5

Vinculações e Verificação de Tipos

"We know what we are, but know not what we may be."

—William Shakespeare

De forma geral, o termo vinculação se refere à associação entre elementos [54]. A vinculação de um tipo a uma variável, por exemplo, associa aquela variável aos possíveis valores que ela pode assumir. A idéia de vinculação pode estar relacionada desde a decisões no projeto da linguagem, como a quais símbolos serão vinculados às operações aritméticas, como até a vinculação de valores às variáveis em tempo de execução. Neste capítulo, discutimos o conceito de vinculação em geral, e em particular a vinculação de tipos e armazenamento de variáveis para que possamos proceder com a verificação de tipos. A avaliação destes elementos são fundamentais para a compreensão das linguagens quanto à sua expressividade, flexibilidade e segurança.

As vinculações de tipos às variáveis e abstrações determinam em grande parte a expressividade da linguagem. Uma vez que os tipos são vinculados, precisamos de

mecanismos que verifiquem se os tipos das variáveis estão sendo usados apropriadamente dentro do programa. Neste capítulo, apresentamos também os conceitos relacionados à verificação de tipos: equivalência e compatibilidade de tipos. Esses conceitos são fundamentais ao programador para o uso adequado dos sistemas de tipos existente nas linguagens e de que forma eles são comparados.

5.1 VINCULAÇÕES

A idéia de vinculação desenvolve-se com a associação de entidades de um programa (tal como variáveis, funções, etc) a atributos relativos àquelas entidades no contexto do programa (um valor, um tipo, uma estrutura ou ainda uma abstração). Cada entidade do programa só pode ser processada quando estes atributos são definidos de forma exata, e a esta definição exata dos atributos chamamos de vinculação. Isso significa que para cada entidade do programa existe um descritor que contém todos os atributos da entidade.

Um programa na linguagem C, por exemplo, possui um conjunto de declarações de variáveis seguido de uma seqüência de comandos. Na declaração de variáveis, vinculamos identificadores a um espaço de memória que pode guardar valores de um determinado tipo. Neste caso, temos a vinculação de tipo mais a vinculação de um endereço de memória destinado ao conteúdo da variável. Da mesma forma, quando uma função é declarada, vinculamos um identificador a um processo juntamente com um conjunto de parâmetros formais, etc. Assim, o contexto de um programa é formado pelas suas várias vinculações (de tipos, armazenamento, valores, etc) a identificadores.

Vinculação é, portanto, um conceito fundamental relativo ao projeto de linguagens de programação: as entidades existentes, os atributos necessários a serem vinculados a cada entidade e o tempo em que cada vinculação ocorre [4]. Ainda sobre o projeto de linguagens, as questões sobre vinculações dizem respeito a: **o que** pode ser vinculado em uma linguagem de programação (quais entidades), **de que forma** e **quando** estes elementos podem ser vinculados.

O momento em que uma vinculação desenvolve-se é chamado de **tempo de vinculação**, enquanto a **forma de vinculação** mostra como os atributos são associados às entidades na linguagens. De uma forma geral, podemos vincular tipos, referências e valores a identificadores de variáveis, valores a identificadores de constantes, novos valores e estruturas a identificadores de tipos, abstrações de processos a identificadores de funções e procedimentos, abstrações de tipos de dados a identificadores de tipos ou classes, tratamento de exceções de tipos, etc; esses são os elementos vinculáveis (**o quê**).

Nas seções que seguem tratamos o tempo e formas de vinculações relativas apenas a variáveis e tipos. As vinculações relativas a abstrações de processos e tipos serão tratadas no Capítulo 7.

5.1.1 Formas de Vinculação

Grande parte das linguagens de programação atuais possui alguns tipos predefinidos, sejam eles primitivos ou compostos (visto nos Capítulos 3 e 4). Dessa forma, todos os tipos predefinidos de uma linguagem são vinculados *a priori* a suas respectivas palavras reservadas. Na linguagem C, por exemplo, a palavra reservada `int` identifica o tipo inteiro, o qual, por sua vez, possui alguns atributos inerentes, tais como as operações aritméticas e relacionais. Essas vinculações são decididas no projeto da linguagem e, por isso, implementadas nos processadores das mesmas (elementos definidos pelo projetista da linguagem e não pelo programador).

Contudo, a maioria das linguagens atuais permitem a criação de novos tipos a partir de novos valores, os tipos enumerados, ou ainda a partir de tipos preexistentes, como é o caso de novas estruturas que são nomeadas como tipos. Para a definição de um novo tipo na linguagem C, por exemplo, precisamos criar uma nova estrutura (`struct`) ou definir um novo identificador de tipo para um elemento já existente (`typedef`). Em todos esses casos temos uma declaração explícita de um novo tipo, seja por uma enumeração, uma estrutura ou renomeação de tipos. Para cada um destes casos, vinculamos um identificador a cada um dos novos elementos de forma explícita, por uma **declaração explícita** dos novos elementos. Assim, uma vez vinculado um

identificador a um novo tipo, ele pode ser usado de forma semelhante aos tipos predefinidos, aumentando a flexibilidade das linguagens de programação.

Algumas outras linguagens de programação, tais como as linguagens orientadas a objetos, têm a capacidade de vincular abstrações além das estruturas, os tipos abstratos de dados. Postergamos uma discussão sobre vinculações relativas a tipos abstratos de dados para o Capítulo 7.

Por outro lado, as variáveis são criadas para armazenar valores de determinados tipos, os quais podem ser preexistentes ou criados (como discutido acima). A vinculação de tipos às variáveis determina o universo de valores que podem ser armazenados em uma dada variável.

Um tipo pode ser vinculado a uma variável por uma **declaração explícita**, a qual é uma instrução da linguagem que lista nomes de variáveis e especifica o tipo associado. Este tipo de declaração é comum na grande maioria das linguagens de programação atuais e pode ser vista nos vários exemplos já apresentados no capítulo anterior.

Uma outra forma de declaração de variáveis é a chamada **declaração implícita**, na qual o tipo da variável é inferido em vez de explicitamente declarado. Uma forma comum de declaração implícita aparece nas primeiras linguagens como Fortran, PL/I e BASIC. Em Fortran, por exemplo, uma variável que não é explicitamente declarada mas aparece no corpo do programa é implicitamente declarada segundo uma convenção: identificador iniciado pelas letras de I a N são do tipo inteiro (`INTEGER`), enquanto os identificadores iniciados pelas outras letras são implicitamente declarados do tipo real (`REAL`).

Das linguagens mais modernas, SML e Perl possuem declarações implícitas. Na linguagem Perl, por exemplo, qualquer identificador que inicie com o caractere $ é um escalar que pode armazenar uma cadeia de caracteres ou um valor numérico. Os identificadores que iniciam com o caractere @ são *arrays*, enquanto os que iniciam com % são estruturas.

Além da vinculação de tipos às variáveis, estas também devem ser vinculadas ao espaço de memória no qual o conteúdo da variável será armazenado. Classificamos as variáveis quanto à sua existência no Capítulo 4 (Seção 4.3). Tal classificação denota a vinculação das variáveis em relação ao atributo de armazenamento. As variáveis

globais e locais têm um espaço de memória vinculado ao identificador da variável mediante a declaração da variável no programa, seja esta uma declaração explícita ou implícita. Por outro lado, as variáveis *heap* têm vinculação do seu atributo de espaço de memória mediante um comando de alocação explícita da variável no decorrer da execução do programa. Neste último caso, a vinculação de armazenamento é realizada de forma explícita com o comando de alocação de memória, enquanto nos outros casos a vinculação do atributo de armazenamento é realizada implicitamente junto com a declaração da variável.

5.1.2 Tempo de Vinculação

Além da forma de vinculações relativa aos tipos e variáveis, uma outra característica importante no projeto das linguagens é **quando** as vinculações são realizadas. Uma vinculação é dita **estática** se ocorrer antes do tempo de execução e permanecer inalterada ao longo da execução do programa. Por outro lado, ela é dita **dinâmica** se ocorrer em tempo de execução ou se for modificada ao longo da execução.

Para os tipos predefinidos das linguagens, as vinculações aos tipos estão embutidas nos próprios processadores das linguagens e, por isso, permanecem inalterados ao longo da computação dos programas nos sistemas monomórficos (Seção 5.3). Para os novos tipos construídos pelo programador, a vinculação aos identificadores de tipos é realizada estaticamente, dado que todos esses elementos são previamente declarados e a vinculação ao identificador do tipo é realizada no momento da declaração.

Por outro lado, o tempo da vinculação de tipos às variáveis pode ser **estático** ou **dinâmico**. Nas linguagens com vinculação estática de tipo, as variáveis têm seus tipos definidos em tempo de compilação. A maioria das linguagens que possui declaração explícita de tipos tem vinculação estática de tipos. Algumas linguagens com declaração implícita de tipos como Fortran também definem os tipos das variáveis estaticamente.

Em linguagens com vinculação de tipos estática, as operações realizadas sobre as variáveis podem também ser verificadas com relação aos tipos em tempo de compilação. Alguns possíveis erros de tipos cometidos pelo programador podem então ser detectados pelos compiladores. Vale ressaltar que algumas linguagens (ou

seus processadores) fazem correções automáticas de tipos sem advertir o programador (coerção de tipos, Seção 5.4.1).

Em linguagens com vinculação dinâmica de tipos os valores possuem um tipo fixo, mas as variáveis ou parâmetros das funções (as quais serão vistas no Capítulo 7) não possuem um tipo predeterminado. Nestas linguagens, as variáveis podem assumir valores de tipos diferentes em diferentes estágios da computação do programa. A conseqüência disso é maior flexibilidade nestas linguagens, nas quais programas podem ser "genéricos" porque os tipos são vinculados às variáveis mediante os dados.

Exemplo 5.1 – Nas linguagens Lisp, Smalltalk, APL e SNOBOL4, por exemplo, a vinculação de uma variável a um tipo é dinâmica. Na linguagem APL, podemos ter uma mesma variável vinculada a tipos diferentes em um mesmo program:

```
list <- 15.8 5.7 3.2  (1)
...
list <- 10            (2)
```

Em (1), `list` é um *array* de valores reais definido no momento em que a variável recebe os dados. Em um estágio posterior (2), `list` passa a ser uma variável do tipo inteiro. □

Apesar da flexibilidade do código quando a vinculação de tipos é realizada dinamicamente, existem algumas desvantagens. Primeiro, erros de mistura indevida de tipos não podem ser detectados pelo compilador, uma vez que uma mesma variável pode assumir tipos diferentes em um mesmo programa. Além disso, atribuições equivocadas podem inclusive não resultar em erros em tempo de execução, e para isso o programador precisa de um cuidado extra. Outra desvantagem está relacionada ao custo da implementação. Os tipos devem ser verificados durante a execução, o que é mais difícil e degrada o desempenho. Além disso, cada variável deve ter um descritor para manter o tipo atual, e o espaço de armazenamento também deve ser gerenciado em tempo de execução, uma vez que os vários tipos requerem espaços de memória diferenciados.

Algumas linguagens orientadas a objetos, tal como Java, realiza a vinculação de tipos estaticamente quando há declaração explícita do tipo do objeto. Contudo,

alguns objetos são genéricos, podendo assumir o tipo de quaisquer objetos. Neste caso, faz-se necessária a vinculação dinâmica de tipos. Como objetos são mais que simples valores, discutiremos tais vinculações apenas no Capítulo 10. Algumas outras linguagens (funcionais), tais como SML, Miranda e Haskell, possuem um mecanismo de inferência de tipos (a ser discutido na Seção 5.4) que podem determinar tipos da maioria das expressões sem que o programador precise determinar os tipos dos parâmetros ou variáveis.

5.2 ESCOPO *VERSUS* VINCULAÇÕES

Um outro conceito relacionado à vinculação é o **escopo** dos elementos das linguagens, ou seja, quais os elementos da linguagem que são **visíveis** nas várias partes dos programas. Uma variável é visível em uma parte do programa se esta pode ser acessada naquela parte; tem os seus atributos vinculados naquele trecho do programa. As regras de escopo de uma linguagem determinam como uma ocorrência particular de um nome está associada à variável. Portanto, o conhecimento destas regras nos habilita a ler e escrever programas nas linguagens.

A maioria das linguagens de programação possui mecanismos para a definição de subprogramas, e em algumas podemos ter blocos que delimitam que todos os elementos ali definidos sejam apenas vistos naquele dado escopo. Assim, os blocos podem ser vistos como delimitadores de escopo. Para efeito de estudos de escopo de elementos vinculáveis em linguagens de programação, podemos considerar os subprogramas como blocos com características especiais de passagem de parâmetros.

Uma dada variável é dita **local** se está definida, e por isso só pode ser usada, dentro de um bloco do programa. Em contrapartida, as variáveis são ditas **não-locais** se são usadas dentro de um bloco do programa, mas foram definidas em um bloco mais externo. Dessa forma, as variáveis podem ser definidas e usadas dentro de um mesmo bloco ou ainda definidas e usadas em blocos distintos; as regras de escopo das linguagens definem como as variáveis (e todos os outros identificadores) podem ser usadas.

Um dado programa pode ter um único bloco. Neste caso, todas as variáveis são locais a este bloco e, por isso, podem ser usadas como variáveis locais. Na maioria das vezes, o programador divide os seus programas em blocos distintos, os quais podem ser disjuntos ou aninhados[1]. As regras gerais para visibilidade dos identificadores permitem que todas as variáveis locais sejam visíveis nos blocos aonde foram declaradas, e além disso, podem ser vistas como variáveis não-locais nos blocos mais internos ao qual a dada variável foi declarada.

Quando uma mesma variável é declarada em blocos aninhados, a regra de escopo determina que a variável a ser considerada seja a de maior proximidade de declaração. Dessa forma, a variável local, ou do aninhamento mais próximo é usada.

Neste ponto, cabe-nos fazer uma distinção entre a declaração de um bloco, ou a vinculação do bloco, e a aplicação de um bloco quando o mesmo é um subprograma (a chamada de uma função por exemplo). A idéia de variáveis não-locais efetivamente usadas na aplicação de blocos depende se a vinculação do escopo é **estática** ou **dinâmica**. Considere o exemplo abaixo, para em seguida discutirmos a diferença entre os escopos estáticos e dinâmicos.

Exemplo 5.2 – Dado um programa, com subprogramas, em uma linguagem esquemática:

```
program main;
 var x : integer;        (1)
 procedure sub1;
   var x: integer;       (2)
   begin
    ...x...;             (3)
   end; {sub1}
 procedure sub2;
   begin
    sub1;
```

[1]Vale salientar que linguagens como C, C++ e Java não permitem aninhamento de sub-programa - um sub-programa definido dentro de outro.

```
    ...x...;              (4)
  end; {sub2}
begin {main}
 ...
end. {main}
```

Como podemos ver, a variável `x` é definida no subprograma `sub1` (2) e no programa principal (1). O uso desta variável no subprograma `sub1` (3) usará a definição local, assim como no programa principal a declaração da variável no programa principal será usada. Contudo, o uso de `x` no subprograma `sub2` (4) depende da vinculação de escopos, discutido a seguir. □

Escopo estático: para as linguagens com escopo estático, a relação entre os blocos é realizada em tempo de compilação. No exemplo acima temos o programa principal (`main`) e os subprogramas `sub1` e `sub2` são blocos dentro do programa principal. As regras de escopo têm como princípio o uso das declarações do escopo mais interno para o mais externo respeitando a ordem de hierarquia dos blocos.

A variável `x` do exemplo acima é declarada tanto no programa principal quanto no subprograma `sub1`. O uso desta no subprograma `sub1` será obviamente a variável `x` declarada nele próprio, e a mesma relação se aplica ao programa principal. Mas qual `x` será usado no subprograma `sub2`? No caso do escopo estático, quando uma variável não é declarada no próprio bloco, o escopo de escrita imediatamente externo a ele é considerado, o bloco pai (pai-estático), e assim sucessivamente com os seus ancestrais até que seja encontrada uma declaração da variável. No exemplo acima, como `sub2` é um subprograma (um sub-bloco) de `main`, o `x` do `main` será considerado por estar no bloco pai do subprograma `sub2`. Note que esta é uma hierarquia sintática de blocos.

Escopo dinâmico: para as linguagens com escopo dinâmico, a vinculação das variáveis ao escopo é realizada em tempo de execução. Apesar da estratégia de procura pelas declarações de variáveis seguir a ordem de hierarquia de blocos para as variáveis não-locais, a idéia de escopo dinâmico está relacionada ao

pai-dinâmico e não pai-estático como no caso anterior. O pai-dinâmico é definido pela proximidade de uso na seqüência da computação, ou seja, a última declaração em tempo de execução.

No exemplo acima, o subprograma `sub2` está estaticamente definido como sub-bloco do `main`, mas o uso da variável `x` dentro do `sub2` é posterior à chamada do subprograma `sub1`. O `x` vinculado ao escopo de `sub2` neste ponto do programa é o último `x` usado, o `x` declarado em `sub1`. Assim, no escopo dinâmico, as variáveis vinculadas ao escopo dependem da ordem de execução do programa, ao invés da ordem de declaração de blocos.

O efeito do uso de escopo dinâmico sobre a programação deve ser observado. Se uma variável é local ao bloco, então o uso da dada variável no bloco será sempre vinculado àquela local. Contudo, se a variável for não-local, a sua vinculação depende da ordem de execução, a última vinculada na execução. A conseqüência disso é que em um mesmo bloco de comandos, um identificador pode ter significados diferentes, e o programador precisa ter a idéia precisa de qual variável está sendo usada.

A vinculação de escopo nas linguagens de programação determina o significado dos programas naquela linguagem, e é imprescindível ao entendimento dos programas. Apesar de trazer mais flexibilidade de programação, o escopo dinâmico carrega alguns problemas relativos à legibilidade e eficiência dos programas, uma vez que não podem ser determinados em tempo de compilação. Por essa razão, a grande maioria das linguagens de programação adota o escopo estático. A linguagem Lisp que inicialmente adotava o escopo dinâmico, tem vários de seus processadores atuais adotando o escopo estático.

Aqui discutimos apenas as idéias de vinculação de variáveis a escopos, mas estas são semelhantes para a vinculação dos outros elementos vinculáveis na linguagens de programação. Os escopos destes elementos serão discutidos à medida que forem apresentados.

5.3 SISTEMA DE TIPOS

Na maioria das linguagens, as declarações de variáveis, constantes e parâmetros de funções são definidas com um único tipo; não há alternativa para os tipos definidos. Em particular, como já visto anteriormente, a vinculação de tipos às variáveis é realizada de forma estática na maioria das linguagens. Em casos como estes, a verificação de tipos é feita de forma trivial. Estes sistemas de tipo são chamados de **monomórficos**.

Contudo, sistemas de tipo puramente monomórficos são insatisfatórios principalmente quando temos como objetivo a reutilização de software. Muitos dos algoritmos que descrevemos poderiam ser usados com diferentes tipos - os chamados algoritmos genéricos. Para sistemas de tipo puramente monomórficos, os algoritmos devem ser reescritos para cada um dos tipos. Além dos algoritmos, temos comumente estruturas de dados que poderiam ser descritas independentemente dos tipos, tais como listas, pilhas e filas. Mais uma vez, estas estruturas devem ser reescritas para cada tipo em particular em linguagens com sistemas de tipo monomórficos.

Sistemas de tipo de linguagens que permitam formas mais gerais de descrição de algoritmos e estruturas precisam embutir conceitos como ***sobrecarga*** (tradução do inglês *overloading*), que é a capacidade de um identificador denotar várias abstrações ao mesmo tempo; e **polimorfismo**, onde abstrações dão um tratamento uniforme a valores de diferentes tipos. Estes conceitos serão vistos em detalhes nas seções que seguem.

5.3.1 Monomorfismo

Uma linguagem tem um sistema de tipos monomórficos quando cada elemento declarado na linguagem possui um único tipo. A maioria das linguagens de programação imperativas possui um sistema de tipos monomórficos.

Um exemplo de tipos monomórficos são as constantes, variáveis, parâmetros e resultados de funções nas linguagens Pascal e C. Cada um destes elementos possui

um tipo determinado na própria declaração, o qual não pode ser modificado em tempo de compilação ou execução.

Exemplo 5.3 – Uma estrutura de pilha de inteiros e outra de caracteres podem ser definida na linguagem Pascal, por exemplo:

```
type stackint = ^ intnode;
     intnode  = record
                  elem : Integer;
                  next : ^stackint;
                end;

type stackchar = ^ charnode;
     charnode  = record
                   elem : Char;
                   next : ^stackchar;
                 end;
```

Neste caso, duas estruturas de pilha precisaram ser definidas, uma para caracteres e outra para inteiros, porque a linguagem não permite que a estrutura seja definida para um tipo genérico e depois instanciado para caracteres ou inteiros. □

Da mesma forma, estas estruturas também devem ser duplicadas na linguagem C, apesar de representarem um mesmo elemento estrutural, e de várias das operações sobre elas serem independentes dos dados.

Exemplo 5.4 – As pilhas de inteiros e caracteres podem ser definidas na linguagem C como segue:

```
struct stackint { int elem ;
                  struct stackint *next;
                }

struct stackchar { char elem ;
                   struct stackchar *next;
                 }
```

□

Operações sobre pilhas tais como `push` e `pop`, que inserem e retiram um elemento da pilha respectivamente, independem do valor guardado, interessa apenas de que forma a pilha está sendo tratada. Assim, o tratamento dado a uma pilha de inteiros para a inserção de um novo elemento é semelhante ao tratamento dado à inserção de um elemento na pilha de caracteres, a menos do tipo dos elementos da pilha. Contudo, para as pilhas de caracteres e as pilhas de inteiros definidas acima tais operações também precisam ser definidas em separado.

Tal repetição de código para algoritmos semelhantes se dá porque as linguagens Pascal e C são primordialmente monomórficas, com exceção de alguns operadores predefinidos nas linguagens. Em Pascal e C, por exemplo, todos os tipos de parâmetros e resultados de funções definidas pelo programador devem ser fixados na definição da função. Isso faz com que as abstrações definidas pelos programadores sejam puramente monomórficas nestas linguagens. Não significa, contudo, que estas linguagens sejam puramente monomórficas; algumas das abstrações predefinidas nas linguagens não são monomórficas. O comando `write` em Pascal, por exemplo, imprime valores de diversos tipos, não respeitando desta forma a característica de uniformidade da linguagem. Este é um caso típico de *sobrecarga*.

5.3.2 Sobrecarga

A sobrecarga é o uso de um mesmo identificador para operações diferentes, ou seja um mesmo identificador pode denotar comportamentos distintos. Como citado acima, o `write(E)` pode ser usado para imprimir inteiros, caracteres, etc, depende do tipo do argumento `E`. Assim, usamos um mesmo nome de comando com serviços distintos: de imprimir caracteres, de imprimir inteiros, etc., apesar de subjetivamente denotarem um mesmo serviço.

A sobrecarga de operadores é um auxílio à programação, como é o caso dos operadores aritméticos da maioria das linguagens. Os operadores de soma (+) e subtração (-), por exemplo, são usados tanto para números inteiros quanto reais na maioria das linguagens, sem comprometer a legibilidade dos programas. Por outro lado, o operador "`-`" é também usado em Pascal para denotar a diferença de conjuntos,

o que pode dificultar a legibilidade de programas em alguns casos. Da mesma forma, o &, na linguagem C, pode representar apontadores ou ainda o `AND` lógico quando duplicado (&&). Note que a sobrecarga de operadores, apesar de ser um auxílio em vários dos casos, pode comprometer a clareza da linguagem quando usada com finalidades muito distintas, como é o caso do & na linguagem C.

Mas se um mesmo identificador pode ser usado com significados diferentes em uma mesma linguagem, como o significado apropriado pode ser identificado quando aplicado nos programas? Por exemplo, o operador "-" será interpretado como subtração de inteiros, reais ou conjuntos em uma expressão da linguagem Pascal? Depende dos operandos usados na subtração. Em outras palavras, todos os operadores aritméticos nas linguagens Pascal e C, por exemplo, são operadores sobrecarregados que dependem apenas dos operandos para sabermos qual operador será usado em cada expressão.

Além dos operadores aritméticos predefinidos na maioria das linguagens, algumas permitem declarações de abstrações com identificadores sobrecarregados. Mas neste caso, qual a abstração que será usada na execução do programa quando a aplicação de um operador sobrecarregado é realizada? Em algumas linguagens estas abstrações são **independentes de contexto**, enquanto outros dependem de onde são aplicados, ou seja, são **dependentes de contexto**.

Sobrecarga independente de contexto: a abstração a ser aplicada depende dos tipos dos argumentos, os quais devem corresponder aos parâmetros da abstração. Nas linguagens Pascal, SML e C, por exemplo, as abstrações dependem exclusivamente dos parâmetros utilizados e não do contexto no qual estão inseridas suas aplicações. Este é o caso, por exemplo, do operador de divisão na linguagem C ("/"): a divisão de dois números inteiros terá como resultado um inteiro (truncado se for uma divisão inexata), independentemente se a variável à qual a expressão é atribuída é do tipo inteiro ou real.

Exemplo 5.5 – Na linguagem C, podemos ter uma divisão de inteiros atribuída a uma variável do tipo ponto-flutuante:

```
...
```

```
 int a, b;
 float x;
...
 x = a / b;
```

No caso acima, o valor de `x` será um real que corresponde à parte inteira da divisão de `a` por `b` porque os operandos da divisão são valores inteiros (a divisão de inteiros será aplicada). Quando uma divisão de números reais é necessária mesmo que os operandos sejam valores inteiros, uma conversão explícita de tipos se faz necessário:

```
...
 x = (float) a / (float) b;
```

Na conversão acima, o operador / a ser usado passa a ser o de divisão de ponto-flutuante porque os operandos são convertidos para ponto-flutuante antes da aplicação do operador. Note que neste caso uma conversão foi aplicada explicitamente, e o operador de divisão continua independente de contexto (depende apenas dos operandos). □

De forma geral, para linguagens que permitem sobrecarga independente de contexto, deve-se ter uma forma de identificar qual operador (abstração) a ser usada quando esta é aplicada no programa. Para isso, temos a seguinte regra:

Definição 5.1 *Suponha duas funções distintas,* $f1\colon S1 \rightarrow T1$ *e* $f2\colon S2 \rightarrow T2$. *Para linguagens que permitem apenas sobrecarga de operadores* **independentes de contexto**, *estas duas funções podem usar um mesmo identificador* I *se* $S1$ *e* $S2$ *são tipos distintos. Assim, a aplicação* $I(E)$ *usará a função* $f1$ *se* E *for do tipo* $S1$, *e* $f2$ *se* E *for do tipo* $S2$.

Note que desta forma podemos identificar a abstração a ser usada por causa do tipo dos argumentos usados. Este é o caso do operador de divisão mostrado no exemplo acima, no qual identificamos se o operador a ser aplicado é de divisão de inteiros ou reais dependendo única e exclusivamente dos operandos.

Vale salientar que nas linguagens Pascal, SML e C, apenas abstrações construídas na própria linguagem podem ser sobrecarregadas, não há como construí-las. Por outro lado, C++ e Fortran 90 permitem a criação de novas funções para a linguagem, mesmo que sobrecarreguem operadores da própria linguagem. Em Java os operadores predefinidos não podem ser sobrecarregados.

Além de sobrecarga de operadores pré-existentes, as linguagens C++, Fortran 90 e Java permitem a sobrecarga de abstrações construídas pelo programador, desde que respeite as regras de independência de contexto.

Sobrecarga dependente de contexto: a abstração a ser aplicada (função/operador) depende não só do tipo dos operandos a serem aplicados, mas também do tipo do resultado esperado na expressão aonde está sendo aplicado. Este é o caso, por exemplo, da linguagem Ada, onde podemos sobrecarregar o operador de divisão ("/") e este depende não apenas dos operandos, mas também do tipo da variável para quem o resultado da divisão está sendo atribuído.

Exemplo 5.6 – Definimos abaixo um pseudocódigo na linguagem Ada, onde um novo operador de divisão é definido e em seguida é aplicado para a divisão de inteiros.

```
...
 function ``/'' (m, n :Integer) return Float is
  begin
   return Float(m) / Float(n);
  end;
...
 a, b, n : Integer;
 x: Float;
...
 n = a / b;       (1) {divisão de reais -
                       uso do operador ``/'' predefinido}
 x = a / b;       (2) {divisão de inteiros -
                       uso do operador ``/'' novo}
```

...

O operador de divisão está predefinido e sobrecarregado em Ada, assim como nas outras linguagens. Aqui mostramos mais uma sobrecarga definida pelo programador que recebe dois números inteiros como argumento e realiza uma divisão de números reais, tendo o resultado deste último tipo. A aplicação do operador de divisão mostra em (1) o operador de divisão de inteiros predefinido na linguagem sendo usado, enquanto em (2) o novo operador é usado. □

Além da sobrecarga de operadores preexistentes, como mostrado no exemplo acima, Ada permite a criação de funções que podem ser distinguidas tanto por seus parâmetros quanto pelo seu resultado. No caso de subprogramas (sem resultado), estes devem ter parâmetros com tipos distintos.

De uma forma geral, à sobrecarga dependente de contexto também devem ser impostas algumas restrições para que a sua aplicação possa distinguir qual serviço usar.

Definição 5.2 *Suponha duas funções distintas,* $f1\colon S1 \rightarrow T1$ *e* $f2\colon S2 \rightarrow T2$*. Para linguagens que permitem sobrecarga de operadores* **dependentes de contexto***, estas duas funções podem usar um mesmo identificador* I *se* $S1$ *e* $S2$ *são tipos distintos, ou* $T1$ *e* $T2$ *são distintos.*

No caso acima, se $S1$ e $S2$ são tipos distintos, a aplicação $I(E)$ usará apenas os argumentos para decidir a função a ser aplicada. Contudo, se $S1$ e $S2$ são o mesmo tipo, a aplicação $I(E)$ usará a função $f1$ se $I(E)$ estiver em um contexto que requer um tipo $T1$ (atribuição do resultado a uma variável do tipo $T1$, por exemplo). Por outro lado, a função $f2$ será usada se $I(E)$ estiver em um contexto onde $T2$ é exigido. Note que esta é uma forma mais abrangente de sobrecarga de operadores se comparada à independente de contexto.

5.3.3 Polimorfismo

Em um sistema de tipos polimórficos, as abstrações operam de maneira uniforme sobre argumentos de famílias de tipos relacionados.

Nós devemos ter o cuidado de não confundir os conceitos de sobrecarga de tipos e polimorfismo. Na sobrecarga, usamos um mesmo identificador para oferecer um conjunto de abstrações; e estas abstrações não precisam estar necessariamente relacionadas. Este é o caso do operador "`-`" usado em Pascal para subtração de números inteiros, reais e ainda diferença de conjuntos; tais abstrações são substancialmente diferentes. Enquanto isso, o polimorfismo é uma propriedade que tem de uma única abstração ser usada para uma família de tipos; a abstração funciona de uma única forma, independentemente do tipo em uso. E, para isso, os tipos usados na aplicação de tal abstração estão relacionados.

Exemplo 5.7 – A função que verifica o final de arquivo predefinida na linguagem Pascal, `eof`, recebe como argumento uma variável de arquivo de qualquer tipo, e gera como resultado um valor-verdade (booleano), o qual é o resultado de teste de final de arquivo. Independentemente do tipo dos valores inseridos no arquivo, esta função opera de maneira uniforme apenas testando o final de arquivo. □

Outro aspecto que devemos ressaltar sobre polimorfismo é que este conceito aumenta a expressividade da linguagem. Enquanto na sobrecarga de abstrações os identificadores podem ser renomeados, e isso não acrescenta expressividade na linguagem, o polimorfismo enriquece a linguagem com o poder de se ter tratamento genérico para uma família de tipos, o que aumenta flexibilidade e reutilização de código na linguagem.

O polimorfismo pode ser aplicado a diferentes tipos de abstrações das linguagens. Podemos ter o polimorfismo sobre abstrações de processos (funções genéricas), polimorfismo sobre tipos (tipos parametrizados ou politipos).

Polimorfismo de abstrações: várias linguagens funcionais, tal como SML, embutem o conceito de polimorfismo para a definição de funções. Podemos definir funções com um tipo determinado, ou ainda deixar os parâmetros de uma função sem um tipo determinado.

Exemplo 5.8 – Uma função que recebe dois números inteiros e tem como resultado o segundo deles pode ser definida em SML como segue:

```
fun segundoint (x: int, y:int) = y
```

Esta função é do tipo `segundoint: (inteiro × inteiro) → inteiro`. Assim, apenas argumentos do tipo inteiro podem ser aplicados a esta função. Poderíamos, contudo, ter uma função que dá como resultado o segundo argumento para quaisquer tipos, inclusive para argumentos de tipos distintos. A função acima pode então ser redefinida para ser aplicada a argumentos de quaisquer tipos:

```
fun segundo (x, y) = y
```

A função agora é do tipo `segundo:` $(\alpha \times \gamma) \rightarrow \gamma$ e pode ser aplicada a argumentos de quaisquer tipos, tais como `segundo(2,a), segundo("maria", "clara") e segundo(2.5,true).` □

Os tipos genéricos α e γ, chamados de **politipos**, são instanciados com os tipos dos argumentos mediante a aplicação da função. Em cada uma das aplicações da função `segundo` acima, os politipos foram instanciados com tipos diferentes: (inteiro × caracter) → caracter, (string × string) → string e (real × booleano) → booleano respectivamente.

A quantidade de aplicações a diferentes tipos que uma função polimórfica pode ser aplicada corresponde à substituição sistemática por todas as possíveis instâncias de tipos que podem ser aplicados.

Tipos parametrizados: além de abstrações polimórficas, podemos ter tipos que possuem um outro tipo como parâmetro, e assim uma definição de tipo passa a ser polimórfica. É comum nas linguagens de programação atuais termos construções tais como *arrays*. Com estas construções predefinidas das linguagens podemos construir *arrays* de números inteiros, números reais, caracteres, etc. Assim, podemos construir novos tipos a partir dos tipos existentes. Os *arrays* são na verdade um tipo genérico que possui um outro parâmetro como tipo, o qual pode ser instanciado por outros tipos predefinidos na maioria das linguagens.

Um outro exemplo do uso de tipos parametrizados são os conjuntos encontrados em Pascal, Modula-2 e Modula-3. Da mesma for como os *arrays*, definimos

conjuntos de algum tipo (`set of real`, por exemplo), o que caracteriza um tipo parametrizado.

Na maioria das linguagens atuais que possuem um sistema de tipos monomórficos, apenas para elementos predefinidos na linguagem podemos ter parametrização de tipos, como discutido acima. Apenas linguagens que possuem o conceito de polimorfismo como princípio, tal como SML e as linguagens orientadas a objetos possuem mecanismos para definição de tipos paramétricos. A construção de tipos paramétricos está relacionada ao conceito de abstrações nas linguagens de programação, o qual será discutido em detalhes no Capítulo 7.

A maioria das linguagens de programação atuais é monomórfica para a construção de novos elementos nos programas. Contudo, isso não significa que essas linguagens são puramente monomórficas, a maioria embute sobrecarga e polimorfismo para elementos predefinidos da linguagem, como ilustrado acima. Sob o ponto de vista de projeto de linguagens de programação, estas possuem sistemas de tipos inconsistentes, pois dão tratamento diferenciado aos elementos predefinidos na linguagem e pelo programador. Estas limitações de polimorfismo são em geral guiadas pelas dificuldades ou baixa eficiência de implementações. Discutimos nas seções que seguem alguns destes aspectos relacionados à verificação de tipos.

O polimorfismo é um dos princípios do paradigma de orientação a objetos e será melhor descrito no Capítulo 10 sob a luz do paradigma e os outros conceitos envolvidos. Aqui, vale apenas salientar que as várias linguagens de programação orientadas a objetos possuem mecanismos para criação de abstrações de processos e tipos polimórficos. Além destas, as linguagens Ada e SML também possuem um sistema de tipos polimórficos tanto para abstrações quanto para tipos.

Um outro aspecto que precisamos observar para os sistemas de tipos é o tempo de vinculação das variáveis com tipos monomórficos ou polimórficos. Quando uma linguagem possui construções dos programadores puramente monomórficas, os tipos podem ser vinculados às variáveis de forma estática. Contudo, quando tipos polimórficos, ou pelo menos sobrecarregados, podem ser construídos pelos programadores, algumas das vinculações dependem dos dados usados, e aí só podem ser realizadas em tempo de execução dos programas (vinculação dinâmica).

5.4 VERIFICAÇÃO DE TIPOS

A **verificação de tipos** é a atividade de assegurar que os operandos sejam de tipos compatíveis. Um tipo é **equivalente** (ou **compatível**) quando ele é válido para o operador ou tem permissão, segundo as regras da linguagem, para ser convertido automaticamente para um tipo válido (**coerção de tipos**). Um **erro de tipo** é tipicamente o uso indevido de operadores com operandos.

Se todas as vinculações a tipos forem realizadas de forma estática na linguagem, então a verificação de tipos pode ser feita estaticamente, em tempo de compilação. Se, contudo, houver vinculação dinâmica de tipos na linguagem, os tipos precisam ser também verificados em tempo de execução. APL, SNOBOL4 e Smalltalk são exemplos de linguagens que precisam de verificação dinâmica de tipos dado que possuem vinculação dinâmica de tipos.

O custo da verificação puramente estática é menor que a dinâmica, mas a vinculação dinâmica permite mais flexibilidade. Além disso, nem todos os erros de tipo podem ser detectados pela verificação estática de tipos. Em linguagens que fornecem tipos que representem a união disjunta, tais como C, C++, Pascal e Fortran, não há como verificar os tipos relacionados a estas variáveis estaticamente, mesmos que a verificação de tipos dos outros tipos de variáveis seja realizada estaticamente, o que é o caso das linguagens aqui mencionadas.

5.4.1 Equivalência de Tipos

A forma mais simples de compatibilidade de tipos pode ser definida pela equivalência de nomes. Dizemos que duas variáveis têm compatibilidade de nomes, se elas são declaradas com o mesmo tipo (mesmo nome de tipo). Sob este ponto de vista, a equivalência de tipos é definida como segue:

Definição 5.3 *Dados dois tipos de dados T e T', eles são ditos **nominalmente equivalentes**, se e somente se $T = T'$ (mesmo nome, definido no mesmo local).*

Se em uma dada linguagem a equivalência de tipos é definida por sua equivalência nominal, então cada novo tipo só pode ser equivalente a ele próprio.

Exemplo 5.9 – Suponha a definição de duas variáveis com tipos diferentes na linguagem Pascal:

```
type Int1 = Integer;
     Int2 = Integer;
var v1: Int1;
    v2: Int2;
...
```

Quando a equivalência de nomes é considerada, as variáveis `v1` e `v2` não possuem tipos equivalentes, dado que são declaradas por tipos nominalmente diferentes: tipos definidos em locais diferentes (apesar de possuírem o mesmo tipo base). □

É comum termos em programas um operador que espera receber operandos de um dado tipo T que, em vez disso, recebe operandos do tipo T' como mostra o exemplo abaixo.

Exemplo 5.10 – Considere o seguinte trecho de código na linguagem C:

```
int i;          {Tipo T}
float f1, f2;   {Tipo T'}
...
f1 = i + f2;
^    ^   ^
T'   T   T'     {Tipos dos operandos acima}
```

No caso acima, o operador + está sendo usado com operandos de tipos diferentes; um inteiro e outro de ponto-flutuante. □

Quando tal operação é verificada quanto ao tipo, pode-se simplesmente notificar que tais operandos não são do mesmo tipo. Outra alternativa seria verificar se os tipos T e T' são equivalentes, denotado por $T \equiv T'$. Contudo, o conceito de equivalência de tipos varia entre linguagens de programação.

Certas linguagens usam a relação de **equivalência estrutural**:

Definição 5.4 *Dados dois tipos de dados T e T', eles são ditos **estruturalmente equivalentes**, escrito por $T \equiv T'$, se e somente se T e T' têm o mesmo conjunto de valores.*

A equivalência estrutural é assim chamada porque para a maioria dos tipos seria impossível verificar cada um dos valores, dado que grande parte deles constitui um conjunto infinito de valores. Então, a forma de verificar a equivalência de tipos é através da redução dos tipos para os elementos base.

Exemplo 5.11 – Suponha uma linguagem com os tipos definidos em termos de produto cartesiano, união disjunta e mapeamentos. A equivalência de tipos (T e T') pode ser verificada como segue:

- T e T' são tipos primitivos. Então, $T \equiv T'$ se e somente se T e T' são idênticos.
- $T = A \times B$ e $T' = A' \times B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$.
- $T = A + B$ e $T' = A' + B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$ ou $A \equiv B'$ e $B \equiv A'$.
- $T = A \to B$ e $T' = A' \to B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$.
- para quaisquer outros casos, T e T' não são equivalentes.

□

Apesar de as regras acima serem simples, a verificação de tipos pela equivalência estrutural pode ser complexa quando a definição de tipos recursivos é permitida.

Grande parte das linguagens atuais utiliza uma mistura destas duas abordagens para a equivalência de tipos. A *ISO Standard Pascal* , por exemplo, define as regras de sua compatibilidade de tipos na qual a maioria é definida por equivalência nominal

de tipos, mas tipos podem ser definidos por nomes de outros tipos, e neste caso são considerados equivalentes (a chamada **equivalência de declaração**).

Exemplo 5.12 – Suponha agora um tipo inteiro definido a partir de outro, o que é o caso de `Int2` abaixo:

```
type Int1 = Integer;
     Int2 = Int1;
...
```

neste caso, as variáveis definidas por quaisquer dos tipos acima possuem tipos equivalentes. □

Note que, apesar dessa flexibilidade facilitar em muitos casos a programação, nem sempre o programador deseja que elementos de novos tipos definidos possam trocar valores entre si. Em geral, novos tipos são definidos para propósitos específicos de modelagem do problema. Então, a troca de valores de variáveis com tipos diferentes pode retratar um uso equivocado das variáveis nos programas.

A linguagem Ada usa equivalência de nomes na sua verificação de tipos, mas provê a criação de subtipos e tipos derivados, os quais denotam explicitamente a relação entre tipos. Quando dois tipos são derivados de um outro, eles são incompatíveis entre si. Mas quando um tipo é um subtipo de outro, variáveis definidas como do subtipo são compatíveis com variáveis definidas como do tipo que deu origem. Desta forma, existe uma flexibilidade de tipos explicitamente controlada pelo programador.

A linguagem C usa equivalência estrutural para todos os tipos, exceto para as estruturas (`struct` e `union`), neste caso a equivalência de declarações é utilizada. Contudo, quando duas estruturas são definidas em arquivos distintos, a equivalência estrutural é utilizada.

As linguagens orientadas a objetos, tais como C++ e Java, fornecem compatibilidade de objetos, o que depende da relação de hierarquia entre os objetos. Tal questão será apenas discutida no Capítulo 7.

Uma outra forma de tornar compatíveis os tipos dos operandos ao do operador, e vastamente usada nas linguagens de programação atuais é a chamada **coerção** de tipos.

Este mecanismo usado nas linguagens faz um mapeamento de valores de um dado tipo para valores de um outro tipo automaticamente. Na linguagem Pascal, por exemplo, a função `sqrt` é definida para número reais, mas se um valor inteiro for passado como argumento, este é automaticamente transformado para um valor real correspondente. Apesar de parecer uma facilidade de programação, a coerção de tipos não é clara para o mapeamento de valores de qualquer tipo, além disso é incompatível com os conceitos de sobrecarga e polimorfismo requeridos nas linguagens de programação atuais. Dados esses fatores, as linguagens modernas evitam o uso de coerção de tipos o quanto possível. Esta só é permitida quando os mapeamentos de valores entre os tipos é clara e não se contra-põe aos outros conceitos embutidos nas linguagens.

5.4.2 Inferência de Tipos

Para linguagens que declaram explicitamente os tipos das suas variáveis, cada variável pode ser associada a um tipo proveniente de tal declaração. Vimos que esta vinculação tanto pode ser realizada em tempo de compilação quanto em tempo de execução. Algumas linguagens, contudo, não possuem declaração explícita dos tipos associados aos identificadores, mas mesmo assim ainda podem ser inferidos a partir de expressões e os operadores usados.

Um exemplo típico de uma linguagem que emprega o mecanismo de inferência de tipos é a linguagem SML. Esta é tipicamente uma linguagem funcional (na sua versão mais pura) que embute algumas características imperativas. Nela, a maioria das expressões pode ter seus tipos determinados a partir de seus operadores, uma vez que a linguagem permite que parâmetros e variáveis sejam usados sem um tipo previamente declarado.

Exemplo 5.13 – Uma função que calcula a circunferência de um círculo, dado o raio, pode ser definida como:

```
fun  circunf(r) = 3.14 * r * r;
```

Na função acima, o tipo de `r` não foi declarado, mas mesmo assim pode ser inferido. Como uma constante de número real é utilizada na expressão, o sistema de inferência de tipos pode concluir que os outros elementos da expressão são também números

reais. Desta forma, conclui-se que a função recebe um argumento do tipo real e produz como resultado um valor do tipo real. Em outras palavras, o compilador é capaz de verificar que esta é uma função com um único tipo, `circunf: real` $\rightarrow$ `real`. □

Os politipos também podem ser inferidos pelo sistema de verificação de tipos. A função `segundo` definida anteriormente pode ter seus politipos detectados pelo sistema de inferência.

Exemplo 5.14 – Uma função que dá como resultado o segundo argumento para quaisquer tipos, inclusive para argumentos de tipos distintos, pode ser definida como segue:

```
fun segundo (x, y) = y
```

A função é do tipo `segundo:` $(\alpha \times \gamma) \rightarrow \gamma$, ou seja, politipos denotados pelos elementos genéricos α e γ. □

Da mesma forma, várias outras expressões podem ter seus tipos inferidos a partir dos operadores. SML rejeita, contudo, expressões para as quais os tipos não possam ser inferidos.

Exemplo 5.15 – Uma função que calcula o quadrado de um número:

```
fun  quadrado(x) = x * x;              {definição ERRADA}
```

Na função acima o tipo de `x` não foi declarado. O sistema tenta então "descobrir" o tipo pela expressão. Contudo, como o operador `*` pode tanto ser usado para números inteiros quanto para números reais, o sistema de inferência de tipos não consegue decidir qual destes tipos deve ser usado. Por outro lado, não se pode usar esta mesma operação para caracteres ou ainda para listas, o que descaracteriza que o tipo possa ser algo genérico. Desta forma, o sistema rejeita esta definição porque não há como decidir os seus tipos. □

Em contrapartida, a definição da função `quadrado` acima seria aceita se o tipo do parâmetro fosse definido.

Exemplo 5.16 – Uma função que calcula o quadrado de um número inteiro:

```
fun  quadrado1(x: int) = x * x;
```

Como na função acima o tipo de x é inteiro, o sistema de tipos conclui que esta é uma função que recebe um valor inteiro e produz como resultado um valor inteiro. As formas seguintes também são aceitas pela linguagem:

```
fun  quadrado2(x) = (x:int) * x;
fun  quadrado3(x) = x * (x:int);
fun  quadrado4(x) = (x:int) * (x:int);
```

□

As linguagens funcionais Haskell e Miranda também possuem um sistema de inferência de tipos.

Note que este mecanismo de inferência de tipos difere das regras predefinidas em Fortran que relaciona o nome da variável a um tipo predefinido (Seção 5.1.1). No caso dessas linguagens funcionais, o sistema precisa decidir o tipo como politipo, quando as abstrações são polimórficas, ou de um tipo específico dependendo de como os parâmetros são usados dentro da função.

5.5 LINGUAGENS FORTEMENTE TIPIFICADAS

Uma das características importantes para uma linguagem de programação diz respeito à forma como ela está definida para verificar os tipos de dados. A não verificação, ou ainda a verificação inapropriada, dos tipos de dados em uma expressão da linguagem pode conduzir o programador a resultados equivocados e muitas vezes imperceptível. Uso inadequado, mistura, de tipos pode gerar resultados errados em uma operação aritmética, por exemplo. Dessa forma, a partir dos anos 70, junto com a programação estruturada, veio a idéia de linguagens **fortemente tipificadas** como aquelas que podem reconhecer todos os erros de tipo. Assim, o programador é advertido dos problemas relativos a tipos dos seus programas, seja em tempo de compilação ou execução.

Se cada identificador em uma linguagem é associado exclusivamente a um único tipo e todos os tipos são vinculados estaticamente, então o tipo de cada identificador

pode ser reconhecido em tempo de compilação. Mesmo podendo reconhecer os tipos associados a cada identificador, a localização do armazenamento à qual o identificador está associado pode armazenar valores de diferentes tipos em diferentes tempos. Considerando o requisito de linguagens fortemente tipificadas, os tipos de todos os operandos devem ser determinados durante a compilação ou em tempo de execução, e permitir a detecção, em tempo de execução, da utilização de valores de tipos incorretos em variáveis que podem armazenar valores de mais de um tipo.

Fortran, por exemplo, não é fortemente tipificada porque a relação entre parâmetros reais e formais não é verificada quanto ao tipo. Além disso, esta linguagem permite que uma variável (com o uso de `EQUIVALENCE`) de um tipo refira-se ao valor diferente, sem que o sistema seja capaz de verificar o tipo do valor quando uma variável é referenciada ou atribuída.

As linguagens Pascal e Modula-2 são quase fortemente tipificadas, salvo o uso da união disjunta (Seção 4.2.2.2). Uma variável do tipo união disjunta é definido por um registro com o descriminador, o qual é usado para verificar se a variável usada corresponde ao tipo usado no dado estágio da computação do programa. Contudo, a linguagem permite que o tipo seja omitido, e assim nenhuma verificação será realizada em tempo de execução.

As linguagens C e C++ não são fortemente tipificadas, uma vez que permitem a existência de funções para as quais os parâmetros não são verificados quanto ao tipo. Além disso, o tipo união disjunta provido pela linguagem também permite o uso de tipos diferentes em um mesmo espaço de memória sem a devida verificação.

Diferente de Pascal, Ada pode verifica os registros variantes (união disjunta) dinamicamente. Contudo, é uma linguagem quase fortemente tipificada porque possui a função de biblioteca `UNCHECKED_CONVERSION` a qual permite uma suspensão temporária de verificação tipos, burlando os princípios da tipificação forte das linguagens. Modula-3 também possui um procedimento com estes mesmos propósitos. Apesar de baseada em C e C++, Java é uma linguagem fortemente tipificada da mesma forma que Ada.

SML é uma linguagem fortemente tipificada. Todos os identificadores são vinculados estaticamente a partir da sua declaração, ou reconhecidos a partir de suas regras de inferência de tipos (Seção 5.4.2).

5.6 LEITURA RECOMENDADA

Os conceitos de polimorfismo, tipos abstratos e subtipos são descritos de maneira uniforme por Cardelli e Wegner em [21].

Juntamente com as teorias de tipos, vários estudos têm sido feitos sobre a implementação da verificação de tipos, seja em tempo de compilação ou de execução. Os leitores interessados no assunto podem consultar os Capítulos 6 e 7 de [3].

Inferência de tipos implementadas em SML e Miranda estão descritas em [69] e [19] respectivamente.

5.7 EXERCÍCIOS

1. Verifique se a vinculação de tipos é estática ou dinâmica na sua linguagem favorita. E a vinculação de armazenamento?

2. Explique a relação entre as variáveis *heap* e a vinculação dinâmica de tipo.

3. "A vinculação estática de tipos às variáveis pode ser implementada de forma eficiente e provê mais confiabilidade, mas a vinculação dinâmica de tipos provê mais flexibilidade". Mostre exemplos que comprovem tal afirmação. Mostre pelo menos um contra-exemplo sobre esta afirmação.

4. Suponha que uma linguagem inclua tipos enumerados definidos pelo usuário e que os valores da enumeração possam ser sobrecarregados; ou seja, o mesmo valor literal poderia aparecer em dois tipos enumerados diferentes:

```
type Cores = (vermelho, azul, amarelo);
     AlgumasCores = (vermelho, branco, verde);
```

O uso do valor `vermelho` não pode ser verificado quanto ao tipo. Proponha uma forma de permitir essa verificação de tipos sem desativar essa sobrecarga completamente.

5. Considere uma linguagem em que o operador "/"(divisão) é sobrecarregado para os tipos: `int X int` $\rightarrow$ `int`, e `int X int` $\rightarrow$ `real`. O procedimento "`write`"também é sobrecarregado nesta linguagem para todos os tipos primitivos.

   (a) Mostre um exemplo no qual o operador de divisão poderia gerar confusão de tipos.

   (b) A sobrecarga de procedimentos é dependente ou independente de contexto?

6. Considere uma linguagem em o operador "/"(divisão) é sobrecarregado para os tipos: `int X int` $\rightarrow$ `int`, e `real X real` $\rightarrow$ `real` (assim como todos os outros operadores aritméticos). A linguagem não possui conversão de tipos int-para-real, mas os literais int (1,2,...) são sobrecarregados com os tipos int e real.

   (a) Comente se a sobrecarga de literais pode ser realizada (sob quais condições), e se o sugerido acima poderia causar problemas.

   (b) A sobrecarga de literais é dependente ou independente de contexto?

7. Discuta a diferença entre polimorfismo e sobrecarga de identificadores. Mostre exemplos dessa diferença.