Organização de Computadores – 2005/2006
Linguagem Forth

Paulo Ferreira
paf@dei.isep.ipp.pt

Maio de 2006

História

- Charles Moore - ≪Fourth Generation Language≫
- Setembro de 1973 -≪A Program for Telescope Control≫
- Fig-Forth (Forth Interest Group)
- Forth 79
- Forth 83
- ANSI Forth

ORGC Linguagem Forth – slide 3

Características

- Pequeno Tamanho
- Interpretada ⇒ Interactiva
- Compilada ⇒ Boa Performance
- Uso em Sistemas ≪*Embedded*≫
- Incorporado em Programas

ORGC Linguagem Forth – slide 4

Confusões

- RPN - Usa uma pilha
- Compreender Forth é compreender o ≪*Source Code*≫ de um Forth
- Compreender a linguagem é compreender um compilador
- Sintaxe Livre
- Compilador Extensível
- Forth não tem X quer dizer que pode-se fazer X como se quiser

ORGC Linguagem Forth – slide 5

Em falta

- Nos forths mais pequenos não há de origem:
  - ☐ Strings
  - ☐ Floating point
  - ☐ Estruturas
  - ☐ *Cases*: pode-se fazer cases da forma que se quiser

ORGC Linguagem Forth – slide 6

Plataformas

- Dos, Windows, Mac OS, Mac OSX, Linux, Unix
- Quase todos os μControladores
- Palm, Lego Mindstorms, Java, .Net
- OpenBios, OpenFirmware

Metacompilação

- Em que linguagem escrever um Forth:
  - Forth $\Rightarrow$ Mais fácil (metacompilador)
  - Assembler $\Rightarrow$ se só existir o assembler para esse μP (mais simples)
  - Em C $\Rightarrow$ Portabilidade mais fácil

Open Source

- Forths no domínio público desde os anos 70
- Vários livres para uso não comercial `gforth` $\Rightarrow$ abrangido pela licença GNU corre em Unix e Windows
  - Escrito em C para portabilidade.

Comerciais

- Forth Inc $\Rightarrow$ `http://www.forth.com`
- MPE $\Rightarrow$ `http://www.mpeld.demon.uk`
- Diferentes das linguagens *normais*
- *Source Code* está normalmente incluído
- Licenças prevêm o caso dos sistemas *embedded*

Acidentes históricos

- Source code em Maiúsculas
  - Legibilidade reduzida
  - Hoje forths não são normalmente *case-sensitive*
  - Minúsculas são mais legíveis
- Source code em *blocos*
  - Sem sistema operativo como usar um disco?
  - 16 linhas de 64 caracteres = 1 bloco = 1 kbyte
  - Escrevem-se definições menores do que 1 bloco
  - Se forem maiores o programa está mal escrito

Gforth

- *Escrito* em C
- Compilável em máquinas Unix de várias Arquitecturas + Windows
  - `gforth`
- Como sair:
  - `bye`
- Ficheiros de source code: Extensão .fs
  - `gforth exemplos.fs`
- De dentro do gforth:
  - `s"exemplos.fs"included` Nota: espaço a seguir a `s"`

Funcionamento slide 13

RPN

- Reverse Polish Notation
- Como funciona:
  - com o uso de uma stack
- Vantagens:
  - ausência de operandos
  - ausência de prioridades
  - todos os operadores são postfix
  - simplicidade do parser

Normalmente

- Notação algébrica normal ou ”infix”onde escrevemos os operadores entre os operandos.
- Exemplos:
  - 10-2*3
  - (6+5)/(7-4)
- Neste tipo de notação necessitamos de definir prioridades entre as diferentes operações, e necessitamos de parênteses, que não sendo operadores nem operandos, são necessários para alterarmos a prioridade normal das operações.

RPN

- Prefix – notação ≪prefix≫ conhecida como ≪Polish Notation≫, uma vez que foi desenvolvida pelo matemático polaco Jan Lukasiewicz (1878–1956) onde os operadores são escritos antes dos operandos.
- Do ponto de vista informático a notação ≪postfix≫ ou RPN (reverse polish notation) é a mais interessante. Em RPN escrevemos primeiro os operandos e depois os operadores.
- Exemplos:
  - 10 2 3 * -
  - 6 5 + 7 4 - /

Regras RPN

- Em RPN não necessitamos de parênteses, e todos os operadores possuem a mesma prioridade.
- Escrever um número de cada vez, e a seguir ao número escrever o que este está a fazer aos números que já escrevemos até aí.
- Exemplo de uma conversão da notação normal para RPN:
  - 10-2*3
  - 10 2 3 * -
- Outro exemplo, converter para RPN a expressão algébrica:
  - (6+5)/(7-4)
  - 6 5 + 7 4 - /

Convenções

- Algumas convenções

Nome	Designação
.	Print
@	Fetch
!	Store
?	Question

Exemplos de uso:

```
26 34 + 5 * .
VARIABLE XIS
26 XIS !
XIS @ 3 * XIS !
XIS ?
```

- Comentários
  - Desde ≪(≫ até ≪)≫ sem *nesting*, normalmente para documentar o efeito na stack de cada uma das *rotinas* existentes
  - Ou então do *backslash* ao fim da linha

Exemplos

- Palavras mais comuns

```
+   ( n1 n2 -- n3 )
.   ( n1 -- )       \ print
*   ( n1 n2 --n3 )
@   ( addr -- n )   \ fetch
!   ( n addr -- )   \ store
u.  ( u -- )        \ u-print
```

- Novas definições

```
: star      42 emit ;   ( -- )
: triplo    3 * ;       ( n1 -- n2 )
: ?         @ . ;       ( addr -- )
```


Detalhes

- A importância dos espaços:
  - O *parser* (na realidade muito mais simples) do forth reconhece cada palavra com base no espaços entre elas, logo cada string (separada por espaços) de um ou mais caracteres é (ou pode ser) uma palavra, isto é um comando forth.
- Como se chamam as novas definições?
  - Dizem-se que são palavras, ou em inglês *words*
  - Confusão com *words* de memória!
  - Uma posição de memória recebe o nome de ≪*cell*?

Filosofia

- Implementação
  - Ir construindo novas palavras usando as que já temos, até que tenhamos o nosso problema resolvido
  - Implementação *Down-Top*
  - Mesmo que o design seja *Top-Down*
- Simplicidade
  - Se não se conseguiu simplificar o problema então não se compreendeu o problema.
- Factorizar
  - Devemos fazer definições curtas e simples
  - Quanto mais simples forem maior potencial de reutilização terão

Mais detalhes

- Uma stack de valores com inteiros de X bits
- Normalmente X=tamanho endereços processador
- Uma stack de retorno para as rotinas
- Menos comum: uma stack para floating point (floating point não é comum)
- Na prática
  - Em processadores de 8 bits, a stack de valores tem normalmente 16 bits de largura.
  - Em processadores de 32 bits tem 32 bits de largura.
  - A stack de retorno tem o mesmo tamanho da stack de dados

Números duplos

- Com o dobro de tamanho de um inteiro normal.
- A parte mais significativa no topo da stack.
- Em termos de input um número com um . (ponto) em qualquer sítio é um numero duplo.

Base numérica

Em que base trabalhamos?

- Existe uma variável chamada
  - `base ( -- addr )`
- Afecta tanto o input como o output de números
- Além de mudarmos a variável base temos:
  - `decimal ( -- )`
  - `hex ( -- )`
  - `binary ( -- )`
- Apenas em gforth temos prefixos:
  - `&10` – 10 em decimal
  - `%101` – 101 em binário – 5 em decimal
  - `$01a` – 1a em hexadecimal — 26 em decimal

Variáveis e constantes

- `variable nome_da_variável`
  - `variable myvar1`
- A partir daqui `myvar1` deixa na stack o seu endereço
- Constantes:
  - `334 constant curso`
- A partir daqui `myconst` deixa na stack o valor 334
- Também existem:
  - `2variable ( -- )`
  - `2constant ( d -- )`

Palavras úteis

- Ver o estado da stack:
  - `.s ( -- )`
- *Descompilar* uma definição:
  - `see nome ( -- )`
- Ver que palavras existem:
  - `words ( -- )`

Flags aritméticas

- `true ( -- f )`
  - deixa na stack de valores um número com todos os bits a 1
- `false ( -- f )`
  - deixa na stack de valores um número com todos os bits a 0

Aritmética simples

```
+        ( n1 n2 -- n3 )
1+       ( n1 -- n2 )
-        ( n1 n2 -- n3 )
1-       ( n1 -- n2 )
*        ( n1 n2 -- n3 )
/        ( n1 n2 -- n3 )
mod          ( n1 n2 -- n3 )
/mod         ( n1 n2 -- n3 n4 )
negate   ( n1 -- n2 )
abs          ( n -- u )
min          ( n1 n2 -- n3 )
max          ( n1 n2 -- n3 )
floored      ( -- flag )
             \ true se divisão for "floored"
```


Divisão

Problemas com divisão de inteiros:

- Divisão *floored* ou simétrica?
- Traduzindo:
  - -1/2 = 0 resto -1 ?
  - -1/2 = -1 resto 0 ?
- A primeira hipótese parece mais acertada, mas se a seguirmos, a função vai apresentar uma descontinuidade à volta do 0.

Aritmética de precisão dupla

```
s>d          ( n1 -- d1 )
d>s          ( d1 -- n1 )
d+           ( d1 d2 -- d3 )
d-           ( d1 d2 -- d3 )
dnegate      ( d1 -- d2 )
dabs         ( d1 -- ud1 )
dmin         ( d1 d2 -- d3 )
dmax         ( d1 d2 -- d3 )
```

Operações lógicas

```
and     ( n1 n2 -- n3 )
or      ( n1 n2 -- n3 )
xor     ( n1 n2 -- n3 )
invert  ( n1 -- n2 )
lshift  ( u1 n -- u2 )
rshift  ( u1 n -- u2 )
2*      ( n1 -- n2 )
d2*     ( d1 -- d2 )
2/      ( n1 -- n2 )
d2/     ( d1 -- d2 )
```


Comparação numérica

```
<       ( n1 n2 -- flag )
<=      ( n1 n2 -- flag )
<>      ( n1 n2 -- flag )
=       ( n1 n2 -- flag )
>       ( n1 n2 -- flag )
>=      ( n1 n2 -- flag )
0<      ( n -- flag)
0<=     ( n -- flag)
0<>     ( n -- flag)
0=      ( n -- flag)
0>      ( n -- flag)
0>=     ( n -- flag)
```


Comparação Numérica

```
u<      ( u1 u2 -- flag )
u<=     ( u1 u2 -- flag )
u>      ( u1 u2 -- flag )
u>=     ( u1 u2 -- flag )
within  ( n1 n2 n3 -- flag )    \   n2 <= n1 < n3
within  ( u1 u2 u3 -- flag )    \   u2 <= u1 < u3
d<      ( d1 d2 -- flag )
d<=     ( d1 d2 -- flag )
d<>     ( d1 d2 -- flag )
d=      ( d1 d2 -- flag )
d>      ( d1 d2 -- flag )
d>=     ( d1 d2 -- flag )
```

Comparação Numérica

```
d0<     ( d -- flag )
d0<=    ( d -- flag )
d0<>    ( d -- flag )
d0=     ( d -- flag )
d0>     ( d -- flag )
d0>=    ( d -- flag )
du<     ( ud1 ud2 -- flag )
du<=    ( ud1 ud2 -- flag )
du>     ( ud1 ud2 -- flag )
du>=    ( ud1 ud2 -- flag )
```


Aritmética mista

```
m+      ( d1 n -- d2 )
*/      ( n1 n2 n3 -- n4 )      \ n4 = (n1*n2)/n3 com o
                                \ valor intermédio duplo
*/mod   ( n1 n2 n3 -- n4 n5 )   \ n1*n2 = n3*n5 +n4
m*      ( n1 n2 -- d )
um*     ( u1 u2 -- ud )
m*/     ( d1 n2 u3 -- dquot )   \ dquot = (d1*n2)/u3
um/mod  ( ud u1 -- u2 u3 )      \ ud =u3*u1+u2
fm/mod  ( d1 n1 -- n2 n3 )      \ d1 = n3 * n1 + n2
                                \ "floored"
sm/rem  ( d1 n1 -- n2 n3 )      \ d1 = n3 * n1 + n2
                                \ "simetric"
```


Floating point

- A maioria dos forths não possui *floating point*.
- Em sistemas embedded não existe essa necessidade.
- Exemplo de falsos *floating point*: quantias monetárias.
- Multiplicar por um número real: usa-se */ com dois inteiros
- Aproximações (para 16 bits)
  - $\pi = 355/113$
  - $e = 25946/9545$
  - $\sqrt{2} = 27720/19601$
  - $\sqrt{3} = 32592/18817$
  - $\sqrt{10} = 273379/8658$

Floating point

- O gforth tem *floating point* usando uma stack dedicada para *floating point*
- Antes de usar convém saber o que estamos a fazer.
- Ter boas noções de análise numérica.
- Ler pelo menos uma boa referência, como ponto de partida.
  - *What every computer scientist should know about floating-point arithmetic?* – David Goldberg, ACM Computing Surveys 23(1):5-48, March 1991

Manipulação da stack slide 39

Stack de dados

```
drop        ( n -- )
nip         ( n1 n2 -- n2 )
dup         ( n1 -- n1 n1 )
over        ( n1 n2 -- n1 n2 n1 )
tuck        ( n1 n2 -- n2 n1 n2 )
swap        ( n1 n2 -- n2 n1 )
pick        ( u -- n1 ) \ usa o stack como "array"
rot         ( n1 n2 n3 -- n2 n3 n1 )
-rot        ( n1 n2 n3 -- n3 n1 n2 )
?dup        ( n1 -- 0 | n1 n1 )
roll        ( x0 x1 ... xn n -- x1 x2 .. xn x0 )
        \ rot com n elementos
2drop       ( d -- )   \ ou ( n1 n2 -- )
2nip        ( d1 d2 -- d2 )
2dup        ( d1 -- d1 d1 )
2over       ( d1 d2 -- d1 d2 d1 )
2tuck       ( d1 d2 -- d2 d1 d2 )
2swap       ( d1 d2 -- d2 d1 )
2rot        ( d1 d2 d3 -- d2 d3 d1 )
```


Stack de retorno

- Se tivermos cuidado podemos usar a stack de retorno para guardar valores intermédios desde que antes do fim de uma definição retiremos da stack de retorno tudo o que pusemos lá.
- Isto não é bem assim, há mais cuidados a ter, mas por agora serve.

```
>r      ( n -- )    (R: -- n )
r>      ( -- n )    (R: n -- )
r@      ( -- n )    (R: n -- n )
rdrop   ( -- )      (R: n -- )
2>r     ( d -- )    (R: -- d )
2r>     ( -- d )    (R: d -- )
2r@     ( -- d )    (R: d -- d )
2rdrop  ( -- )      (R: d -- )
```

Stack de retorno

- Saber em que base estamos a trabalhar?

```
: base?   ( -- )
              base @ dup >r   \ guardar a base corrente
              decimal .       \ imprimir
              r> base !  ;    \ restaurar a base corrente
```

- A linha seguinte dá sempre 10. Porquê?

```
base @ .
```


Exemplos

```
: //        ( n1 n2 -- n3 ) \ (n1*n2)/(n1+n2)
       2dup + */  ;

: iva19%    ( n1 -- n2 )
             11900 10000 */ ;

: quadrado  ( n1 -- n2 )
             dup  * ;
```


Estruturas de controle

if

```
: testa-zero    ( n -- )
                   0 =
               if    ."  Igual a zero"
               then ;     \ a seguir a ." tem um espaço !

           : testa-zero    ( n -- )
                   0 =
               if    ."  Igual a zero"
               endif ;
```


Funcionamento

Se topo da stack for diferente de zero então é verdadeiro.

```
             : teste     ( f -- )
                if  ." Verdadeiro"
                    else ." Falso  "
                endif ;

           0 1 = teste
           4 5 < teste
           0 teste
           1 teste
```

Cuidados

- As estruturas de controle são perigosass
- Usar apenas dentro de definições
- Cuidado com efeitos na stack
- Não há avisos de *mismatch*
- O utilizador pode fazer o que quiser
- Se o utilizador escreveu aquilo é porque queria fazer aquilo !

Input/Output

```
cr          ( -- )      \ carriage return
space       ( -- )      \ espaço
spaces      ( n -- )    \ n espaços
emit        ( n -- )    \ caracter n
key         ( -- n )    \ valor ascii tecla
key?        ( -- f )    \ flag que indica
                        \ se se carregou em alguma tecla
noop        ( -- )      \ não faz nada
char caracter  ( -- n ) \ devolve o código ascii do char
```


Mais estruturas

```
if      ( flag -- )
endif   ( -- )
else    ( -- )
begin   ( -- )
again   ( -- )


: infinito  begin noop again ;  ( -- )
```


Begin. . . Until

```
begin   ( -- )
until   ( f -- )

: espera-tecla  ( -- )
    begin
        ." Waiting" cr
        key?
    until ;
```

Begin...while...repeat

```
begin       ( -- )
while       ( f -- )
repeat      ( -- )


:  espera-space         ( -- )
    begin
        key 32 <>
    while
         ." Carregue em Space" cr
    repeat ;
```


Do...loop

```
do  ( limit start --- )
loop    ( -- )

: 4stars        ( -- )
    4 0 do
        star
        loop ;
i   ( -- n )    \ valor do indice

: tloop 10 0 do     ( -- )
    i  .  cr
 loop ;
```


Do...loop

```
j   ( -- n )    \ valor do indice exterior


: 2tloop        ( -- )
    5 0 do
    5 0 do
        j . i . cr
    loop
    loop ;
```

do...+loop

```
do          ( limit start -- )
+loop       ( n -- )    \ para avançar de n em n

: pares     20 0 do         ( -- )
            i . cr
2 +loop ;
```


leave

- Sair de um `do ... loop`

```
  : tleave    ( - )
10 0 do
    i dup . 3 = if leave endif
loop
." Fim loop" ;
```


Exit

- Sair de uma definição antes do tempo

```
: safe/     ( n1 n2 -- n3 ) \ n3=n1/n2 ou n1 se n2=0
            dup
            0= if drop exit endif
            / ;
```


Cuidados com exit

- A stack de retorno deve estar limpa
- Os `do ... loop` usam a stack de retorno
- `unloop` – como usar `exit` dentro de um ciclo

```
: tloopexit         ( -- )
                10 0 do
                    i dup . 3 = if unloop exit endif
                loop
                ." Fim loop" ;
```

Output de números

Se o números têm sinal ou não, é um detalhe que só afecta a sua impressão no écran:

```
.   ( n -- )        \ imprimir um número simples com sinal
u.  ( u -- )        \ imprimir um número simples sem sinal
.r  ( n1 n2 -- )    \ imprimir um número simples com sinal
                    \ num campo de n2 caracteres
u.r ( u n -- )      \ imprimir um número simples sem sinal
d.  ( d -- )        \ imprimir um número duplo com sinal
ud. ( ud -- )       \ imprimir um número duplo sem sinal
d.r ( d n -- )      \ imprimir um número duplo com sinal
ud.r   ( ud n -- )  \ imprimir um número duplo sem sinal
s>d    ( n -- d )   \ converter um número simples num duplo
```


Output de números

- Para imprimir um número este tem de ser duplo, se for simples converte-se com `s>d` ou `0`.
- Explicação do 0 : um número duplo é posto na stack com a parte mais significativa no topo.

```
<#     ( -- )            \ começar a conversão
#      ( +d1 -- +d2 )    \ converter um digito
#s     ( +d -- 0 0 )     \ converter os digitos todos
hold   ( char -- )       \ inserir char na string
sign   ( n -- )          \ se n <0 inserir - na string
#>     ( d -- addr +n )  \ terminar conversão e deixar
                         \ endereço  para type
type   ( addr +n -- )    \ imprimir uma string
```


Output de números

- Para imprimir uma quantia em escudos, necessitamos de, no sentido da direita para a esquerda, gerar dois digitos, o sinal de cifrão e o restante do número.

```
: .esc         ( n -- )
        0 <# # # [char] $ hold #s  #>  type ;
```

- Outro exemplo: supondo que temos um número em que os dois digitos menos significativos indicam os segundos, os dois seguintes os minutos e os dois últimos as horas.
- Queremos imprimir o número no seguinte formato: `##h##m##s`

```
: .horas       ( n -- )
       0 <#  [char] s hold  # #
       [char] m hold # #
       [char] h hold  # #   #> type ;
```

Output de números

É conveniente notar que desta maneira para imprimir as horas usamos sempre dois dígitos, se quisermos eliminar o zero quando este representa ≪as dezenas de horas≫ então a definição será a seguinte:

```
:  .horas2      ( n -- )
       0 <#  [char] s hold  # #
       [char] m hold # #
       [char] h hold  #s   #> type ;
```

ORGC Linguagem Forth – slide 62

Output de números

- A meio de uma conversão pode-se mudar a base utilizada, para exemplificar vamos supor que estamos a trabalhar num forth de 32 bits, e que temos uma ≪cell≫ dividida da seguinte forma:

Bits	32	a	16	⇒	quarto campo
Bits	15	a	8	⇒	terceiro campo
Bits	7	a	4	⇒	segundo campo
Bits	3	a	0	⇒	primeiro campo

- Queremos imprimir o primeiro campo em hexadecimal, o segundo em binário, o terceiro em hex e o ultimo em decimal separados por /.

```
:   barra   [char] /  hold  ;   ( -- )
:   binary  2 base ! ;      ( -- )
:   weird.               ( n -- )
          0 <# hex # barra
          bin # # # # barra
          hex ## barra
          decimal #s barra #> type ;
```

ORGC Linguagem Forth – slide 63

Palavras deferidas slide 64

Palavras deferidas

- Podemos deixar uma definição *no ar*. Por Exemplo:

```
defer  putchar  ( n -- )    \ por convenção

       : linha         ( -- )
           begin
               key dup putchar
               13 =        \ carriage return
           until ;
       ' emit is putchar     \ para já
```

- A seguir podemos fazer:

```
       ' . is putchar
```

ORGC Linguagem Forth – slide 65

Palavras deferidas

Explicação:

```
  defer nome_palavra             \ deixa nome_palavra "no ar"
  ' nome_palavra  ( -- addr )    \ deixa na stack ender. palavra
                                 \  em "runtime"
 ['] nome_palavra ( -- addr )    \ deixa na stack ender. palavra
                                 \ em "compile time"
  is nome_palavra ( addr -- )    \ pôe nome_palavra a
                                 \ "apontar" para addr
```


Palavras deferidas

```
    defer nome      ( -- )  \ por convenção
         :  nome1   ." Da Silva" ;   ( -- )
         :  nome2   ." Da  Costa" ;  ( -- )
         :  nome3   ." Fagundes" ;   ( -- )
         ' nome1 is nome     \ a partir daqui já podemos
                             \ usar a palavra nome
```

- Experimentar:

```
nome
' nome2 is nome
nome
' nome3 is nome
nome
```


Palavras deferidas

```
    \ sem ser interactivamente
    : nome-silva        [']   nome1 is nome ;   ( -- )
    : nome-costa        [']   nome2 is nome ;   ( -- )
    : nome-fagundes     [']   nome3 is nome ;   ( -- )
```

- Experimentar:

```
nome-silva
nome
nome-costa
nome
nome-fagundes
nome
```

Palavras deferidas

```
\ agora vem a utilização mais estranha

: change-to     ' is nome ; ( -- )
```

- Experimentar:

```
change-to nome1
nome
change-to nome2
nome
change-to nome3
nome
```


Acesso à memória/dicionário

Palavras mais comuns

```
@       ( addr -- n )
2@      ( addr -- d )
c@      ( addr -- char )
!       ( n addr -- )
2!      ( d addr -- )
c!      ( char addr -- )
here    ( -- addr )     \ onde estamos na memória
,       ( n -- )        \ guarda uma "cell" na memória/dic.
c,      ( char -- )     \ guarda char (um byte se Ascii)
2,      ( d -- )        \ guarda um número duplo na memória
allot   ( n -- )        \ reserva n unidades de endereçamento
```


Tamanhos?

- Que tamanho possuem os chars/cells/duplos?
  - Depende do processador/sistema operativo...

```
chars      ( n1 -- n2 )
char+      ( addr1 -- addr2 )
cells      ( n1 -- n2 )
cell+      ( addr1 -- addr2 )
```

Palavras *Compiladoras* slide 73

Palavras Compiladoras

- Servem para criar novas palavras, logo são bastante poderosas.

```
create    nome_palavra  ( -- )
```

- cria uma nova palavra no dicionário e a partir dessa altura

```
nome_palavra ( -- addr )
```

- deixa na stack o endereço da palavra

ORGC Linguagem Forth – slide 74

Exemplos

- Criar uma variável à mão:

```
create xis 0  ,
```

- É o mesmo que:

```
variable xis
```

- Se quisermos variáveis com um valor inicial de -1

```
 : myvar     create -1 ,  ;  ( -- )

  myvar xis2              ( -- addr )
  myvar tt                ( -- addr )
```

ORGC Linguagem Forth – slide 75

Mais variáveis

A palavra `allot` apenas reserva espaço, logo se não quisermos um valor inicial nas variáveis estas poderão ser definidas como:

```
: myvar2     create 1 cells allot ;     ( -- )

  myvar2      ww      ( -- addr )
  myvar2     qqq      ( -- addr )
```

ORGC Linguagem Forth – slide 76

Arrays (DIY)

- Vamos tentar criar um array de chars:

```
: array     create  chars allot ;     ( n -- )
           \ sem inicialização   ou

: array1    create 0 do 0 c,  loop ;  ( n -- )
              \ com inicialização

     23 array1 vector1           ( -- addr )
     4 array1  vector2           ( -- addr )
```

- Assim vector1 devolve um apontador para o inicio de uma àrea de memória com 23 chars enquanto que vector2 devolve um apontador para o inicio de uma àrea com 4 chars.
- Se quisermos o endereço de um elemento do vector temos de o calcular:

```
 5 chars vector1  +      \ devolve o endereço
                           \ do elemento 5 do vector1
```


Simplificando

```
: palavra-compiladora
       create
           ......  \ palavras a executar na
           ........    \ compilação
       does>
           ........    \ palavras a executar pela
           ........    \ palavra compilada
       ;
```


Primeiro exemplo

- Variável com valor inicial fornecido pelo utilizador

```
: init-var      ( n -- )
       create    ,  ;
```

- Experimentar:

```
 2 init-var  hjk       ( -- addr )
56 init-var  uub       ( -- addr )

     hjk ?
     uub ?
```

- Neste caso não usamos o `does>` porque as palavras que estamos a criar apenas devem deixar na stack o seu endereço, o que é feito automaticamente.

Segundo exemplo

- Uma constante pode ser uma variável com um valor inicial que se coloca na stack fazendo `@` dela própria.

```
: myconst       ( n -- )
          create  ,
          does>   @   ;
```

- Ao correr cada palavra definida por `myconst` vai ler um valor da memória usando o seu endereço.

```
2 myconst dois  ( -- 2 )
7 myconst sete  ( -- 7 )
```


Definindo[2]

- Ou definindo palavras definidoras: outro exemplo será um ntuplo isto é um palavra que permite definir múltiplos:

```
: ntuplo        ( n -- )
   create  ,
   does>  @    * ;
```

- Cada uma das palavras definidas por ntuplo lê um valor da memória, e a seguir multiplica esse valor pelo topo da stack.

```
2 ntuplo duplo  ( n -- 2*n )
3 ntuplo triplo ( n -- 3*n )
```


Complicando

- Um array de chars

```
: array-chars   ( n -- )
    create chars allot
    does>  swap chars +  ;
```

- Exemplo de uso:

```
 10 array-chars vectorb      ( n -- addr )
```

- Assim: 2 `vectorb` produz o endereço do char 2 do vector

Com Cells

- Um array de cells

```
: array-cells   ( n -- )
       create cells allot
       does>  swap cells +  ;
```

- Exemplo de uso:

```
   5 array-cells vector32    ( n -- addr )
```

- Assim: 3 `vector32` produz o endereço da cell 3 do vector

Com *range-checking*

- Mais sofisticado:

```
: safe-array-cells          ( n -- )
    create dup ,     \ guardar tamanho array
    cells allot      \ reservar espaço
  does>
    2dup             \ guardar indice e endereço
    @ u<  0=         \ fetch tamanho do array
    if               \ 0=  equivale a not
      abort"   Array range error"
    endif
    1 cells +        \  saltar por cima do tamanho
                     \  do array
    swap cells + ;   \  calcular o endereço
```


Comentários

- Ao usarmos `u<` em vez de `<` a comparação além de detectar os casos em que indice é positivo e maior do que o limite, também detecta os casos em que este é negativo, uma vez que os números negativos aparecem como números positivos muito grandes.
- O uso de `abort"` não é recomendado. Foi usado para o código ficar mais simples e compreensível.
- Para um código mais correcto ver o mecanismo de excepções do gforth.

Parte 1

```
: star  42 emit ;    ( -- )
: .row               ( b -- )
       cr
     8  0 do
         dup 128 and
          if
             star
           else
             space
           endif
               2 *
           loop
       drop ;
```

Parte 2

- Agora a palavra definidora:

```
: shape   ( b1 b2 b3 b4 b5 b6 b7 b8 -- )
       create
            8 0 do c, loop
       does>
           dup 7 chars + do
           i c@ .row
           1 chars +loop
          cr ;
```

ORGC Linguagem Forth – slide 87

Parte 3

- O final:

```
hex
18 18 3C 5A 99 24 24 24     shape man       ( -- )
81 42 24 18 18 24 42 81     shape equis     ( -- )
AA AA FE FE 38 38 38 FE     shape castle    ( -- )
decimal
```

ORGC Linguagem Forth – slide 88

Palavras imediatas slide 89

Palavras *imediatas*

- São executadas mesmo que se esteja a fazer uma compilação.
- Exemplo:

```
: hello ." Hello" ;  ( -- )
: hi ." hi" ; immediate ( -- )
```

- Experimente definir:

```
: g1 hello ;
: g2 hi    ;
```

- Vamos ver mais à frente para que servem

ORGC Linguagem Forth – slide 90