1 Actividade 0x07 - Parser para a linguagem *Ya*!

Agora que estamos à vontade para gerar a APT a partir de uma linguagem simples com expressões aritméticas, podemos começar a preparar um compilador para uma linguagem mais completa. Nesta actividade, iremos estender a linguagem *CaLC* com declarações, statements, tipos e arrays.

2 A linguagem *Ya*!

2.1 Léxico e sintaxe

A linguagem *Ya*!obedece às seguintes especificações:

- Um programa é uma sequência de declarações;
- Todas as instruções são terminadas por ponto e vírgula (`;`)
- Uma declaração pode ter os seguintes formatos (exemplos):
  - `i: int` (declaração de variável)
  - `i: int = 1` (declaração com valor de inicialização)
  - `i,j,k: int = 1` (declaração múltipla com valor de inicialização - todas as variáveis ficam com o mesmo valor)
  - `f(): int { <corpo> }` (declaração de função)
  - `f(a: int, b: bool): int { <corpo> }` (função com argumentos)
  - `define Nome Tipo` (declaração de novo tipo)
- Os tipos pré-definidos são os seguintes:
  - `int`
  - `float`
  - `string`
  - `bool`
  - `Tipo[IntExp]` (array com elementos do tipo `Tipo`)
  - `void` (tipo para funções sem valor de retorno - procedimentos)
- Os literais têm o formato "habitual":
  - Inteiros (`1; 30; 5000`)
  - Floats (`1.2; 0.1; .23; .22e-20`)
  - Strings (`"hello, world!"; "1.2"`)
  - Bools (`true; false`)
- Expressões binárias:
  - `+, -, *, / (int, float)`
  - `mod, ^ (int, float)`
  - `==, != (int, float, bool, string, arrays)`

  - ◇ `<, >, <=, >= (int, float)`
  - ◇ `and, or (bool)`
- Expressões unárias:
  - ◇ `-` (valor negativo)
  - ◇ `not` (negação booleana)
- Afectações também são expressões:
  - ◇ `a = 1`
  - ◇ `a = b = c = 1`
  - ◇ `a[20] = b[i=2] = 3 - x`
- O corpo de uma função é constituído por statements. Statements podem ser:
  - ◇ Declarações (de variáveis locais, não existem declarações de funções dentro de funções);
  - ◇ Expressões (caso especial para afectações, outras expressões não produzem código "interessante" – mas podem ser aceites);
  - ◇ Instrução de retorno (`return Exp`)
  - ◇ Condicionais:
    - `if BoolExp then { <corpo> }`
    - `if BoolExp then { <corpo> } else { <corpo> }`
  - ◇ Ciclos (`while BoolExp do { <corpo> }`)
- O corpo dos ciclos dos ciclos e condicionais é semelhante ao das funções.
- Um ciclo pode ser forçado a terminar com a instrução `break`, ou forçado a passar à próxima iteração, com a instrução `next` (equivalente ao `continue` do C ou Java).

2.2 Palavras reservadas e símbolos

- `; " ( ) [ ] { } . , : =`
- `+ - * / ^`
- `== < > <= >= !=`
- `mod and or not`
- `int float string bool void`
- `define if then else while do`
- `return break next`

2.3 Funções pré-definidas (parte da "biblioteca" do *Ya*!)

- `print(Exp)` → mostra o resultado de `Exp` no ecrã
- `input(lvalue)` → guarda um valor escrito no teclado em `lvalue` (tendo em conta o tipo do *lvalue*)

3 Exercício

Pode usar o código desenvolvido na actividade anterior como base. No entanto, deve ter cuidado com as diferenças, nomeadamente:

- O separador deixa de ser o símbolo `'\n'`, que passa a ser ignorado, passando a ser o símbolo `';'`
- É necessário diferenciar literais inteiros de literais de vírgula flutuante

3.0.1 Código a implementar

1. Implemente os analisadores lexical e sintáctico para a linguagem *Ya*![1]. Não se preocupe inicialmente com as acções semânticas nem com a geração da APT.
2. Implemente as estruturas da sintaxe abstracta e as acções semânticas que permitam gerar a APT.

[1] Como falado nas aulas, pode começar por implementar um subconjunto da linguagem.