Léxico e sintaxe da linguagem *Ya!*

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 - e 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).

Palavras reservadas e símbolos

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

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*)

TRADUÇÃO APT → RI

- Para cada nó da APT geram-se nós de RI
- Levantam-se algumas questões:
  - Será que um nó do tipo *expressão* na APT deverá ser representado por um nó do tipo *expressão* na RI?
  - Acessos a variáveis, encaradas como endereços de memória.
    - Variáveis locais: `MEM(BINOP(+, TEMP(fp), CONST(k)))`
    - Variáveis externas: `MEM(BINOP(+, LABEL(DISPLAY), CONST(SCOPE)))`
    - Isto gera sempre um *LOAD* ou um *STORE*
    - No caso de arrays ou structs, o princípio é o mesmo, mas as expressões serão mais complexas, pois terão de incluir o cálculo do endereço como uma base e um deslocamento.
  - Declarações
    - Alocação de espaço na stack frame
    - Inicialização de variáveis gera instruções RI para colocar o valor inicial na zona de memória correspondente da stack frame

TRADUÇÃO APT → RI

- Declarações de função
  - Produção de código específico para entrada (prólogo) e saída (epílogo) da função
  - Prólogo
    - Definição de uma label para a função
    - Ajuste do *stack pointer*, suficiente para acomodar as variáveis locais e temporárias: no fundo, o espaço para o RA.
    - Instruções para guardar o contexto que for necessário no RA: guardar os registos *callee-saved*; ajustar o *static link* (via `fp` e ajuste do *display*)
  - Corpo da função
  - Epílogo
    - Guardar o valor de retorno no local estipulado
    - Repor os registos *callee-saved*
    - Repor o *static link*
    - Repor o *stack pointer*
    - Salto para o *return address*, efectivando o regresso à função chamadora

STATIC LINK, DYNAMIC LINK

Em linguagens com procedimentos imbricados, como referir os *frame pointers* das funções de nível superior?

- Mecanismo de *DISPLAY*
  - Vector (array) global de apontadores (endereços) para o registo de activação mais recente, indexado pelo nível lexical dos procedimentos/funções.
  - Estritamente necessário quando a linguagem permite declarar procedimentos imbricados.
  - Mas também *dá jeito* para referir contextos superiores (e.g., variáveis globais em *Ya!*)
- Em alternativa, podemos usar *dynamic linking*, onde se usa a cadeia de referências de *old fp*s

ÁRVORES CANÓNICAS

- Objectivo: tornar a RI mais próxima de arquitecturas reais
- Método: transformações simples ao nível dos nós
  - `CJUMP`: pode saltar para dois endereços distintos; em máquinas reais, apenas se salta num dos casos, continuando para a próxima instrução no outro caso;
  - Os nós `ESEQ` dentro de expressões são inconvenientes, porque o resultado poderá ser diferente consoante a ordem de avaliação das sub-árvores;
  - Nós `CALL` dentro de expressões sofrem do mesmo problema;
  - Nós `CALL` dentro de outros nós `CALL` dão problemas se a convenção de chamada impuser a colocação dos parâmetros formais em registos específicos.

REGRAS PARA REESCRITA

- Uma *árvore canónica* goza de duas propriedades:
  1. Não contém `ESEQ`s dentro de `SEQ`s
  2. O *pai* de um `CALL` é sempre um `EXP(...)` ou um `MOVE(TEMP(t), ...)`

TRANSFORMAÇÕES

- `ESEQ`: mover os nós para "cima", o mais possível, até poderem ser transformados em nós `SEQ` (na prática, transformar a árvore numa lista)
- Elevação dos nós `CALL`: reescrever de forma a que um `CALL` dentro de outro `CALL` passe a ser um `ESEQ` que é avaliado antes do `CALL` superior, com o seu resultado guardado num temporário.
- `CJUMP`s: reordenar as sequências de instruções construídas anteriormente, de forma a garantir que um `CJUMP` é imediatamente seguido pela sequência que começa no *label* referido no *false* do `CJUMP`

REGRAS PARA TRANSFORMAÇÕES (1)

- `ESEQ(s1, ESEQ(s2, e))` → `ESEQ(SEQ(s1, s2), e)`
- `BINOP(op, ESEQ(s, e1), e2)` → `ESEQ(s, BINOP(op, e1, e2))`
- `MEM(ESEQ(s, e))` → `ESEQ(s, MEM(e))`
- `JUMP(ESEQ(s, e))` → `SEQ(s, JUMP(e))`
- `CJUMP(op, ESEQ(s, e1), e2, t, f)` → `SEQ(s, CJUMP(op, e1, e2, t, f))`

REGRAS PARA TRANSFORMAÇÕES (2)

- `BINOP(op, e1, ESEQ(s, e2))` → `ESEQ(MOVE(TEMP(t), e1), ESEQ(s, BINOP(op, TEMP(t), e2)))` → `ESEQ(SEQ(MOVE(TEMP(t), e1), s), BINOP(op, TEMP(t), e2)))`
- `CJUMP(op, e1, ESEQ(s, e2), l1, l2)` → `SEQ(MOVE(TEMP(t), e1), SEQ(s, CJUMP(op, TEMP(t), e2, l1, l2)))`

REGRAS PARA TRANSFORMAÇÕES (3)

- `BINOP(op, e1, ESEQ(s, e2))` → `ESEQ(s, BINOP(op, e1, e2))`
- `CJUMP(op, e1, ESEQ(s, e2), l1, l2)` → `SEQ(op, e1, e2, l1, l2)`

(Nestes dois casos, apenas se `s` *e* `e1` *comutarem)*

BLOCOS BÁSICOS

- Reordenar as sequências de instruções, de forma a garantir que uma sequência que termina num CJUMP seja imediatamente seguida pela sequência que começa pelo *label* referido no *false* do CJUMP
- Análise do *fluxo de controlo* do programa: observa-se a RI disponível, ignorando tudo o que não forem *labels* nem saltos.
- Um *bloco básico* é uma sequência de instruções em que:
  - A primeira é um LABEL
  - A última é um JUMP ou um CJUMP
  - Não há nenhum outro LABEL, JUMP ou CJUMP
- A divisão em blocos básicos pode resultar em qualquer ordem para o programa, que o significado será o mesmo

TRAÇOS

- Dado um programa dividido em blocos básicos, a forma como eles se ordenam relativamente uns aos outros pode influir no código produzido.
- Chama-se um *traço* a uma sequência de blocos básicos que reflecte o programa original
- Construímos um conjunto de traços que cubra todo o programa (i.e., que inclua todos os blocos básicos). Note-se que o conjunto final preserva o significado do programa, pois este não depende da ordem relativa dos blocos.
- Se o bloco termina num JUMP, não há dúvida quanto ao que fazer
- Se termina num CJUMP, podemos optar por construir um traço que siga por qualquer uma das saídas. Por conveniência:
  - Um CJUMP seguido pela sua *label* false fica inalterado;
  - Num CJUMP seguido pela sua *label* true, trocam-se os labels true e false e inverte-se a condição
  - Num CJUMP que não é seguido de nenhuma das suas *label*s, cria-se um bloco básico artificial (que o segue) com duas instruções: um LABEL e um JUMP.

LIVENESS ANALYSIS

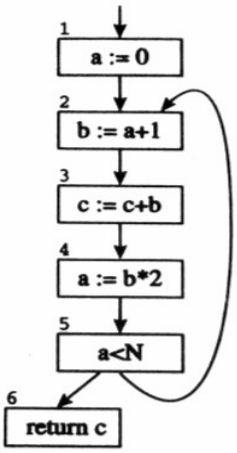
- Construímos um grafo de fluxo
- Variável "viva": o seu valor vai ser usado no futuro
- Análise faz-se do futuro para o passado.
- Se a variável é *usada* num nó, significa que está viva à entrada desse nó

GRAFO DE FLUXO

Programa

```
1       a = 0
2  L1:  b = a + 1
3       c = c + b
4       a = b * 2
5       if a < N goto L1
6       return c
```

GRAFO DE FLUXO



LIVENESS ANALYSIS (2)

- b está viva em $\{2 \to 3\}$ e $\{3 \to 4\}$
- a está viva em $\{1 \to 2\}$ e $\{4 \to 5\}$
- c está viva desde o início até ao fim. (talvez seja um argumento da função?)
- Se c não é um argumento, então detectámos uma variável não inicializada!

ATRIBUIÇÃO DE REGISTOS

- Problema NP-Completo
- Tal como coloração de grafos...
- Usamos o algoritmo BSSS que dá uma boa aproximação, em tempo linear
- (Build, Simplify, Spill and Select)

ALGORITMO BSSS - BUILD

- Criar o grafo de interferências
- Cada temporário representa um vértice do grafo
- Temporários que estão vivos em simultâneo têm um arco a ligá-los

ALGORITMO BSSS - SIMPLIFY, SPILL

- K = número de registos disponíveis = número de cores do grafo
- Recursivamente, removemos do grafo os vértices com menos de K arcos e colocamo-los numa pilha.
- Se num momento do passo anterior o grafo só tem vértices de grau >= K, escolhemos um desses nós para *spilling*, removendo-o para a pilha. Continuar com a simplificação.

ALGORITMO BSSS - SELECT

- Retiram-se, um a um, os vértices da pilha e atribui-se uma cor não usada em nenhum "vizinho"
- Quando retiramos um vértice colocado na pilha sob ameaça de *spilling*, pode haver ainda cor para ele, pois apesar de ter mais do que K vizinhos, alguns deles podem ter cores iguais.
- Se não houver cor disponível, marcamos esse vértice como *spilled* e continuamos o *Select*.
- Se ficámos com temporários *spilled*, temos de reescrever o programa de forma a que esses temporários se transformem em vários temporários (cada um), com um tempo de vida mais curto (vão ser transferidos de e para a memória).
- Começar de novo, pois estes novos temporários podem ter influência no resto do grafo.
- Normalmente, o processo termina em uma ou duas iterações (quando o *Simplify* termina sem dar origem a *spills*.

COALESCING

- Quando no grafo de interferências não há arco entre a origem e o destino de um `move`, as variáveis podem ser *aglutinadas* numa só, eliminando a necessidade do `move`.
- Agressivamente, quaisquer duas variáveis que não têm um arco entre elas podem ser aglutinadas.
- É preciso cuidado, pois a variável resultante vai ter a conjunção dos arcos das duas que lhe deram origem (podemos precisar de mais *spills*...)

GARBAGE COLLECTION

- Não se faz em *compile-time*
- São procedimentos que se *link*am ao executável, para correr em *run-time*
- Vários algoritmos *standard*
  - Mark-&-sweep
  - Copying collection
  - Generational collection
  - Incremental collection
  - Etc.

MARK-AND-SWEEP (MARK)

Depth-first search

```
1  function DFS(x)
2    if x is a pointer to the heap
3      if *x is not marked
4        mark x
5        for each field f(i) of record x
6          DFS(x.f(i))
```

MARK-AND-SWEEP (SWEEP)

Sweep

```
1  let p = first heap address
2  while p < last heap address
3    if *p is marked
4      unmark p
5    else let f(i) = first field in p
6      p.f(i) = freelist
7      freelist = p
8    p = p + sizeof(*p)
```

OUTRAS OPTIMIZAÇÕES (1)

- Remoção de `loads` e `stores` redundantes

```
1          MOVE R0, a
2          MOVE a, R0
```

- Código inacessível

```
1          #define DEBUG 0
2          . . .
3          if (DEBUG) {
4            . . .
5          }
```

OUTRAS OPTIMIZAÇÕES (2)

- Saltos para saltos

```
1          goto L1
2          . . .
3      L1: goto L2
4          . . .
5      L2:
```

- Simplificação algébrica

```
1          x := x + 0
2          /* ou */
3          x := a * 0
```

Universidade de Évora

Compiladores

1ª Frequência - Correcção

(1) 1. Considere o seguinte conjunto de números de vírgula flutuante:

`{1.0, .365, 0.234E25, 0.123e--10, .23e+15}`

Qual das seguintes expressões regulares identifica números (literais) do tipo dos que pertencem ao conjunto? (considere a notação do `flex`)

A. `[0-9]\.[0-9]*[Ee]?[0-9]+`

B. `[0-9]?\.[0-9]+([Ee]?[0-9]+)*`

C. `[0-9]*\.[0-9]+([Ee][+-]?[0-9]+)?`

D. `[0-9]+(\.)?[0-9]*[Ee+-]?[0-9]?`

2. Usando a questão anterior como exemplo, explique como se processam, em termos de compilação, os casos em que:

(1) (a) o número é negativo (e.g., `{-1.0, -.365, -0.234E25, -0.123e-10, -.23e+15}`).

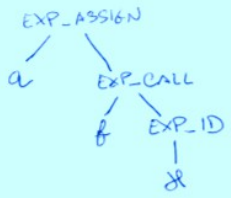
> **Solução:** Os números são *tokens* sem sinal. O sinal de menos é um token. Na gramática há uma regra para o menos (unário), que determina que um sinal de menos antes de algo é uma operação unária. Resumindo, os números negativos são tratados como operações unárias cujo operador é `-`.
>
> (Nota: geralmente, é feito o mesmo para o sinal de `+`, mas este pode ser ignorado quando se gera a APT).

(0.75) (b) aparecem vários sinais (e.g., `3.5 - -2.1`).

> **Solução:** Neste caso temos uma operação unária de `-` aplicada ao segundo número e uma operação binária de `-` entre dois argumentos. Resumindo, temos uma subtracção (normal) entre uma expressão que é um literal float e outra expressão que é uma operação unária de `-` aplicada a outra expressão que é outro literal float.

3. Para as seguintes linhas de código, proponha uma representação para a sintaxe abstracta e desenhe as APTs respectivas:

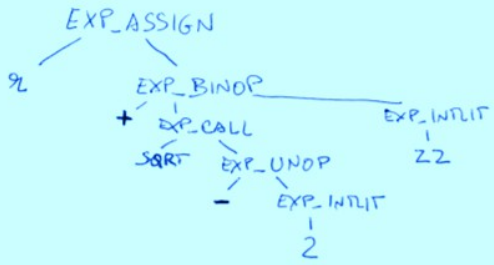
(1.5) (a) `a = f(x);`

> **Solução:** Para este exemplo, precisamos de 3 tipos de nós na sintaxe abstracta:
>
> - `exp_assign(ID, EXP)`
> - `exp_call(ID, ARG)` (simplificamos e assumimos que só há 1 argumento)
> - `exp_id(ID)`
>
> 
>

(1.5) (b) `r = sqrt(-2) + 22;`

Solução: Acrescentamos alguns nós à sintaxe abstracta:

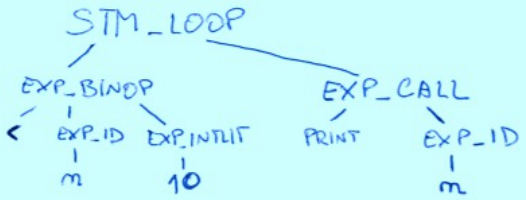
- `exp_binop(OP, EXP, EXP)`
- `exp_unop(OP, EXP)`
- `exp_intlit(NUM)`



(1.5) (c) `while (n<10) do { print(n); };`

Solução: Para este excerto, já temos quase tudo, apenas nos falta um nó para o ciclo:

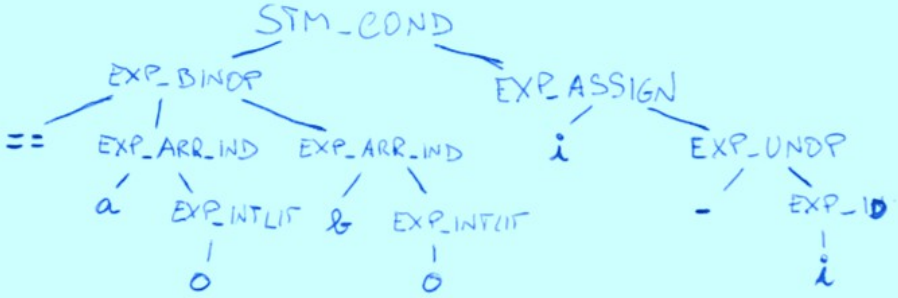
- `stm_loop(COND, BODY)` (vamos assumir que o body tem só uma instrução)



(1.5) (d) `if a[0] == b[0] then { i = -i };`

Solução: Agora temos um condicional (igual ao ciclo, para este exemplo até podemos ignorar o *iffalse branch*), mas a maior novidade é o acesso a índices de array:

- `stm_cond(COND, IFTRUE)`
- `exp_arr_ind(ID, EXP)`



4. Considere o processo de compilação de um determinado programa. Na fase da análise semântica, obteve-se a seguinte evolução da *Symbol Table*:

1.

id	type	args
f	float	int, int
a	int(arg)	
b	int(arg)	
i	int	
r	float	

ENV

2.

id	type	args
f	float	int, int
g	int	float
a	float(arg)	
i	int	

ENV

3.

id	type	args
f	float	int, int
g	int	float
o	int	
main	void	void
t	string	

ENV

(1) (a) A *Symbol Table* apresentada ainda sofrerá alterações até ao fim da análise semântica? Justifique.

Solução: Sofrerá pelo menos uma alteração: ao chegar ao fim da sub-árvore da função `main()`, será chamada a função `drop_environment()`, que remove o scope de nível 2 da Symbol Table (ou seja, a variável `t` vai desaparecer).

Por outro lado, nada sabemos sobre o código do programa nem sobre a linguagem, consoante os quais ainda podemos ter:

- mais variáveis na função `main()`
- mais variáveis globais depois da função `main()`
- mais funções declaradas após a função `main()`

.5) (b) Proponha um excerto de programa que possa ter dado origem à *Symbol Table* apresentada.

Solução: Basta um simples exemplo, mesmo sem código "executável":

```
f(a:int, b:int) : float {
  i: int;
  r: float;
}
g(a:float) : int {
  i: int;
}
o: int;
main() : void {
  t: string;
}
```

Note-se que a variável `o` é global, pois não aparece dentro de nenhum contexto local na Symbol Table.

5. Considere o seguinte programa em *Ya!*:

```
1  factrec (n: int) : int {
2    a,b: float;
3
4    if n == 1 then { b = 1; }
5    else {
6      a = n - 1;
7      b = n * factrec(a);
8    };
9    return b;
10 };
11
12 main () : void {
13   a: int[10];
14   print(factrec(3));
15 };
```

(1.5) (a) Suponha que se introduz uma nova linha de código entre as linhas 7 e 8: `a[1] = 0`. Que tipo de erro estamos a introduzir? Justifique em 10 palavras ou menos.

Solução: Estamos a introduzir um erro semântico, pois apesar de a sintaxe estar correcta, a variável `a` não é de um tipo indexável.

(1.5) (b) Mostre uma representação da *Symbol Table*, aquando da análise semântica do ramo da APT correspondente ao código da linha 8.

Solução:

id	type	args
factrec	int	int
n	int(arg)	
a	float	
b	float	

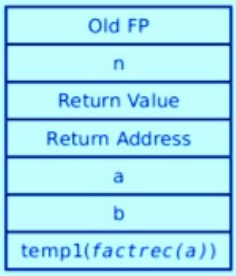
ENV

(2) (c) Explique, de forma sucinta, como é feita a distinção entre a variável `a` da função `factrec` e a variável `a` da função `main`, durante o processo de compilação.

Solução: Na análise semântica, as variáveis ficam em contextos diferentes (nunca coexistem na Symbol Table. Durante a execução, cada uma está num registo de activação diferente.

(1.25) (d) Proponha um desenho para o *Registo de Activação* da função `factrec()`.

Solução: Uma hipótese, assumindo que o valor da chamada recursiva necessita de um temporário:



(1.25) (e) Considerando o desenho proposto por si na alínea anterior, dê uma estimativa do espaço máximo ocupado na *stack* pela função `factrec()`, assumindo a execução do código da linha `14`.

Solução: Aqui basta uma estimativa. Assumindo que a arquitectura de destino é de 32 bits, o RA apresentado (7 words) ocuparia $7 \times 4 = 28$ bytes. Como a chamada vai ser recursiva 2 vezes, o tamanho máximo seriam 84 bytes.

Se não considerarmos a chamada com o argumento 3, então a stack poderá crescer infinitamente (característica das funções recursivas).

(1.25) (f) Suponha agora que alguém implementou um compilador para a linguagem *Ya!*, do qual não sabemos nada sobre a sua implementação. Ao correr esse compilador sobre o código anterior, obtém-se a seguinte mensagem de erro:

```
Erro de tipos na linha 7.
```

Resumidamente, e assumindo que o compilador está bem implementado (com excepção das mensagens de erro não estarem detalhadas), dê uma ideia sobre o que se passa para o erro ter ocorrido.

Solução: Claramente, este compilador não faz a conversão implícita entre `int` e `float`... O erro tanto pode ser por estarmos a passar um argumento `float` à função `factrec()`, como por estarmos a afectar o resultado de `factrec()` (`int`) à variável `b` (`float`).