Introdução às variáveis e expressões aritméticas

Nesta seção apresentamos os conceitos de *variável* e de *expressão aritmética*. Esses são conceitos essenciais para a construção de algoritmos, sem eles seria inviável produzir um **algoritmo flexível**, que pudesse ser aplicado sobre dados distintos (e.g., o *algoritmo* da soma é flexível no sentido de podermos somar diferentes valores, não apenas 2 valores fixados).

Neste texto existem algumas "explicações adicionais", não são essenciais para o entendimento de variáveis, mas aprofundam algumas das explicações. Elas estão indicadas em quadros como esse. Portanto, se achar a explicação de determinado quadro difícil, pode pulá-la.

1. O que é uma variável?

Do ponto de vista prático, uma variável é um *nome* utilizado para armazenar "valores". Geralmente as *linguagens de programação* exigem que a *sintaxe* para esses nomes seja: deve começar com uma letra, sendo composto por letras, dígitos e "barra baixa" (_); não pode coincidir com o nome de algum comando da *linguagem*. Por exemplo, `s`, `soma` e `soma_1` são nomes válidos.

Do ponto de vista computacional, uma variável está sempre associada com uma posição de *memória*, ocupando uma quantidade de *bytes* correspondente ao *tipo de dados* que armazena. Por exemplo, na figura 2 uma variável para armazenar *inteiro* ocupa 2 *bytes*, enquanto que uma variável para armazenar "reais" precisa de 4 *bytes*. Esses tamanhos dependem do computador utilizado.

Usualmente, em qualquer linguagem de programação, sempre que usamos em um código uma variável, significa que deve-se pegar o valor armazenado nesta variável. Assim, um trecho com `10*b`, implica em pegar o valor corrente armazenado na variável `b` e multiplicá-lo por `10`.

Mas como o nome sugere, deve ser possível **alterar o valor armazenado em uma variável** e isso é feito por meio uma instrução de *atribuição*. Pode-se **atribuir** um valor à uma variável, como por exemplo, no comando `i = 0`, no qual o *expressão do lado direito* do símbolo = é calculado (valor nulo no caso) e seu resultado é atribuido à uma variável de nome *i*.

Um comando de atribuição pode ser divido em 3 partes:

1. o lado esquerdo da atribuição é composto pela *variável*;
2. o símbolo (usualmente) para igualdade =, na figura 1 indicado por uma seta); e
3. o lado direito da atribuição que é composto por qualquer *expressão aritmética* (podendo inclusive ser o resultado da chamada de uma função).

A **leitura** de uma *atribuição* deve ser: **lado esquerdo recebe valor resultante do lado direito**. Abaixo ilustramos dois outros comandos de atribuição:

- i = k significa que a variável de nome *i* (a partir desse ponto do código) passará a armazenar o valor que estava na variável *k*;
- fat = i*fat é exemplo mais "sofisticado", significa que a variável de nome *fat* deve receber o valor que anteriormente (à este comando) era guardado em *fat* multiplicado pelo valor guardado na variável *i*.

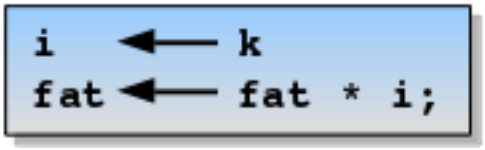

Fig. 1. Variáveis i e fat recebendo valores (constante e expressão).

Portanto, o símbolo "=" **não** tem o sentido usual da Matemática, em boa parte das linguagens de programação "`var = EXPRESSAO`" significa que *deve-se computar o valor da expressão no **lado direito da atribuição** e depois atribuir este valor à variável (no caso) de nome var (no **lado esquerdo da atribuição**)*. Isso fica mais claro na segunda *atribuição* na figura 1; "`fat = i*fat`" significa que deve-se *primeiro* computar *i*fat*, depois *atribuir* seu resultado para a *variável* "`fat`".

As linguagens que usam **=** para atribuição, usam **==** (dois símbolos de identidade) para *comparar valores*, como em `if (a==b) k = a;` em *C* e `if (a==b) : k = a` em *Python*. Uma exceção é a linguagem Pascal que apresentou uma solução mais elegante: usa **:=** para atribuição e = para comparações.

2. Sabendo um pouco mais sobre o que é uma variável

Desse modo, o conceito de **variável** está presente em todas as linguagens de programação, **não** pode coincidir com o nome de algum comando da linguagem (palavra reservada) e o seu correspondente em *código de máquina* (*linguagem baixo nível*) é uma posição *memória*, ou seja, cada *variável* estará associada à uma posição específica de *memória* (*RAM* do computador).

Existem vários **tipos de variáveis**, como variável *inteira* ou *flutuante*. O que as difere é o número de *bits* necessário para representar e como esses *bits* são interpretados. Para ilustrar como é esse processo em *baixo nível*, vamos considerar um modelo muito simplificado: um computador que consiga representar valores *inteiros positivos* (*natural*) com apenas 4 *bits*. Nesse computador, existiriam apenas 16 valores distintos (`0000`, `0001`, `0010`, `0011` e assim por diante até `1110` e `1111`).
Desse modo, usando **análise combinatória**, concluimos que nesse modelo de computador simplificado, com 4 *bits*, seria possível representar apenas $2^4=16$ valores distintos.

Em um computador atual, o número de *bits* para tipos numéricos *naturais* (ou *inteiros positivos*), usam ao menos 16 *bits*. Portanto, usando novamente *análise combinatória*, concluímos que atualmente os *naturais* podem variar de *0* até *65525*, pois $2^{16}=65526$.
Desse modo, usando um tal computador, ao tentar imprimir *65525* obteríamos um número "estranho".

Nota para usuário *Python*. Como a *linguagem Python* implementa um *algoritmo* para tratar inteiros com qualquer quantidade de dígitos (precisão arbitrária), essa discussão de quantidade de *bits* usados pode não parecer relevantes, mas é. Pois o espaço em memória depende para armazenar um inteiro depende dessa conversão, além disso o tempo de execução é proporcional ao número de dígitos.

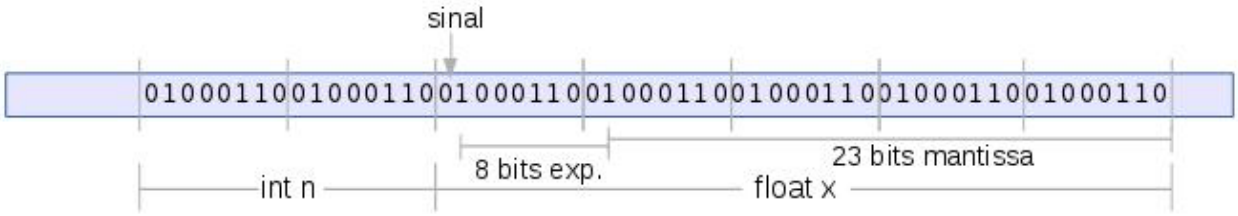

Fig. 2. Representação da memória com agrupamentos em bytes (8 bits).

Na figura 2 está ilustrado a associação de 16 *bits* à uma variável inteira de nome *n*, seguido de 32 *bits* correspondentes à uma variável *float*, usando o padrão IEEE 754 (o primeiro *bit* é o sinal *s*; os 8 *bits* seguintes correspondente ao expoente *e* e os últimos 23 à mantissa *m* - valor de *x* é $s * m * 10^e$, sendo *m* entre 0 e 1). Por exemplo, se *x=-102.003*, então *s=-1*, *m=102003* e *e=-3*, pois $-102.003 = -1 * 102003 * 10^{-3}$.

3. Para que serve variáveis

Ilustraremos a ideia e necessidade de variáveis a partir de um exemplo simples: como realizar o cálculo de gastos usando uma calculadora muito elementar.

Imagine uma calculadora simples, como ilustrado ao lado, dispondo de apenas um mostrador e apenas 4 operações (somar, subtrair, multiplicar e dividir). Se desejamos computar o gasto mensal com a padaria, devemos digitar sequencialmente os valores gastos, um a um, seguido do operador de soma `+`, sendo que ao final teclamos `=`. Vamos supor que os valores dos gastos foram `5`, `2`, `3`, `5` e `3`. Mas como foi possível obter a soma de todos os valores? Isso só foi possível por existir um "acumulador" (`AC`) para armazenar o primeiro valor digitado e depois disso, esse mesmo "acumulador" teve um novo valor a ele adicionado (ou acumulado): `AC := 5`, depois `AC := 5+2 = 7`. depois `AC := 7+3 = 10`. depois `AC := 10+5 = 15` e por último `AC := 15+3 = 18`.

Podemos pensar que calculadora está fazendo o papel da Unidade Lógico Aritmética (*Arithmetic logic unit (ALU)* em Inglês) de um computador. O recurso da calculadora que armazena os valores computados (que denotamos por `AC`) seria o equivalente a um *registrador* ou a uma posição de memória em um computador. Para se ter uma noção da diferença, deve-se notar que um computador moderno dispõe de vários registradores e o que tem sido

denominado *memória*, ordens de grandeza maiores que os registradores, hoje compra-se computadores com ao menos *4 Gb* de memória *RAM* (*Gb* = *giga bytes* = 1 bilhão de *bytes*)

Deve-se destacar que o valor armazenado em um registrador ou na memória pode ser alterado, podendo-se dizer que esses valores *variam*, vindo dai seu nome de **variável**.
Para **nomear** as variáveis, pode-se usar qualquer combinação de letras e números (iniciando por uma letra). Por exemplo, uma instrução alterando o valor da variável de nome `discriminante` em uma linguagem de programação qualquer:

```
discriminante = b*b - 4 * a * c
```

que significa que a variável de nome `discriminante` recebe o resultado da expressão "o quadrado do valor armazenado em `b` menos `4` vezes o conteúdo em `a` vezes o conteúdo em `c`".
Voltando aos *tipos de variáveis*, existe a necessidade de **declarar** qual é o tipo da variável, em uma *linguagem de programação* como *C* isso deve ser feito de modo explicito, por exemplo, podemos declarar e iniciar uma variável inteira e uma *flutuante* com o valor *5* da seguinte forma:

```
int AC = 5; float ac = 5; // em C, "AC" NAO e' a mesma coisa que "ac"!.
```

Por outro lado, a linguagem *Python* não dispõe de um mecanismo para indicar o tipo de variável explicitamente, a primeira atribuição usando um nome funciona como declaração:

```
AC = 5; ac = 5.0; # em Python, "AC" NAO e' a mesma coisa que "ac"!.
```

Portanto, **variável** é essencial para que possamos fazer com que o programa receba algum dado de nosso interesse. Mesmo em um *editor de texto* isso ocorre, o texto que digitamos é armazenado (de alguma forma, em variáveis!).

Se você estiver interessado em saber um pouco mais sobre o que é o *conceito de variável*, estude o restante desse texto.

3.1. Memória, *bits* e *bytes*

O conceito de *variável* foi introduzido a partir do surgimento as *linguagens de programação*, antes disso era necessário fazer um acesso direto às posições de memória. Portanto, variável está associada às linguagens de programação (em um *compilador* ou via um *interpretador*), tendo sido definido para facilitar o armazenamento e recuperação de valores em determinada posição de memória (para facilitar a *programação*).

Em última análise, os dados presentes na memória do computador são uma sequência de *bits* e um *bit* corresponde a uma posição de memória que pode armazenar o valor *0* ou o valor *1*. No início da computação moderna a menor quantidade de *bits* que podia ser acessada eram 8 *bits*, que foi denominado por *byte*.

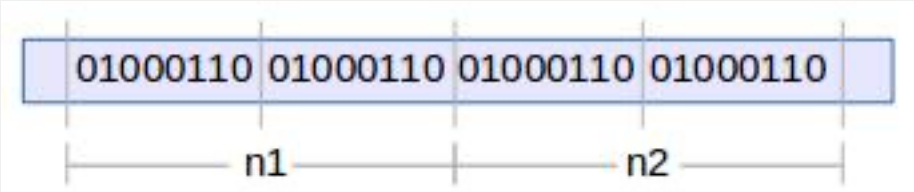

Fig. 3. Representação da memória com agrupamentos em bytes (8 bits).

Desse modo, toda informação armazenada no computador é uma sequência de *bytes* e o tratamento de cada que se dá para cada uma dessas sequências pode variar, por exmeplo, pode considerar a sequência `0010` com uma letra ou como um inteiro. Ou seja, de acordo com o **contexto**, se ela estiver associada a uma variável do tipo inteiro pode-se interpretar a sequência como um número inteiro, se for uma variável do tipo caractere, como um caractere.

3.2. Variáveis inteiras e reais

Mas qual a relação entre *bytes* e variáveis?
Primeiro, deve-se saber a priori quanto "espaço" cada tipo de variável ocupará na memória. Esse "tamanho" também determinará quantos "valores distintos" tal tipo de variável poderá armazenar. Por exemplo, suponha que determinado computador (muito, muito antigo) utilize apenas um *byte* para armazenar inteiros, como em um *byte* existem oito (8) *bits*, então nesse computador poderia existir 2^8 (ou seja, 256) inteiros distintos. Entretanto, como existem negativos e positivos, seria necessário usar (aproximadamente) metade das possibilidades para negativos e a outra metade para positivos, mas como 2^8 é um número par, um dos intervalos ficaria com $2^7=128$ valores e o outro com $2^7-1=127$.
Desse modo, nesse computador de inteiros com 8 *bits*, uma variável inteira na verdade guardaria o endereço inicial (a posição de seu primeiro *bit*) e onde houver referência a ela, o computador estaria programado para pegar os 8 *bits* a partir dessa posição e interpretá-los como um valor inteiro.
Exemplos: a sequência de *bits* `00000101` seria interpretado como *5* (pois $1*2^2+1*2^0=5$) e a sequência de *bits* `00010101` seria interpretado como *5* (pois $1*2^4+1*2^2+1*2^0=21$).

Existe uma **regra sintática** para a construção de nomes de variáveis, por exemplo, em linguagens de programação como *C* ou *Python*, uma variável é uma sequência de letras, dígitos e do caractere "barra baixa" ('_'), devendo começar com letra, mas **não** pode coincidir com nome de algum comando da linguaguem (que são denominados **palavra reservada**).

Existem outros *tipos de variáveis* como os valores "reais", que devido à técnica de implementação (vide figura 2) é denominado *flutuante* (*float*).

Do ponto de vista prático, vejamos como se usa variáveis do tipo *int* e do tipo *float* nas linguagens *C* e *Python*.

	C	*Python*
1	int n1,n2;	# desnecessário declarar em *Python*

2	n1 = 1;	n1 = 1 # primeira atribuicao equivale `a declarar em *Python*
3	scanf("%d", &n2);	n2 = int(input())
4	printf("n1=%d e n2=%d\n", n1, n2);	print("n1=%d e n2=%d" % (n1, n2)); # usando formatadores
5		print "n1=", n1, " e n2=", n2; # Python 2
6		print("n1=", n1, " e n2=", n2); # Python 3

Note as diferenças entre *C* e *Python*:

- De modo geral, todo comando em *C* precisa de ';' como finalizador.
- Na linha 1 percebe-se que em *C* é obrigatório **declarar** as variáveis, enquanto que em *Python* não.
- Na linha 3 nota-se que *C* utiliza a função pré-definida de nome *scanf* para pegar valores digitados pelo usuário enquanto *Python* usa o *input*. O *scanf* usa o formatador especial '%d' para forçar o computador a interpretar os *bytes* como um inteiro, enquanto o *input* pega os *bytes* digitados e o submete à outra função pré-definida *Python*, o *int(...)*, que converte os *bytes* lidos para um inteiro.
- Na linha 4 nota-se o mesmo tipo de diferença, *C* utiliza a função *printf* para imprimir também com o formatador para inteiro '%d', além de separar em 2 blocos, o primeiro para formatar a saída, que é cercado por aspas dupla e o segunda, uma lista de variáveis compatíveis com o formatador. Já em *Python*, usa-se os caracteres entre aspas e as variáveis misturados, separador por vírgula.
- Vale destacar que a linha 4 apresenta a sintaxe do *Python* antes da versão 3, enquanto que a linha 5 apresenta o mesmo resultado mas para o *Python* a partir da sua versão 3.

3.3. Expressões aritméticas

Do mesmo modo que em matemática é essencial efetuarmos operações aritméticas com valores numéricos, o mesmo ocorre com o computador, na verdade efetuar contas de modo rápido e "sem erro" (na verdade existem erros numéricos, mas este é outro assunto) foi a grande motivação para se construir os computadores.

Neste, os agrupamentos de valores, variáveis e operadores aritméticos recebem o nome de *expressão aritmética*. De modo geral, podemos conceituar uma *expressão aritmética* **EA** como:

1. EA := K: uma constante numérica é uma *expressão aritmética*
2. EA := EA + EA | EA - EA | EA * EA | EA / EA: uma *expressão aritmética* seguida de um **operador binário** (com 2 itens) e seguida por outra *expressão aritmética* é uma *expressão aritmética*

Os **operadores artiméticos binários**, tanto em *C* quanto em *Python* são:

Operação	Operador	Exemplo
soma	+	2 + 4

subtração	-	n1 + 1
multiplicação	*	3 * n2
divisão	/	n1 / n2

Note que foi usado espaço em branco entre os operandos e os operadores, mas isso não é obrigatório.

3.4. O resultado de uma expressão aritmética depende do contexto

É importante observar que dependendo do contexto o resultado de uma expressão é um ou outro, quer dizer, se os valores envolvidos forem todos eles inteiros, o resultado será inteiro, entretanto havendo um valor real, a resposta final será real.

A importância disso fica clara ao examinar dois exemplos simples: *3 / 2 * 2* e *3.0 / 2 * 2*. Em várias linguagens de programação a primeira expressão resulta o valor *2*, enquanto a segunda *3.0*. A razão é que no primeiro caso todos valores são inteiros, então o cômputo é realizado com aritmética de precisão inteira, ou seja, ao realizar o primeiro cômputo *3/2*, o resultado é *1* (e não *1.5* como no segundo caso), daí o segundo operador é feito com os valores *1 * 2* resultando o valor *2*.

Por exemplo, na linguagem *C*, a sintaxe e o resultado para as expressões acima está indicada no código abaixo:

```
#include <stdio.h>
void main (void) {
  //                                                Resultados
  printf("3 / 2 * 2 = %d\n",    3 / 2 * 2);     //  3 / 2 * 2 = 2
  printf("3.0 / 2 * 2 = %f\n",  3.0 / 2 * 2);   //  3.0 / 2 * 2 = 3.000000
  }
```

Entretanto existe exceção, na versão 3 do *Python*, ao usar o operador de divisão `/`, ele automaticamente converte o resultado para número real. O bloco de comandos abaixo ilustra a diferença entre o *Python 2* e o *Python 3*:

```
# Resultados em :                        Python 2                | # Python
3
print("3 / 2 * 2=", 3 / 2 * 2);       # ('3 / 2 * 2=', 2)       | 3 / 2 *
2= 3.0
print("3.0 / 2 * 2=", 3.0 / 2 * 2);   # ('3.0 / 2 * 2=', 3.0)   | 3.0 / 2
* 2= 3.0
print("3 // 2 * 2=", 3 // 2 * 2);     # ('3 // 2 * 2=', 2)      | 3 // 2 *
2= 2
print("3.0 // 2 * 2=", 3.0 // 2 * 2); # ('3.0 // 2 * 2=', 2.0)  | 3.0 // 2
* 2= 2.0
```

Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações