "<"  shift_expression
          | relational_expression ">"  shift_expression
          | relational_expression "<=" shift_expression
          | relational_expression ">=" shift_expression

shift_expression= additive_expression
          | shift_expression "<<" additive_expression
          | shift_expression ">>" additive_expression

additive_expression= multiplicative_expression
          | additive_expression "+" multiplicative_expression
          | additive_expression "-" multiplicative_expression

multiplicative_expression= cast_expression
          | multiplicative_expression "*" cast_expression
          | multiplicative_expression "/" cast_expression
          | multiplicative_expression "%" cast_expression

cast_expression= unary_expression
          | "(" type_name ")" cast_expression
```

```
unary_expression= postfix_expression
          | "++" unary_expression
          | "--" unary_expression
          | unary_operator unary_expression
          | "sizeof" unary_expression
          | "sizeof" "(" type_name ")"
          | "delete" postfix_expression

unary_operator= "&" | "*" | "+" | "-" | "!" | "~"

postfix_expression= primary_expression
          | postfix_expression "[" expression "]"
          | postfix_expression "(" [argument_expression] ")"
          | postfix_expression "." identifier
          | postfix_expression "->" identifier
          | postfix_expression "++"
          | postfix_expression "--"

argument_expression= assignment_expression { "," assignment_expression }

primary_expression= constant
          | identifier
          | "(" expression ")"
          | "self"
          | "super"
          | "initial" "(" expression ")"
          | "result"
          | "exception"
```

A.6 Constantes

```
constant= integer_constant | float_constant
          | char_constant  | set_constant
          | string

set_constant= "[" [set_element_list] "]"

set_element_list= set_element { "," set_element }

set_element= integer_constant [ ".." integer_constant ]
```