TAREFA – entrega no SIGAA até 19/01

Disciplina – Paradigmas de Linguagens de Programação

Prof Benedito Ferreira

O material abaixo é uma reprodução de uma parte do livro

ADVANCED PROGRAMMING LANGUAGE DESIGN de Raphael A. Finkel

Destacar os conceitos apresentados, copiando-os, juntamente com a definição que é dada pelo autor e dando, para cada conceito, pelo menos dois exemplos diferentes dos dados pelo autor, na linguagem de sua preferência (escolher entre Java, C, C++ ou Python).

3.1 Variáveis, tipos de dados, literais e expressões

Vou me referir repetidamente ao exemplo a seguir, que foi projetado para ter um pouco de tudo sobre tipos. Um tipo é um conjunto de valores nos quais as mesmas operações são definidas.

```
variable                                                    1
     First : pointer to integer;                            2
     Second : array 0..9 of                                 3
          record                                            4
               Third: character;                            5
               Fourth: integer;                             6
               Fifth : (Apple, Durian, Coconut, Sapodilla,  7
                    Mangosteen)                             8
          end;                                              9

begin                                                       10
     First := nil;                                          11
     First := &Second[1].Fourth;                            12
     First^ := 4;                                           13
     Second[3].Fourth := (First^ + Second[1].Fourth) *      14
          Second[First^].Fourth;                            15
     Second[0] := [Third : 'x'; Fourth : 0;                 16
          Fifth : Sapodilla];                               17
end;                                                        18
```

Figura 1.2

Linguagens imperativas (como Pascal e Ada) têm variáveis, que são posições de memória nomeadas. A Figura 1.2 apresenta duas variáveis, First (linha 2) e Second (linhas 3-9). As linguagens de programação geralmente restringem os valores que podem ser armazenados nas variáveis, tanto para garantir que os compiladores possam gerar código acurado para manipular esses valores quanto para evitar erros comuns de programação. As restrições geralmente se revestem da informação do tipo. O tipo de uma variável é uma restrição sobre os valores que ela pode conter e as operações que podem ser aplicadas a esses valores. Por exemplo, o tipo inteiro engloba valores numéricos inteiros dentro de uma faixa de valores, dependente da linguagem (ou da implementação); valores desse tipo podem atuar como operandos em operações aritméticas, como a adição. O termo integer não está grafado em negrito, porque na maioria das linguagens, os tipos predefinidos não são palavras reservadas, mas identificadores comuns que podem receber novos significados (embora isso não seja uma boa prática).

Pesquisadores desenvolveram várias taxonomias para categorizar os tipos [ISO / IEC 94; Meek 94]. Apresentarei aqui uma taxonomia bastante simples. Um tipo primitivo é aquele que não é construído a partir de outros tipos. Os tipos primitivos padrão fornecidos pela maioria das linguagens incluem inteiro, booleano, caractere, real e, às vezes, string. A Figura 1.2 inclui inteiro e

caractere. Os tipos enumerados também são primitivos. O exemplo usa um tipo enumerado nas linhas 7–8; seus valores são restritos aos valores especificados. Os tipos enumerados geralmente definem a ordem de suas constantes. Na Figura 1.2, entretanto, não faz sentido considerar uma fruta maior do que outra.

Tipos estruturados são construídos a partir de outros tipos. Vetores (ou matrizes), registros e ponteiros são tipos estruturados[1]. A Figura 1.2 mostra essas três modalidades de tipos estruturados padrão. Os blocos de construção de um tipo estruturado são seus componentes. Os tipos componentes fazem parte da criação do tipo estruturado; os valores dos componentes fazem parte do valor de um tipo estruturado. O tipo ponteiro na linha 2 da Figura 1.2 tem um tipo de componente (inteiro); um valor de ponteiro tem um valor componente. Há dez valores componente do tipo array nas linhas 3–9, cada um do tipo de registro. Em geral os vetores devem ser homogêneos; ou seja, todos os valores componentes devem ser do mesmo tipo. Vetores são indexados por elementos de um tipo de índice, geralmente um subintervalo de inteiros, caracteres ou um tipo enumerado. Portanto, uma matriz tem dois tipos componentes (o tipo base e o tipo de índice); ele tem tantos valores de componentes quanto membros do tipo de índice.

Vetores flexíveis não têm limites declarados; os limites são definidos em tempo de execução, com base em quais elementos da matriz tiveram valores atribuídos. Vetores de tamanho dinâmico possuem limites declarados, mas os limites dependem do valor, em tempo de execução, das expressões que definem os limites. As linguagens que provêem vetores de tamanho dinâmico fornecem sintaxe para se obter os limites inferior e superior em cada dimensão.

Fatias de vetor, como Second [3..5], também são componentes para os propósitos desta discussão. Linguagens (como Ada) que permitem fatias de array geralmente só permitem fatias na última dimensão. (APL não tem tal restrição.)

Os componentes do tipo registro nas linhas 4–9 são dos tipos caractere e inteiro. Registros são como vetores, no sentido de que têm vários valores componentes. No entanto, os valores são indexados não por membros de um tipo índice, mas sim por campos nomeados. Os valores dos componentes não precisam ser do mesmo tipo; registros não precisam ser homogêneos. As linguagens de programação de sistemas às vezes permitem que o programador controle exatamente quantos bits são alocados para cada campo e como os campos são agrupados na memória.

Escolha[2] é um tipo estruturado menos comum. É como um registro, pelo fato de possuir tipos de componentes, cada um selecionado por um campo. No entanto, possui apenas um valor componente, que corresponde a exatamente um dos tipos de componentes. Escolhas geralmente são implementadas alocando-se a quantidade de espaço que o maior tipo componente requer. Algumas linguagens (como Simula) permitem que o programador restrinja uma variável a um componente específico quando a variável é declarada. Nesse caso, é alocado apenas espaço suficiente para esse componente e o compilador não permite acesso a outros componentes.

Qual campo está ativo em um valor escolha determina as operações que podem ser aplicadas a esse valor. Geralmente, há alguma maneira de um programa determinar em tempo de execução qual campo está ativo em qualquer valor do tipo de escolha; caso contrário, existe o perigo de que um valor seja acidentalmente (ou intencionalmente) tratado como pertencente a um campo diferente, que pode ter um tipo diferente. Frequentemente, as linguagens fornecem uma especificação (tagcase) com ramificações nas quais a variante específica é conhecida tanto pelo programa quanto pelo compilador. Pascal permite que parte de um registro seja uma escolha e os outros campos estejam ativos em qualquer variante. Um dos últimos campos indica qual variante está em uso. Não faz sentido modificar o valor desse campo sem modificar igualmente a parte variante.

Um literal é um valor, geralmente de um tipo primitivo, expressamente denotado em um programa. Por exemplo, 243 é um literal inteiro e a Figura 1.2 tem os literais 0 , 1 , 3 , 4 , 9 e 'x' .

---

1 Denominar ponteiros como primitivos ou estruturados é assunto controvertido. Escolhi denominá-los estruturados porque são construídos a partir de outro tipo.

2 [Nota minha - Benedito Ferreira] – em geral, se denomina esse tipo de **união** (por exemplo, em C/C++, “union”) já que a união agrupa tanto elementos de um conjunto como do outro. A união de, por exemplo, reais e caracteres terá os elementos dos dois tipos, mas apenas um estará armazenado em um dado momento.

Alguns valores são providos como constantes pré-declaradas (ou seja, identificadores com valores predefinidos e imutáveis), como false (Boolean) e nil (ponteiro).

Um construtor denota expressamente um valor de um tipo estruturado. A Figura 1.2 tem um construtor de registro nas linhas 16–17.

Uma expressão é um literal, um construtor, uma constante, uma variável, uma invocação de uma função que retorne de valor, uma expressão condicional ou um operador com operandos que são, por sua vez, expressões. A Figura 1.2 contém expressões nas linhas 11–17. Um operador é um atalho para uma invocação de uma função com retorno de valor cujos parâmetros são os operandos. Cada operador tem uma aridade, ou seja, o número de operandos que ele espera. As aridades comuns são unárias (um operando) e binárias (dois operandos). Operadores unários são comumente escritos antes de seu operando (como -4 ou &myVariable), mas alguns são tradicionalmente escritos após o operando (como ptrVariableˆ). Às vezes é útil considerar literais e constantes como operadores com aridade zero (sem operando). Por exemplo, true é um operador com aridade zero do tipo Boolean.

Os operadores não necessariamente aceitam apenas operandos numéricos. O operador de desreferenciamento (ˆ / ou * em certas linguagens), por exemplo, produz o valor apontado por um ponteiro. Este operador é unário e pós-fixado, ou seja, vem após sua expressão (pré-fixado no caso de * de C/C++). Você pode ver exemplos na Figura 1.2 nas linhas 13, 14 e 15. Algumas linguagens, como Gedanken, Ada e Oberon-2, forçam os ponteiros (repetidamente, se necessário) para os valores que eles referenciam se o contexto deixar claro qual tipo é requerido. O operador unário pré-fixado de referência (&) na linha 12 gera um ponteiro para um valor.

Operadores comuns incluem os da tabela da próxima página. Em muitos operadores é realizada sobrecarga; ou seja, ela denota mais de uma operação e seu significado dependerá do número e dos tipos de operandos. É interessante conceber os operadores sobrecarregados como funções com várias definições, das quais o compilador escolhe aquela com o número e tipo de parâmetros apropriados.

Cada operador tem uma precedência atribuída, que determina a forma como a expressão é agrupada na ausência de parênteses. Na Figura 1.2, linhas 14-15, o significado provavelmente seria diferente sem os parênteses, porque a multiplicação geralmente tem uma precedência maior do que a adição.

Operator	Left type	Right type	Result type	Comments
+ - *	integer	integer	integer	
+ - * /	real	real	real	
/	integer	integer	real	or integer
div mod	integer	integer	integer	
-	numeric	none	same	
**	integer	integer	integer	exponentiation
**	numeric	real	real	exponentiation
=	any	same	Boolean	
< > >= <=	numeric	same	Boolean	
+	string	string	string	concatenation
~	string	pattern	Boolean	string match
and	Boolean	Boolean	Boolean	
or	Boolean	Boolean	Boolean	
not	Boolean	none	Boolean	
^	pointer	none	component	
&	any	none	pointer	

[Observação minha – Benedito Ferreira] - Uma questão deve merecer a atenção do programador, para se evitarem erros. Ver o comentário na linha onde há a divisão / entre inteiros, produzindo-se real. Ou inteiro ("or integer"). Nas linguagens que possuem o operador de divisão inteira **div**, a barra (/) faz divisão real, resultando um valor real. Porém, em C/C++ e Java, por exemplo, o tipo dos operandos é que determinará a operação. Se pelo menos um operando for real, a divisão será real. Se ambos forem inteiros, será feita a divisão inteira, resultando portanto interiro (será desprezado o resto).

As expressões são avaliadas para um valor-a-direita (R-value no original, R significando "right"). Variáveis e componentes de variáveis de tipos estruturados também possuem um valor-a-Esquerda (L-value no original, L significando "left"), ou seja, um endereço onde o valor-a-direita correspondente é armazenado. A instrução de atribuição (linhas 11 a 17 na Figura 1.2) requer um valor-a-esquerda no lado esquerdo e um valor-a-direita no lado direito. Na Figura 1.2, as linhas 11 e 12 mostram uma variável usada em seu valor-a-esquerda; as próximas linhas mostram os componentes usados em seus valores-a-esquerda.

Os tipos do lado esquerdo e do lado direito devem ser compatíveis para atribuição (assignment-compatible). Se forem do mesmo tipo, são compatíveis. (O que significa ter o mesmo tipo é discutido no Capítulo 3.) Se forem de tipos diferentes, a linguagem pode permitir que o valor do lado direito seja convertido implicitamente no tipo do lado esquerdo. As conversões de tipo implícitas são chamadas de coerções. Por exemplo, Pascal converterá forçosamente inteiros em reais, mas não o contrário. As coerções são propensas a erros, porque o tipo de destino pode não ser capaz de representar todos os valores do tipo de origem. Por exemplo, muitos computadores podem armazenar alguns números grandes precisamente como inteiros, mas apenas imprecisamente como reais.

A conversão de tipos, seja explicitamente (casting) ou implicitamente (coercing), às vezes pode alterar o formato dos dados. No entanto, às vezes é necessário tratar uma expressão de um tipo como se fosse de outro tipo, sem nenhuma conversão de formato de dados. Por exemplo, uma mensagem pode parecer um array de caracteres para uma função, enquanto outra função deve entendê-la como um registro com cabeçalho e campos de dados. Wisconsin Modula introduziu um operador ***qua*** de conversão explícita sem conversão de formato para este propósito. Em C, que não possui tal operador, o programador que desejar uma conversão sem conversão de formato deve converter um ponteiro para o primeiro tipo em um ponteiro para o segundo tipo; ponteiros têm a mesma representação, não importando o que eles apontam (na maioria das implementações C). O código a seguir mostra os dois métodos.

```
type                                                    1
     FirstType = ... ;                                  2
     SecondType = ... ;                                 3
     SecondTypePtr = pointer to SecondType;             4
variable                                                5
     F : FirstType;                                     6
     S : SecondType;                                    7
begin                                                   8
     ...                                                9
     S := F qua SecondType; -- Wisconsin Modula         10
     S := (SecondTypePtr(&F))^; -- C                    11
end;                                                    12
```

Figura 1.3

A linha 10 mostra como F pode ser convertida sem conversão de formato para o segundo tipo em Wisconsin Modula. A linha 11 mostra a mesma coisa para C, onde uso o nome do tipo

SecondTypePtr como uma rotina de conversão explícita. O operador de referência & produz um ponteiro para F. Em ambos os casos, se os dois tipos discordarem no comprimento da representação, pode ocorrer o caos, porque o número de bytes copiados pela atribuição é o número apropriado para SecondType .

Os operadores booleanos e (*and)* e ou (*or)* podem ter semântica de curto-circuito; isto é, o segundo operando só é avaliado se o primeiro operando for avaliado como true (para and) ou false (para or). Essa estratégia de avaliação é um exemplo de avaliação preguiçosa, discutida no Capítulo 4. Operadores de curto-circuito permitem que o programador combine testes, sendo que o segundo só faz sentido se o primeiro for bem-sucedido. Por exemplo, talvez eu queira testar primeiro se um ponteiro é nil e somente se não for, testar o valor para o qual ele aponta.

Expressões condicionais são constituídas por um construto if. Para garantir que uma expressão condicional sempre tenha um valor, cada if deve conter tanto um *then* quanto um *else* . As expressões nas partes then e else devem ter o mesmo tipo. Aqui está um exemplo:

write (se a > 0 então a else -a);