**Universidade de Coimbra**
**Faculdade de Ciências e Tecnologia**
**Licenciatura em Engenharia Informática**


FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

# Relatório Projeto Compiladores 2021/22

Compilador para a linguagem DeiGo
Professor: Alexandre Jesus

**Realizado por:**

- Sofia Santos Neves **N.º:** 2019220082 **Email:** sofianeves@student.dei.uc.pt
- Tatiana Silva Almeida **N.º:** 2019219581 **Email:** tsalmeida@student.dei.uc.pt

# Índice

# 1. Gramática

## 1.1. Notação EBNF

Durante a alteração da gramática EBNF tomámos algumas decisões para que a mesma aceita-se listas de produções, nomeadamente:

1. Quanto aos casos em que a produção era opcional. Como por exemplo:

   VarsAndStatements ⟶ VarsAndStatements [VarDeclaration | Statement] SEMICOLON | ϵ

   Aqui, optámos por criar duas produções:

```
VarsAndStatements: /* epsilon */
                 |  SEMICOLON VarsAndStatements
                 |  VarDeclaration SEMICOLON VarsAndStatements
                 |  Statement SEMICOLON VarsAndStatements
                 ;
```

2. Nos casos onde a produção podia ter zero ou mais repetições. Por exemplo:

   VarSpec ⟶ ID {COMMA ID} Type

   Decidiu-se criar uma nova produção como no seguinte exemplo:

```
VarSpec: ID VarSpecs Type
       ;
VarSpecs: COMMA ID VarSpecs
        | /* epsilon */
        ;
```

Em ambos os casos anteriores, adotou-se a estratégia de utilizar a recursividade à direita de modo a se conseguir obter uma lista de tamanho variável.

## 1.2. Precedência

Para a reescrita da gramática foi necessário tomarmos algumas decisões a fim de resolver conflitos. Para isso definimos precedências:

| Precedência | Operador | Associatividade |
|---|---|---|
| 1 | UNARY | %nonassoc |
| 2 | ELSE | %nonassoc |

| | | |
|---|---|---|
| 3 | LPAR, RPAR, LSQ, RSQ, LBRACE, RBRACE | %left |
| 4 | NOT | %right |
| 5 | STAR, DIV, MOD | %left |
| 6 | PLUS, MINUS | %left |
| 7 | EQ, NE, GT, GE, LT, LE | %left |
| 8 | AND | %left |
| 9 | OR | %left |
| 10 | ASSIGN | %right |
| 11 | COMMA | %left |

# 2. Algoritmos e estruturas de dados da AST e da tabela de símbolos

## 2.1. Estruturas de dados da AST

Para criar a AST usámos listas ligadas simples. Cada nó pode, ou não, ter um filho, também, pode ter zero ou mais irmãos.

Cada estrutura **Node** usou os seguintes tipos de dados que representam:

- **type:** o nome, ex: ParamDecl, Block, If, entre outros;
- **data:** uma estrutura onde fica guardado o valor, linha e coluna;
- **child:** o nó filho na árvore;
- **nextSibling:** o nó irmão na árvore;
- **annotation:** o tipo do nó, ex: Int, String, etc;
- **param_list:** os parâmetros de uma função, inicializado no ''Call'';
- **is_expr:** um booleano para saber que ID's são da produção expressão(Expr);
- **is_valid:** um booleano para verificar se uma dada função é válida.

```
typedef struct _Node Node;
struct _Node{
    // Meta 3
    Type annotation;
    param *param_list;
    bool is_expr, is_valid;

    // Meta 2
    Type type;
    Data data;
    Node *child;
    Node *nextSibling;
};
```

Cada estrutura **Data** usou os seguintes tipos de dados que representam:

- **line:** a linha;
- **column:** a coluna;
- **value:** o conteúdo, ex: o Id tem o valor 1.

```
typedef struct Data{
    int line;
    int column;
    char *value;
}Data;
```

## 2.2. Algoritmos da AST

Para a construção da AST criámos os nós ao longo do yacc, usando posteriormente alguns algoritmos para a percorrer, como nas funções de **printAST** e **freeNode**.

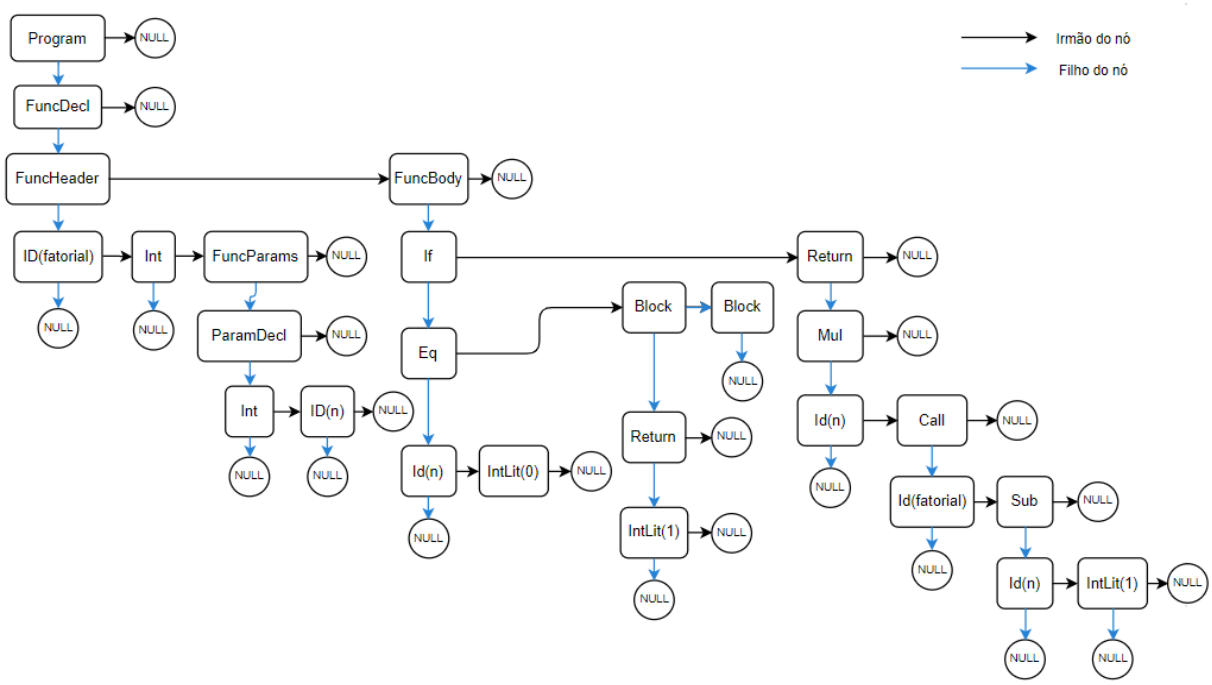
Para estas mesmas funções, a travessia pelos nós da árvore é iniciada pelo primeiro nó - Program -, continuando depois recursivamente para os filhos e, posteriormente, para os irmãos.

```
void printAST(Node *node, int depth, int semantic_analysis) {
  printNode(node, depth, semantic_analysis);
  if (node->child) {
    printAST(node->child, depth + 1, semantic_analysis);
  }
  if (node->nextSibling) {
    printAST(node->nextSibling, depth, semantic_analysis);
  }
}
```

Para o exemplo abaixo, segue-se a sua representação da AST:

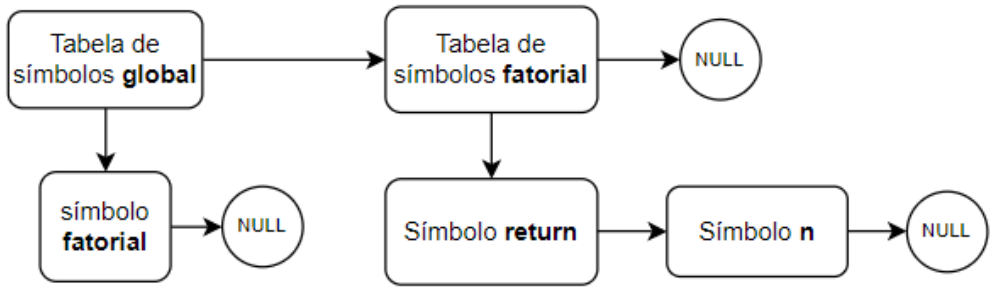
```
package main;

func factorial(n int) int {
    if n == 0 {
        return 1;
    };
    return n * factorial(n-1);
};
```




## 2.3. Tabela de símbolos

A construção da tabela de símbolos passa pela travessia da árvore. Essa tabela foi construída, recursivamente, como sendo uma lista ligada de tabelas(sym_table), onde a primeira tabela corresponde à global e cada uma das tabelas seguintes correspondem a funções válidas. Cada uma das tabelas tem também associada uma lista de símbolos, como representado na seguinte figura:




Cada estrutura **sym_table** usou os seguintes tipos de dados que representam:

- **name:** o nome da tabela (data->value);
- **symbol_list:** uma lista de símbolos (symbol) dessa mesma tabela;
- **next:** um ponteiro para a próxima tabela.

```
typedef struct _sym_table sym_table;
struct _sym_table{
    char *name;
    symbol *symbol_list;
    sym_table *next;
};
```

Cada estrutura **symbol** usou os seguintes tipos de dados que representam:

- **is:** se o símbolo é uma função, variável, parâmetro ou um return;
- **name:** o conteúdo do nó, ex: o Id tem o valor 1;
- **type:** o tipo do nó, ex: Int, String, entre outros;
- **param_list:** os parâmetros do símbolo;
- **next:** um ponteiro para o próximo símbolo da tabela;
- **used:** com um booleano, se o símbolo foi usado ou não;
- **line:** a linha;
- **column:** a coluna.

```
typedef struct _symbol symbol;
struct _symbol{
    enum
    {
        func,       //1
        var,        //2
        parameter,  //3
        return_,
    } is;


    char *name;
    Type type;
    param *param_list;
    symbol *next;

    // error declared but never used
    bool used;
    int line, column;
};
```

Cada estrutura **param** usou os seguintes tipos de dados que representa:

- **id:** o nome da variável;
- **type:** o tipo do parâmetro, ex: Int, String, etc;
- **next:** um ponteiro para o próximo parâmetro da função;

```
typedef struct _parameters param;
struct _parameters{
    char *id;
    Type type;
    param *next;
};
```