Linguagem Diy (Do it Yourself)

Manual de Referência

(1 de Março de 2019)

1- Introdução

Este manual de referência descreve a linguagem de programação **diy**. Embora procurando ser o mais preciso possível, utiliza-se português para descrever a linguagem. Desta forma o documento torna-se mais intuitivo, mas menos rigoroso, que uma descrição formal.

A linguagem **diy** é uma linguagem imperativa, não estruturada, e fracamente tipificada pois embora possuindo tipos estes podem, em certos casos, ser misturados.

1.1 - Tipos de dados:

A linguagem define 3 tipos de dados:

1.1.1 - números inteiros:

designados por integer, representam números inteiros positivos, negativos ou nulos, ocupando 4 bytes em complemento para dois, alinhados a 32 bits.

1.1.2 - cadeias de carateres:

designadas por string, representam sequências de carateres UTF-8, terminadas pelo caráter com o valor 0 ASCII (NULL).

1.1.3 - números reais:

designados por number, representam números reais em vírgula flutuante com 64 bits e devendo ser alinhado a 32 bits.

1.2 - Verificação de tipos:

Na linguagem **diy** as operações dependem dos tipos de dados a que são aplicadas. Os tipos suportados por cada operador e a operação a realizar são indicados na definição das expressões (ver secção 6).

1.3 - Manipulação de nomes:

Os nomes são usados para designar as entidades de um programa em **diy**, ou seja, constantes, variáveis e funções.

1.3.1 - espaços de nomes:

o espaço de nomes global é único, pelo que um nome utilizado para designar uma variável não pode ser utilizado para designar constantes ou funções.

1.3.2 - alcance das variáveis:

as variáveis globais (declaradas fora de qualquer função ou corpo), e as restantes entidades, existem do início ao fim da execução do programa. As variáveis locais aos corpos existem apenas durante a execução destes, e os argumentos formais estão válidos enquanto a função está ativa.

1.3.3 - visibilidade dos identificadores:

os identificadores estão visíveis desde a sua declaração até ao fim do seu alcance, ficheiro (globais) ou bloco (locais). A redeclaração de um mesmo identificador num corpo mais interior cria uma nova variável que encobre a anterior até ao fim do respetivo corpo. Uma função não pode declarar no seu corpo principal identificadores com mesmo nome dos seus argumentos formais.

2- Convenções lexicais

A linguagem de programação **diy** é constituída por seis grupos de elementos léxicais (tokens), devendo o elemento léxical ser constituído pela maior sequência de carateres que constitua um elemento lexical válido:

2.1 - Carateres brancos:

são considerados carateres brancos aqueles que, embora servindo para separar os elementos lexicais, não representam nenhum elemento lexical. São considerados carateres brancos: **espaço** ASCII SP (0x20), **mudança de linha** ASCII LF (0x0A), **recuo do carreto** ASCII CR (0x0D) e **tabulação horizontal** ASCII HT (0x9).

2.2 - Comentários:

Os comentários funcionam como separadores de elementos lexicais. Os comentários são iniciados por ('=<') e terminam com ('=>'), desde que não façam parte de cadeias de carateres. Estes comentários podem ser aninhados (entalhados ou nested), isto é, podem ser colocados uns dentro dos outros. Os comentário podem ainda ser iniciados por ('=='), caso em que terminam implicitamente no fim da respetiva linha.

2.3 - Identificadores:

por vezes designados por nomes , são constituídos por uma letra (maiúscula ou minúscula) seguida por 0 (zero) ou mais letras, dígitos ou o caráter ('_'). O número de carateres que constituem um identificador é ilimitado e dois nomes designam identificadores distintos se houver alteração de maiúscula para minúscula, ou vice-versa, de pelo menos um caráter.

2.4 - Palavras chave:

os identificadores indicados na tabela abaixo estão reservados pela linguagem **diy**, são palavras chave da linguagem, e não podem ser utilizados como identificadores normais. Estes identificadores têm de ser escritos exatamente como na tabela abaixo:

void integer string public number const if then else while do for in step upto downto break continue

2.5 - Literais:

são notações para valores constantes de alguns tipos predefinidos da linguagem. (Notar que constantes são identificadores que designam elementos cujo valor não pode sofrer alterações durante a execução do programa.)

2.5.1 - inteiros:

um literal inteiro, ao contrário de uma constante inteira, é um número positivo. (Notar que os números negativos são construídos pela aplicação do operador menos unário (-) a um literal positivo.)

Um literal inteiro em decimal é constituído por uma sequência de 1 (um) ou mais dígitos decimais (dígitos de '0' a '9') em que o primeiro dígito não é um '0' (zero), exceto no caso do número 0 (zero) composto apenas pelo dígito '0' (zero), que é igual qualquer que seja a base de numeração. Um literal inteiro em octal é começa sempre pelo dígito '0' (zero), sendo seguido de um ou mais dígitos de '0' a '9'. Note-se que nesta linguagem o número 09 é um inteiro octal válido com valor igual a 011 (octal). Um literal inteiro em binário é começa sempre pela sequência '0b', sendo seguido de um ou mais dígitos '0' ou '1'.

Se não for possível representar o literal inteiro na máquina, devido a um *overflow*, deverá ser gerado um erro lexical.

2.5.2 - cadeia de carateres:

começam e terminam com o caráter aspa ("). Uma cadeia de texto pode conter qualquer número de carateres, podendo estes ser valores ASCII (exceto o 0 ou NULL) ou carateres ISO-LATIN-15 para carateres portugueses. Os carateres utilizados para iniciar ou terminar comentários como ('<'), ('>') ou ('=') têm o seu valor normal ASCII não iniciando ou terminando qualquer comentário. O caráter aspa (") pode ser utilizado desde que precedido de (\). Os carateres ASCII LF, CR e HT podem ser representados pelas sequências '\n', '\r' e '\t', respetivamente. Qualquer outro caráter pode ser representado por 1 ou 2 dígitos hexadecimais precedidos do caráter '\', por exemplo '\0a' ou apenas '\A' se o caráter seguinte não representar um digito hexadecimal.

2.5.3 - reais em vírgula flutuante:

O literal real é um número real com partes inteira e decimal obrigatórias separadas por um ponto decimal, além de uma parte exponencial facultativa, tal como definido pela linguagem **Pascal**.

Se não for possível representar o literal na máquina, devido a um *overflow*, deverá ser gerado um erro lexical.

2.6 - Operadores de expressões:

são considerados operadores da linguagem **diy** os seguintes elementos lexicais,

\- + * / % < > = >= <= <> := ++ -- ! | & ~

2.7 - Delimitadores e separadores:

Os elementos lexicais seguintes são considerados delimitadores da linguagem **diy**: '#', '{', '}', '(', ')', '[', ']', ';', e ',' (vírgula). O caráter '\n' ou **mudança de linha** ASCII LF (0x0A) funciona como o delimitador ';' quando a linha termina com um literal, um identificador ou os carateres ')' ou '!', se a linha não terminar já com o caráter ';'.

3 - Gramática

3.1 - Gramática:

A gramática da linguagem **diy** pode ser resumida pelas regras abaixo. Os elementos a negrito são literais, os parênteses curvos agrupam elementos, elementos alternativos são separados por uma barra vertical, elemento opcionais estão entre parênteses retos e os elementos que se repetem zero ou mais vezes estão entre chavetas. Cuidado que a barra vertical, os parênteses e as chavetas são elementos lexicais da linguagem quando representados a negrito.

ficheiro ={ declaração }
declaração =[**public**] [**const**] tipo [*****] ident [init] **;**
tipo =**integer** | **string** | **number** | **void**
init =**:=** inteiro |
:= [**const**] cadeia |
:= real |
:= ident |
([parâmetros] **)** [corpo]
parâmetros=parâmetro { **,** parâmetro }
parâmetro =tipo [*****] ident
corpo =**{** { parâmetro **;** } { instrução } **}**
instrução =**if** expressão **then** instrução [**else** instrução] |
do instrução **while** expressão **;** |
for left-value **in** expressão (**upto** | **downto**) expressão [**step** expressão] **do** instrução |
expressão **;** |
corpo |
break [inteiro] **;** |
continue [inteiro] **;** |
left-value # expressão **;**

3.1.1 - elementos lexicais:

foram omitidos da gramática, por já terem sido definidos acima, os seguintes elementos:

3.1.1.1 - ident:
definido em 2.3

3.1.1.2 - inteiro:
definido em 2.5.1

3.1.1.3 - cadeia:
definido em 2.5.2

3.1.1.4 - real:
definido em 2.5.3

3.1.2 - expressão:
foi deliberadamente omitida da gramática e será tratada em na secção 6.

3.2 - Constantes:
A linguagem define 2 tipos de constantes:

3.2.1 - identificadores constantes:
sucedem a declaração pela palavra reservada const, impedindo que o identificador declarado possa ser utilizado em operações que modifiquem o seu valor. Caso um identificador designe uma constante inteira que não seja *public* (ver adiante) o seu valor poderá ser diretamente substituído no código, não ocupando espaço. Os identificadores declarados constantes que não sejam *public* têm de ser iniciados.

3.2.2 - valores constantes:
são todos os corpos das funções (devido à gestão das *caches* pelos processadores, que não gostam de *self-modifying code*) e os inicializados de strings precedidas pela palavra reservada const. Como as funções são sempre constantes a palavra reservada const não deve ser utilizada.

3.3 - Ficheiros:
Um ficheiro descrito em **diy** é constituído por uma sequência de declarações.

3.3.1 - programas:
a execução de um programa exige uma só função pública entry. Assim, nos diversos ficheiros que podem constituir um programa em**diy** apenas um deles terá de incluir uma função com a assinatura: public integer entry (integer argc, string *argv)

3.4 - Símbolos globais:
Para a utilização de compilação separada em **diy** existe a palavra reservada public que torna o símbolo visível de/para outros ficheiros.

3.4.1 - importação:
um símbolo declarado public mas não iniciado é considerado exterior e tratado como pertencente a outro módulo.

3.4.2 - exportação:
um símbolo declarado public e iniciado é considerado global e tratado como podendo ser acedido de outros módulos.

3.4.3 - declaração por avanço (*forward declaration*):
um símbolo que não é declarado public e não é iniciado é considerado como declarado por avanço devendo ficar completamente definido posteriormente no mesmo ficheiro.

3.5 - Declaração de variáveis e constantes:
cada declaração permite declarar uma única variável ou constante. Uma declaração inclui os seguintes componentes:

3.5.1 - constante:
designada pela palavra reservada const, que torna o identificador constante, ou seja, cujo valor representado não pode ser modificado. Tal não significa que os valores indiretamente referidos (apontados) pelo identificador sejam constantes.

3.5.2 - tipo de dados:
designado por um dos 3 tipos de dados (integer, string ou number).

3.5.3 - identificador:
designada por um identificador que passa a nomear a entidade declarada.

3.5.4 - ponteiro:
designado por * permite referir posições de memória contendo dados to tipo base.

3.5.5 - inicialização:
a existir inicia-se com o operador de atribuição ':=' seguido de valores constantes dependentes do tipo declarado:

3.5.5.1 - inteiros:
o valor a iniciar é um número inteiro, representado em decimal, octal ou binário.

3.5.5.2 - cadeia de carateres:
a cadeia é constante se for precedida da palavra reservada const e variável em caso contrário, independentemente do identificador que a refere ser constante ou não. Notar que o identificador de uma cadeia de carateres é sempre um ponteiro e nunca um vetor, pelo pode ser constante ou não.

3.5.5.3 - reais:
o valor a iniciar é um número real representado em vírgula flutuante tal como definido em 2.5.3.

3.5.5.4 - identificadores:
caso o identificador seja um ponteiro, o seu valor pode ser iniciado com um outro identificador do mesmo tipo base.

3.5.5.5 - funções:
ver secção 4.
Notar que declarações de constantes não iniciadas só é possível se forem identificadores importados.

4 - Funções

Uma função permite agrupar um conjunto de instruções num corpo, que é executado com base num conjunto de parâmetros (os argumentos formais) quando é invocada a partir de uma expressão.

4.1 - Declaração :

As funções em **diy** são sempre designadas por identificadores constantes precedidos do tipo de dados devolvido pela função e de uma lista de argumentos formais delimitados por parêntesis. As funções, que recebam argumentos, devem indicá-los no cabeçalho. Rotinas que não devolvem qualquer valor são declaradas do tipo void.
Uma declaração de uma função não iniciada é utilizada para tipificar um identificador importado (quando público) ou para efetuar*forward declarations* (utilizada para pré-declarar uma função que seja usada antes de ser definida, por exemplo, entre duas funções mutuamente recursivas). Uma declaração de uma função iniciada define uma nova função constituída pelo corpo que a inicia.
O valor devolvido por uma função é mantido numa variável com o mesmo nome da função e que é implicitamente declarado, caso a função não seja do tipo void.

O corpo da função, designado por bloco principal, não pode definir variáveis designadas por identificadores com o mesmo nome de nenhum dos argumentos formais da função. Nem os argumentos formais nem nenhum dos blocos da função pode declarar variáveis com o nome da própria função.

4.2 - Invocação :

A função só pode ser invocada através de um identificador que refira uma função previamente declarada ou definida.
Caso existam argumentos, na invocação da função o seu identificador é seguido de uma lista de expressões delimitadas por parênteses curvos. A lista de expressões é uma sequência de expressões separadas por vírgulas. As expressões são avaliadas da direita para a esquerda antes da invocação da função e o valor resultante passado por cópia (passagem de argumentos por valor).

O tipo e número de parâmetros atuais deve ser igual ao tipo e número de parâmetros formais. A ordem dos parâmetros atuais deverá ser a mesma dos argumentos formais da função a ser invocada. Os parâmetros atuais devem ser colocados na pilha de dados pela ordem inversa da sua declaração (o primeiro no topo da pilha) e o endereço de retorno no topo da pilha.
Quem coloca os argumentos na pilha (a função chamadora) também é responsável pela sua remoção após o retorno da função chamada (chamada *à lá* C).

4.3 - Corpo :

O corpo, quer seja o bloco principal de uma função ou um sub-bloco de uma instrução condicional ou de iteração, pode definir apenas variáveis não iniciadas.

5 - Instruções

Excepto quando indicado as instruções são executadas em sequência, sendo os seus efeitos traduzidos, em geral, pela alteração do valor de variáveis.

5.1 - Instrução condicional:

Se a expressão que segue a palavra reservada **if** for diferente de 0 (zero) então a instrução que segue a palavra reservada **then** é executada.
Caso exista um conjunto **else**, a instrução é executada quando a expressão que segue a palavra reservada **if** tem o valor 0 (zero).

5.2 - Instrução de iteração por valor:

A instrução que segue a palavra reservada **do**, sendo a sua execução repetida enquanto a expressão que segue a palavra reservada**while** for diferente de 0 (zero).

5.3 - Instrução de iteração por contagem:

iniciada pela palavra reservada for, deve começar por atribuir ao left-value o resultado da expressão que segue palavra reservada in. A instrução do ciclo é sucessivamente repetida até o valor do left-value ser superior (upto) ou inferior (downto) ao valor da expressão que segue as palavras reservadas upto e downto. Se existir a expressão que segue a palavra reservada step, o left-value deverá ser incrementado (upto) ou decrementado (downto) do seu valor, caso contrário o valor deve ser considerado unitário.

5.4 - Expressão como instrução:

qualquer expressão pode ser utilizada como instrução, mesmo que não produza qualquer efeito secundário.

5.5 - Corpo como instrução:

um corpo pode substituir uma instrução, permitindo a execução sequencial de mais de uma instrução em seu lugar.

5.6 - Instrução de continuação:

iniciada pela palavra reservada continue, a existir deverá ser a última instrução do bloco em que se insere. Esta instrução reinicia uma instrução de iteração, ignorando tantos ciclos quantos o valor constante inteiro positivo que se lhe segue subtraído de uma unidade. Deve ser considerado o valor unitário, caso o valor inteiro seja omitido.
Assim, continue 3 ignora o ciclo mais interior e o seguinte, reiniciando a execução na avaliação da expressão do terceiro ciclo mais interior, enquanto continue 1 reinicia o próprio ciclo tal como a instrução continue. Esta instrução só pode existir dentro de um ciclo.

5.7 - Instrução de terminação:

iniciada pela palavra reservada break, a existir deverá ser a última instrução do bloco em que se insere. Esta instrução termina a execução de tantos ciclos quantos o valor constante inteiro positivo que se lhe segue. Deve ser considerado o valor unitário, caso o valor inteiro seja omitido.
Esta instrução só pode existir dentro de um ciclo, devendo o número de ciclo aninhados ser igual ou inferior ao valor constante inteiro positivo que se lhe segue.
Quando esta instrução é executada, os conjuntos **else** associados aos ciclos terminados não são executados.

5.8 - Reserva de memória na pilha

A instrução de reserva de memória na pilha permite que numa função se reserve uma quantidade de memória variável.

1 6 - Expressões

Uma expressão é uma representação algébrica de uma quantidade. Assim, todas as expressões devolvem um valor. As expressões na linguagem de programação **diy** utilizam operadores algébricos comuns: soma, subtração, multiplicação e divisão inteira e resto da divisão, além de outros operadores.

As expressões são sempre avaliadas da esquerda para a direita, independentemente da associatividade do operador. A precedência dos operadores é a mesma para operadores na mesma secção, sendo as secções seguintes de menor prioridade que as anteriores. O valor resultante da aplicação da expressão bem como a sua associatividade são descriminados em cada operador.

A tabela seguinte que resume os operadores da linguagem **diy**, por grupos de precedência decrescente:

designação	**operadores**	**associatividade**
primária	( ) [ ]	não associativos
unária	* & ! - ++ --	não associativos
multiplicativa	* / %	da esquerda para a direita
aditiva	+ -	da esquerda para a direita
comparativa	< > <= >=	da esquerda para a direita
igualdade	= <>	da esquerda para a direita
`não' lógico	~	não associativo
`e' lógico	&	da esquerda para a direita
`ou' lógico	|	da esquerda para a direita
atribuição	:=	da direita para a esquerda

Os operadores têm o mesmo significado que em **C**, com a exceção do operador de comparação (que usa **<>** em vez de **!=**) e do operador fatorial (que usa **!**), como noutras linguagens de programação.

6.1 - Expressões primárias:

6.1.1 - identificadores:

Um identificador é uma expressão desde que tenha sido devidamente declarado. Um identificador pode denotar uma variável ou uma constante.
Um identificador é o caso mais simples de um *left-value*, ou seja, uma entidade que pode ser utilizada no lado esquerdo (*left- value*) de uma atribuição.

6.1.2 - literais:

Os literais podem ser valores constantes inteiros ou reais não negativos, tal como definidos nas convenções lexicais, ou cadeias de carateres.

6.1.3 - parênteses curvos:

Uma expressão entre parênteses curvos tem o valor da expressão sem os parênteses e permite alterar a prioridade dos operadores. Uma expressão entre parênteses não pode ser utilizada como *left-value* (ver em indexação).

6.1.4 - indexação:

Uma expressão indexação referencia uma entidade através da sua localização em memória. O identificador deverá designar apenas uma variável ou constante do tipo ponteiro.

O resultado de uma expressão de indexação é o valor existente na posição de memória indicada pelo identificador somado com o deslocamento, em quantidades da dimensão do tipo base, do valor da expressão. A indexação de cadeias de carateres é o valor inteiro correspondente ao caráter. O deslocamento é calculado, ``*à la C*'', em que o primeiro elemento tem deslocamento 0 (zero), ou seja, base[0].
Uma indexação também pode ser utilizada como *left-value*.

6.1.5 - invocação:
A função só pode ser invocada através de um identificador que refira uma função previamente declarada ou definida. Variáveis ou constantes do tipo void * referem funções, podendo ser diretamente utilizadas na invocação da função para que apontam. O número e tipo de argumentos deve ser igual, não havendo lugar a conversões implícitas de tipos.

6.2 - Expressões unárias:

6.2.1 - localização:
A expressão localização devolve a posição de memória ocupada pelo *left-value*. É possível obter a localização de uma função utilizando o seu nome sem parênteses.

6.2.2 - indireção:
A expressão de indireção devolve o conteúdo da posição de memória indicada pela expressão.

6.2.3 - simétrico:
A expressão valor simétrico devolve o simétrico do valor inteiro ou real.

6.2.4 - fatorial:
A expressão fatorial devolve o valor real do fatorial de um valor inteiro positivo, ou o valor unitário se o argumento não for positivo.

6.2.5 - incremento e decremento:
as operações de incremento e decremento só podem ser aplicadas a *left-values* do tipo inteiro. A operação modifica o valor designado de tal forma que este passe a referir o elemento seguinte ou anterior, dependendo de se tratar de incremento ou decremento, respetivamente. O valor devolvido pela operação corresponde ao valor do *left-value* no momento da avaliação da expressão. Cada operação de pré incremento ou decremento é efetuada antes do cálculo do respetivo *left-value*. Cada operação de pós incremento ou decremento é efetuada depois do cálculo do respetivo *left-value*.

6.3 - Expressões multiplicativas:
as operações são apenas aplicáveis a valores inteiros ou reais, devolvendo o resultado da respetiva operação algébrica.

6.4 - Expressões aditivas:
as operações são apenas aplicáveis a valores inteiros ou reais, devolvendo o resultado da respetiva operação algébrica.

6.5 - Expressões de grandeza:

as operações são aplicáveis a valores inteiros ou reais, devolvendo o valor inteiro 0 (zero) caso seja falsa e 1 (um) caso contrário, a cadeias de carateres, devolvendo o resultado da sua comparação alfabética segundo o código ASCII (logo 'A' é diferente de 'a').

6.6 - Expressões de igualdade:

as operações são aplicáveis a valores inteiros ou reais e a cadeias de carateres, tal como no caso anterior.

6.7 - Expressões de negação lógica:

a operação é aplicável a valores inteiros, devolvendo o valor inteiro 0 (zero) caso o argumento seja diferente de 0 (zero) e 1 (um) caso contrário.

6.8 - Expressões de junção lógica:

a operação é aplicável a valores inteiros, devolvendo o valor inteiro 1 (um) caso ambos os argumentos sejam diferente de 0 (zero) e 1 (um) caso contrário. Caso o primeiro argumento seja 0 (zero) o segundo argumento não deve ser avaliado.

6.9 - Expressões de alternativa lógica:

a operação é aplicável a valores inteiros, devolvendo o valor inteiro 0 (zero) caso ambos os argumentos sejam iguais a 0 (zero) e 1 (um) caso contrário. Caso o primeiro argumento seja 1 (um) o segundo argumento não deve ser avaliado.

6.10 - Expressões de atribuição:

O valor da expressão do lado direito do operador é guardado na posição indicada pelo *left-value* do lado direito do operador de atribuição.

7 - Interface com o sistema operativo

7.1 - Função principal:

A execução de um programa em **diy** inicia-se com a invocação da função:

```
public integer entry (integer argc, string *argv, string *envp);
```

O primeiro argumento representa o número de argumentos indicados na linha de comando, incluindo o nome do programa. Os segundo e terceiro argumentos são sequências de cadeias de carateres que correspondem ao valor dos argumentos e das variáveis de ambiente, respetivamente. O valor de retorno desta função é devolvido ao sistema operativo que invocou o programa. Assim, o valor de retorno da função segue as regras do sistema operativo, ou seja, 0 (zero) se a execução correu sem erros, 1 (um) se os argumentos são em número ou valor inválido, 2 (dois) erro detetado durante a execução. Os valores superiores a 128 indicam que o programa terminou com uma interrupção. Para que muitos programas funcionem corretamente exige-se que todas as ferramentas, e programas desenvolvidos, devolvam 0 (zero) se a execução correu sem problemas e um valor diferente de 0 (zero) se existiu um erro.

7.2 - rotinas de biblioteca:

O ficheiro **linux.asm** contém a rotina de arranque (_start) que deve invocar a rotina de entrada (entry) para os programas desenvolvidos em **diy**, bem como a rotina de terminação (_exit). O ficheiro **lib.asm** contém um conjunto de rotinas de biblioteca que poderá utilizar, com nomes auto-explicativos e semelhantes aos da biblioteca de **C**: println, printsp, prints, printi, readln, readb, readi, strlen, atoi e itoa.

O ficheiro **dbl.c** contém rotinas de leitura, escrita e conversão de números reais representados em vírgula flutuante: readd, readr, printd, printr, atod, atof, dtoa e dtof. Deverão igualmente ser adicionadas quaisquer rotinas necessárias ao correto funcionamento dos operadores existentes na linguagem.

7.3 - chamadas ao sistema operativo:

O ficheiro **sys.asm** contém as chamadas ao sistema que pode realizar em programas escritos em **diy**. Uma explicação das chamadas ao sistema pode ser obtida através de:

```
  prompt$ man 2 intro
  prompt$ man 2 syscalls
```

Algumas destas chamadas não existem na biblioteca de **C**, outras, como o **brk**, têm um comportamento diferente da rotina homónima da biblioteca de **C**.

8 - Exemplos

Os exemplos apresentados não são exaustivos, pelo que nem todas as construções da linguagem são utilizadas.

O já habitual hello world:

```
public void prints(string s)
public integer entry (integer argc, string *argv, string *envp) {
  prints("olá pessoal!\n")
  entry := 0
};
```

O cálculo da função de Ackermann. (Esta função tem um crescimento muito elevado pelo que nos computadores atuais, mesmo utilizando **C**, os argumentos não deverão exceder m=3 e n=12 para executar em poucos segundos)

```
public void prints(string s)
public void printi(integer i)
public void println()
public integer atoi(string s)

integer cnt := 0;
integer ackermann (integer m, integer n) {
  cnt := cnt + 1
  if m = 0 then ackermann := n+1
  else if n = 0 then ackermann := ackermann(m-1, 1)
  else ackermann := ackermann(m-1, ackermann(m, n-1))
};

public integer entry (integer argc, string *argv, string *envp) {
  if argc > 2 then {
    printi(ackermann(atoi(argv[1]), atoi(argv[2])))
    prints(" #")
    printi(cnt)
    println()
  }
  entry := 0
};
```

Estes e outros exemplos podem ser obtidos na página do fénix.

(C)IST, *Pedro Reis dos Santos*, 2019-02-18