



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE DE  
**COIMBRA**

# Compiladores

## Relatório Projeto

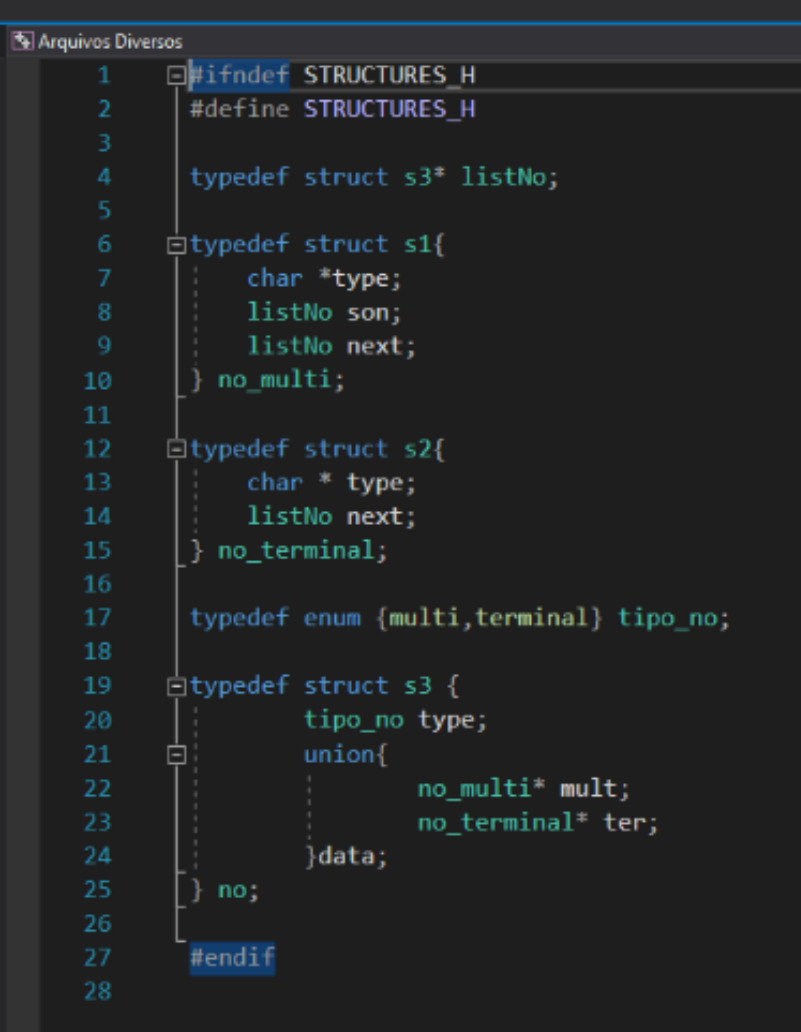
Francisco Faria Nº 2019227649

Iago Bebiano Nº 2019219478

Departamento de Engenharia Informática

2021/2022

## Meta 2: gramática re-escrita



```
1  #ifndef STRUCTURES_H
2  #define STRUCTURES_H
3
4  typedef struct s3* listNo;
5
6  typedef struct s1{
7      char *type;
8      listNo son;
9      listNo next;
10 } no_multi;
11
12 typedef struct s2{
13     char * type;
14     listNo next;
15 } no_terminal;
16
17 typedef enum {multi,terminal} tipo_no;
18
19 typedef struct s3 {
20     tipo_no type;
21     union{
22         no_multi* mult;
23         no_terminal* ter;
24     }data;
25 } no;
26
27 #endif
28
```

Estrutura:

- Optámos por uma estrutura de árvore simples de árvore com níveis em lista, dividimos os nós em dois tipos o *no\_terminal* e o *no\_multi*, associando os mesmos a uma outra estrutura intitulada *no* que através do *union* permite ter um pointer ou para um *no\_terminal* ou para o *no\_multi*.
- Para inserir e manipular esta árvore criámos as seguintes funções:
  - *no\** getNext(*no\** n) → Devolve o *no* no field next;

- `int count_next(no* n)` → Conta o numero do irmãos do `no n`;
- `no* create_id(char* type)` → cria o `no` terminal do `tid`;
- `no* create_id(char* type)` → cria o `no_terminal` do tipo `id`;
- `no* create_char(char* type)` → cria o `no_terminal` do tipo `char`;
- `no* create_reallit(char* type)` → cria o `no_terminal` do tipo `realint`;
- `no* create_intlit(char* type)` → cria o `no_terminal` do tipo `intlit`;
- `no* create_type(char* type)` → cria o `no_terminal` do tipo `type`;
- `no* create_multi(char* type)` → cria o `no_multi`;
- `void add_son_ini(no* n, no* node)` → adiciona o nó `node` como primeiro filho do `node n`;
- `void add_son_end(no* n, no* node)` → adiciona o nó `node` como ultimo filho do `node n`;
- `void add_sibling(no* n, no* node)` → adiciona o nó `node` como irmão do `node n`;
- `void showTree(int level, no* n)` → Dá *print* de toda a árvore;
- `void freeMemory(no* n)` → liberta a memória ocupada pela árvore;
- `void fixVarSpec(no* n, no* t)` → função implementada para simplificar casos de múltiplas variáveis defenidas em simultâneo que permite adicionar a todos os irmãos do *nó* `n` o nó `t` como ;

Alterações na gramática:

- Tanto nos casos de 0 ou mais vezes como nos opcionais optámos por criar regras de gramática separadas. Por exemplo:
  - No varspec ficou
    - VarSpec :IDENTIFIER VarspecRepeat Type
    - VarspecRepeat:VarspecRepeat COMMA IDENTIFIER

Para resolver ambiguidades definimos as seguintes precedências com prioridade de cima para baixo:

Operador	Associatividade
ASSIGN	Right
OR	Left
AND	Left
XOR	Left
GT GE LT LE EQ NE	Left
PLUS MINUS	Left
STAR DIV MOD	Left
NOT	Right

### Meta 3: algoritmos e estruturas de dados da AST e da tabela de símbolos

#### Alterações estrutura árvore:

- Foi adicionado a linha, coluna e type aos parâmetros da estrutura *no\_terminal* e *no\_multi*, ainda se adicionou o parâmetro *error* à estrutura *no* e o array *params* também ao *no\_multi*;
- Também foram atualizadas todas as funções de modo a se adaptarem às novas estruturas;
- Para além disso foram efectuadas pequenas alterações no yacc e lex de modo a se conseguir dar a coluna e linhas as funções referidas no ponto anterior.

#### Estrutura tabela:

- Optámos por uma estrutura semelhante à árvore, tendo uma tabela global que conecta não só a uma lista de elementos mas também a uma lista de tabelas e ainda possui um ponteiro para outra tabela global.
- Para trabalhar com esta estrutura criámos as seguintes funções:
  - *insert\_local()* → insere na *local\_table*, um elemento com as características dadas como parâmetros;
  - *insert\_global()* → insere na *global\_table*, uma tabela global, devolve a tabela criada;

- `insert_elem_local()` → insere na *table2*, um elemento com as características dadas como parâmetros;
- `show_table()` → imprime as várias tabelas globais recorrendo a `show_global_elem()` e ao `show_local()` para apresentar os elementos e tabelas das mesmas, respectivamente;
- `print_type()` → imprime o type correspondente ao *int t*;
- `search_el()` → procura o elemento com o nome *str* nas tabelas local e global e devolvem-no;
- `search_el2()`, `search_el3()` → procuram o elemento com o nome *str* nas tabelas local e global, respectivamente e devolvem-no;

#### Identificação dos erros:

- De modo a percorrer a árvore o mínimo possível dividimos a análise de erros em duas fases:
  - Encontrar os erros, definir tipos e adicionar valores à tabela : nesta fase percorre-se toda a árvore, verificando se existe erro ou não, definindo o tipo e adicionando à tabela os valores necessários, caso se encontre erro definir o valor *error* adicionado a estrutura de *no* para 1;
  - Print dos erros: voltar a correr a árvore e verificar o *error* da estrutura *no* caso em que se encontre em 1 imprime-se o *error*

correspondente ao nó. Ainda nesta fase, faz-se a verificação se a variável é ou não usada através do valor *used* da estrutura do elemento (*table\_elem*);

### Meta 4: geração de código

Para obter os prints do modo que o professor pediu foi necessário a inicialização de todos os modos de print e ainda uma função que dá print ao *bool*, todos estas inicializadas no início do código.

Também foi implementada uma função que percorre toda a árvore e inicializa todas as *strings* necessárias para a execução do programa.

De seguida foram implementadas todas as outras funções, destacando-se as seguintes decisões:

- Para poder implementar cálculos com *float* tivemos de passar todos os *float* primeiro para *double* e depois de efectuar as alterações passar o valor para *float* novamente. Também escolhemos transformar o *float* em *double* antes do print do programa;
- No nosso if optámos pelos seguintes labels: *if%d*, *else%d* e *cont%d*, em que *%d* é o valor guardado na variável global *ifNum*;
- Para o for optámos pelas labels seguintes: *for%d*, *forfunc%d*, e *contfor%d*, *forNum*.

- Durante a implementação optámos por usar o nome `get%s` e `gen%s` em que `%s` é o nome do nó que está a ser analisado (ex: `getVarDecl`, `genNode`) também usamos o nome `gen%s` ;
- De seguida irá estar uma explicação resumida de todas as funções mais importantes:
  - `void genCode(no* tree)` → função inicial, que é chamada pelo `gocompiler.y`, e nesta função que se dá print às variáveis globais e chama-se a função para mostrar o resto;
  - `void printTipo(int n)` → dá print do tipo correspondente ao tipo `n`, ex(`n=1` dá print a `i2`);
  - `void genFuncDecl(no* no)` → dá print do header da função, no caso de a função ser a `main` assumir tipo de return `i32( int )`;
  - `void genStr(no* n)`; → função que percorre toda a árvore, caso encontre um nó do tipo `str` recorre a função *strprint*;
  - `void setStored(no* n, char* name)`; → altera o valor `stored` para 1 em todos os `ids` com o nome `name`;
  - `void strprint(no* n)`; → recebe o nó correspondente à string e dá print de acordo com o formato da linguagem `llmv`;