LINGUAGEM YAIL (Yet Another Invented Language)

I – Esquema geral de um programa em YAIL

```
# comentários começam com o símbolo # e vão até ao fim da linha
structs{
  # definição das estruturas
}

const{
  # definição das constantes
}

global{
  # definição das variáveis globais
}

# funções
```

II – Variáveis

As variáveis básicas podem ser de 3 tipos simples:

int – valores inteiros com sinal

float – valores reais com sinal

bool – valores booleanos (true ou false)

Os nomes das variáveis são *case-sensitive* (x diferente de X), começam sempre com uma letra, maiúscula ou minúscula, seguida de zero ou mais letras, dígitos ou '_' (*underscore*).

São declaradas ou no bloco global ou no bloco local (existente em cada função). Se uma variável local tiver o mesmo nome que uma variável global, sobrepõe-se sobre esta, e a variável global passa a estar inacessível até ao término da função. Quando declaradas no bloco const, são consideradas constantes, isto é, são iniciadas, implícita ou explicitamente, e nunca mais alteradas.

As variáveis não podem ser iguais às palavras reservadas que veremos mais abaixo, nem às constantes.

Se não forem iniciadas aquando da declaração assumem o valor 0 (se do tipo int ou float) ou false (se do tipo bool). Exemplos:

```
int x; float y; bool z; # iniciadas implicitamente a 0 ou false
int x=1; float y=3.14; bool z=true; # iniciadas explicitamente
```

Conversões automáticas entre tipos básicos:

	Bool	Int	Float
bool	-	false=0; true = 1	false=0; true = 1
int	0 = false; ≠ 0 = true	-	Sem alteração
float	]-1,1[= false;]- ∞,-1]∪[1, +∞[= true	Parte inteira (truncagem)	-

Nas expressões, quando houver numa operação variáveis de tipos diferentes, as conversões serão para float quando houver uma variável do tipo float, e para int quando não haja uma variável do tipo float (int ou bool com float, converte para float; bool com int, converte para int).

Nas atribuições, as conversões são sempre feitas para o tipo da variável do lado esquerdo (variável à qual o valor é atribuído).

Exemplo:

float x = 0.5; int y = 2; bool z = true;

x = y + z; # y+z = 3 (z convertido para 1), x = 3 (sem alteração); int + bool → int → float.

y = x+z; # x+z=1.5 (z convertido para1); y=1 (truncagem); float + bool → float →int.

z = x+y; # x+y=2.5 (y convertido sem alteração); z=true; float + int → float → bool.

As variáveis podem também ser compostas, de duas formas:

- estruturas;

- vetores.

Os vetores podem ser de tipos básicos ou de estruturas.

As estruturas podem conter estruturas definidas anteriormente e vetores (se os vetores forem de estruturas, estas têm de estar definidas anteriormente).

As estruturas são definidas no espaço próprio e são da seguinte forma:

nome { campos da estrutura };

Exemplo:

```
structs { #início
  point2D {float x,y;}; # point2D é composta por 2 reais x e y
  point3D {float x,y,z;}; # point3D por 3 reais x, y e z
  pointND {float x[];}; # pointnD é um vetor de reais
} #fim
```

Para aceder a cada campo da estrutura usa-se o operador '.' (ponto). Exemplo:

```
point2D p;
```

p.x → campo x da variável to tipo point2D p;

Para estruturas dentro de estruturas, o princípio é igual:

```
structs{
  a1 {int x, y;};
  a2 {a1 a, b;};
}
# parte do código
a2 w;
```

w.a.x → variável x dentro da variável a do tipo estrutura a1, que por sua vez é parte da variável w dentro na estrutura a2.

Os vetores são dinâmicos, podendo o seu tamanho variar ao longo do programa. Podem ser declarados sem tamanho ou com um tamanho inicial inteiro maior do que 0, e podem ou não ser iniciados.

```
int v[]; # por omissão, é iniciado com o tamanho 1 e o valor 0
int v[]={1,2,3}; # se for iniciado sem indicação do tamanho,
                 # assume o tamanho da lista dada
int v[10]; # o vetor tem tamanho 10, inicia tudo com 0
int v[10] = {1,2,3}; # inicia os 3 primeiros, o resto é 0
int v[2] = {1,2,3}; # ERRO!!! Tamanho da lista maior que o vetor
int v[x+2*y]; # inicia com o valor da expressão x+2*y,
              # se for maior que 0, caso contrário dá erro
# pode ser usada a função especial interna
# gen(valor_inicial, incremento)
# o valor inicial é o elemento de índice 0,
# o incremento é o valor que se soma a cada passo
# seria o equivalente a ter um ciclo for para iniciar o vetor
# tanto o valor inicial como o de cada elemento são calculados
# como valores do tipo float
# para vetores do tipo int e do tipo bool,
# as conversões são feitas segundo as regras indicadas.
int v[10] = gen(0,1); # igual a int v[10]={0,1,2,3,4,5,6,7,8,9};
```

```
int v[10] = gen(9,-1); # igual a v[10] = {9,8,7,6,5,4,3,2,1,0};
int v[10] = gen(1,0); # igual a v[10] = {1,1,1,1,1,1,1,1,1,1};
# existem mais duas funções especiais internas para vetores
# size(v) dá o tamanho do vetor v, -1 se não existir
# resize(v,n) redimensiona o vetor para o tamanho n,
# devolve n se conseguir, -1 caso contrário
```

III – Funções

As funções seguem o seguinte esquema:

```
nome(parâmetros) tipo_de_retorno { #início do bloco da função
  local {
  # declaração de variáveis locais
  }
  # resto da função
} # fim da função
```

Os nomes das funções obedecem às mesmas regras das variáveis. Quando se cria uma função, cria-se implicitamente uma variável com o mesmo nome, onde está colocado o valor de retorno.

Uma função pode ter zero ou mais parâmetros, sendo cada um referenciado pelo tipo e pelo nome. A passagem é sempre feita por valor, nunca por referência. O tipo de retorno só pode ser uma variável básica (int, float ou bool).

A primeira função a ser chamada é sempre a função main.

(NOTA: as funções especiais internas podem fugir à regra)

Exemplos:

```
distance3D(Point3D p,Point3D q) float {
  distance3D = square_root(pow((q.x-p.x),2) + pow((q.y-p.y),2) +
                pow((q.z-p.z),2));
}

distance2D(Point2D p,Point2D q) float {
  distance2D = square_root(pow((q.x-p.x),2) + pow((q.y-p.y),2));
}
```

```
distanceND(PointND p,PointND q) float {
  local { int i, n = size(p.x);}
  if (size(q.x) == n){
    for(i,0,n-1,1){
      distanceND += pow((q.x[i]-p.x[i]),2);
      }
    distanceND = square_root(DistanceND);
    }
  else {
    distanceND = -1;
  }
}
```

Verificamos aqui a atribuição do valor de retorno, usando para tal a variável com o mesmo nome da função.

Usámos ainda duas funções especiais internas:

- pow(base, potência); eleva o valor na base à potência dada (ambos os valores são float);

- square_root(valor); calcula a raiz quadrada.

Também verificamos já a instrução condicional if e o ciclo for.

O ciclo for tem 4 parâmetros: a variável, o valor inicial, o valor final, o incremento. Quando o valor final for ultrapassado (for maior, se o incremento for positivo, ou menor, se o incremento for negativo; se o incremento for 0, dá erro). As instruções estão sempre entre chavetas, seja qual for o número de instruções dentro do ciclo.

A instrução condicional if pode ou não ter um else correspondente. Terá sempre chavetas, independentemente do número de instruções associadas. O if tem como argumento uma condição, cujas regras veremos mais abaixo.

Existe ainda o ciclo while, que tem também como argumento uma condição. Exemplo:

```
while(x>0) {
  x -= 1000;
}
```

IV – Atribuições, Expressões e Condições

As atribuições são da forma var = expressão. Exemplo:

x = 3 * y + 5;

Os termos da expressão podem ser valores literais (inteiros, reais ou booleanos) ou variáveis básicas (do tipo int, float ou bool), podendo pertencer a estruturas ou vetores, e funções (incluindo funções especiais internas). Terão sempre de ser verificadas as conversões automáticas.

Os operadores das expressões são +,-,*,/ e % (resto da divisão inteira). Existem duas formas de simplificar atribuições:

- var op= expressão; # o mesmo que var = var op expressão;

- var ++; var--; # o mesmo que var = var + 1; e var = var – 1;

Os operadores condicionais são <,>,<=,>=,==,!=.

Os operadores lógicos, aplicáveis às condições são and, or e not.

As regras de associatividade e precedências são as habituais.

V – Input/Output

Para entrada e saída de dados, existem seis funções especiais internas que permitem ler do standard input e escrever no standard output.

Para escrita, temos as seguintes:

write(arg1, arg2, …, argn);

write_all(vector_or_struct);

write_string(int_vector);

Começando pelo write, vejamos alguns exemplos:

```
# write muda sempre de linha no fim da instrução
write(); # escreve uma linha em branco
write("Olá"); # escreve o texto entre "" e muda de linha
write("Março tem ", meses[2], " dias.");
# supondo meses[2] igual a 31, escreve
Março tem 31 dias.
# o mesmo seria válido para meses(2),
# supondo que a função meses retorna um valor inteiro 31
# tudo o que está entre "" é escrito como texto
# nos restantes é escrito o respetivo valor
```

```
# de acordo com o tipo de variável
# não há carateres de escape no texto
# apenas as aspas têm de ser repetidas
write(“A palavra reservada ””if”” não pode ser variável.”);
A palavra ”if” não pode ser variável.
# write_all escreve todos os valores de uma estrutura ou vetor
# supondo que v[4]={0,1,2,3}
write_all(v);
0, 1, 2, 3
#supondo que a variável do tipo point2D p1 tem os valores 1,2
# e a variável do tipo pointND p2 tem os valores 1,2,3,4
write_all(p1);
1, 2
write_all(p2);
1, 2, 3, 4
# a ordem é a das variáveis na estrutura
# e o índice dos vetores
# write_string permite escrever um vetor de inteiros como string
# usa o respetivo código ASCII
# acrescenta mudança de linha
# se tiver s[4]={89, 65, 73, 76}
write_string(s);
YAIL
```

Para leitura, temos também 3 funções especiais internas:

read();

read_all();

read_string();

```
# read lê a entrada e interpreta conforme o tipo da variável
# à qual o resultado vai ser atribuído
# tudo o que não for necessário após uma leitura correta
# é descartado
```

```
# caso a entrada não tenha nada de relevante no início
# são atribuídos os valores 0 ou false
# suponhamos int n, float x e bool v
n = read();
# escrevemos “12x”, então n=12
# escrevemos “x12”, então n=0, pois x é lido primeiro
# escrevemos “3.14”, então n=3 (converte float para int)
# escrevemos “qualquer coisa”, então n=0
# escrevemos “true”, então n=1 (converte bool para int)
x = read();
# escrevemos “12x”, então x=12
# escrevemos “x12”, então x=0, pois x é lido primeiro
# escrevemos “3.14”, então x=3.14
# escrevemos “qualquer coisa”, então x=0
# escrevemos “true”, então x=1 (converte bool para float)
v = read();
# escrevemos “12x”, então v=true (converte int para bool)
# escrevemos “x12”, então v=false, pois x é lido primeiro
# escrevemos “3.14”, então v=true (converte float para bool)
# escrevemos “qualquer coisa”, então v=false
# escrevemos “true”, então v=true
# read_all() segue as mesmas regras do read()
# mas pede explicitamente os valores um a um
# se tivermos int v[4] e point2D p
v = read_all();
int v[4]
v[0]: # introduzir valor como para read()
v[1]:
v[2]:
v[3]:
p = read_all();
struct point2D p
p.x:
```

p.y:

```
# read_all mostra ao utilizador a estrutura interna
# caso não se queira isso, tem de se fazer a leitura item a item
# Finalment, read_string lê a entrada
# e converte para código ASCII
# o valor é sempre atribuído a um vetor de inteiros
# se a entrada lida for maior que o vetor, é truncada
# e o resto descartado
# se for menor, então o resto é preenchido com zeros
# consideremos int v[5];
v = read_string();
# escrevemos “YAIL”, então v[]={89,65,73,76,0}
# escrevemos “Hello”, então v[]={72,101,108,108,111}
# escrevemos “Hello, world!”, então v[]={72,101,108,108,111}
# escevemos “true”, então v[]={116,114,117,101,0}
# escrevemos “100”, então v[]={49,48,48,0,0}
```

Exemplo de programa 1 – fatorial (versão não recursiva)

```
# Programa fatorial, versão não recursiva
# Não tem estruturas nem variáveis globais
const{ int max=100; }

main() bool {
  local {int n};
  write("introduza um número entre 0 e ",max);
  n = read();
  if(n > max or n < 0){
    write("Erro!!!");
    }
  else {
    write(n,"! = ",fact(n));
    main = true; # altera o valor de retorno da função main
    }
}

fact(int n) int {
  {local int i;}
  fact=1; #altera valor de retorno para 1
  for(i,2,n,1){
    fact *= i;
    }
  # fact tem o valor de retorno
}
```

Exemplo de programa 2 – fatorial (versão recursiva)

```
# Programa fatorial, versão recursiva
# Não tem estruturas nem variáveis globais
const{ int max=100; }

main() bool {
  local {int n};
  write("introduza um número entre 0 e ",max);
  n = read();
  if(n > max or n < 0){
    write("Erro!!!");
    }
  else {
    write(n,"! = ",fact(n));
    main = true; # altera o valor de retorno da função main
    }
}

fact(int n) int {
  if(n>1){ fact = n * fact(n-1); }
  else { fact = 1; }
}
```

Exemplo de programa 3 – distância entre dois pontos 3D

```
# Programa dist3D
# Não tem constantes nem variáveis globais
structs {
  point3D {float x,y,z;};
}
main() bool {
  local {point3D p1,p2; };
  write("introduza os dois pontos");
  p1 = read_all();
  p2 = read_all();
  write("Distância entre p1=(",write_all(p1),") e p2 =(",
         write_all(p2),") é igual a ", distance3D(p1,p2));
  main = true; # altera o valor de retorno da função main
}

distance3D(Point3D p,Point3D q) float {
  distance3D = square_root(
                 pow(q.x-p.x,2) +
                 pow(q.y-p.y,2) +
                 pow(q.z-p.z,2));
}
```

CRITÉRIOS GERAIS

(estes critérios vão ser esmiuçados em breve)

e-fólio A: 4 valores

análise léxica (2 valores): reconhecer corretamente todos os *tokens* da linguagem

análise sintática (2 valores): reconhecer corretamente a estrutura sintática de um programa escrito em YAIL (NOTA: nesta fase ainda não é necessário verificar as conversões entre tipos).

e-fólio B: 4 valores

geração de código intermédio (2 valores): gerar código intermédio, pode ser para TAC (Three Address Code), para uma linguagem de pilha ou notação pós-fixa.

otimização de código (2 valores): otimizar o código, ao nível das expressões.

e-fólio global: 12 valores

análise léxica (2 valores): reconhecer corretamente todos os *tokens* da linguagem

análise sintática (2 valores): reconhecer corretamente a estrutura sintática de um programa.

geração de código intermédio (2 valores): gerar código intermédio, pode ser para TAC (Three Address Code), para uma linguagem de pilha ou notação pós-fixa.

otimização de código I (2 valores): otimizar o código, ao nível das expressões.

otimização de código II (2 valores): otimizar o resto do código, incluindo ciclos e outros.

geração de código final (2 valores): pode ser assembly executável online, código máquina a correr num linha de comandos ou bytecodes para java.