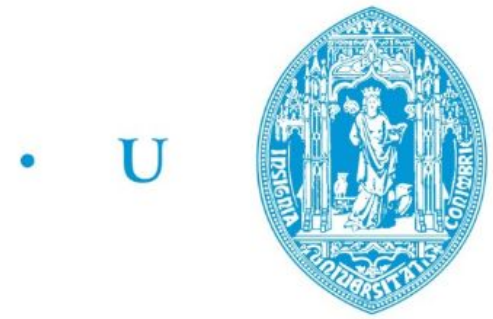

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

# Compilador para a linguagem Juc

**Projeto de Compiladores 2019/20**

António Marques Maria - 2017265346
David Jesus Vaz Cortesão Silva – 2008109004

# Introdução

No âmbito da unidade curricular de Compiladores inserida na Licenciatura em Engenharia Informática da Universidade de Coimbra, iremos implementar um compilador para uma linguagem baseada em Java, nomeadamente Juc.

Neste relatório iremos apresentar como implementámos a gramática descrita no enunciado do problema e que opções tomámos para o fazer, tal como as estruturas de dados e algoritmos usados para a construção da AST e da tabela de símbolos.

Assim de forma a implementar esta programa recorremos ao Lex, um analisador léxico, e ao Yacc, que é um analisador sintático, de forma a implementar a gramática e verificar a sua correta derivação. Já para programar todas as funções utilizadas na construção da Árvore de sintaxe abstrata (AST), utilizamos a linguagem C.

# Analisador Sintático

A gramática fornecida, era ambígua como tal tivemos de recorrer a novas produções para os opcionais e algumas produções que se repetissem 0 ou mais vezes para proceder a uma recursão à esquerda. Devido à possibilidade de haver 0 ou mais repetições criamos irmãos na árvore onde são tratados os casos graças a produções opcionais:

- Program: CLASS ID LBRACE Program_Aux RBRACE | CLASS ID LBRACE RBRACE
- FieldDecl: PUBLIC STATIC Type ID FieldDecl_aux SEMICOLON
- MethodHeader: Type ID LPAR FormalParams RPAR | Type ID LPAR RPAR | VOID ID LPAR FormalParams RPAR | VOID ID LPAR RPAR
    - Aqui sendo a parte de FormalParams opcional criámos gramática para cada um dos casos
- FormalParams: Type ID FormalParams_Aux | STRING LSQ RSQ ID| Type ID
- MethodBody: LBRACE MethodBody_Aux RBRACE
- VarDecl: Type ID VarDecl_Aux SEMICOLON
- MethodInvocation: ID LPAR Expr MethodInvocation_Aux RPAR | ID LPAR RPAR | ID LPAR error RPAR

Quanto aos auxiliares dos anunciados acima, pode haver um conjunto vazio, representado em comentário por /*epsilon*/, ou uma combinação de cada um dos tokens necessários:

- Program_Aux: Program_Aux MethodDecl | Program_Aux FieldDecl | Program_Aux SEMICOLON | FieldDecl | MethodDecl | SEMICOLON
    - neste caso apenas chegamos aqui caso no Program a parte da declaração não seja 0, podendo repetir ou fazer reduce
- FormalParams_Aux:COMMA Type ID | COMMA Type ID FormalParams_Aux
    - Aqui pode repetir ou fazer reduce, esta produção é criada para FormalParams
- FieldDecl_aux: FieldDecl_aux COMMA ID | /* empty */, produção criada em FieldDecl podendo admitir 0 ou mais vezes
- MethodBody_Aux: MethodBody_Aux Statement | MethodBody_Aux VarDecl | /* empty */
    - Esta produção é criada em MethodBody como 0 ou mais vezes
- VarDecl_Aux: VarDecl_Aux COMMA ID | /* empty */
    - Esta produção é criada em VarDecl como 0 ou mais vezes
- Statement_Aux: Statement_Aux Statement | Statement
    - Esta produção auxiliar é criada em Statement para os casos de 0 ou mais vezes
- MethodInvocation_Aux: MethodInvocation_Aux COMMA Expr | /* empty */
    - Es produção tem como Expr opcional e uma lista de 0 ou mais vezes COMMA Expr;

Criámos ainda Expr: Assignment | Expr_Aux onde Expr_aux é a nossa gramática de Expr para conseguirmos evitar shift/reduce e contornar o problema de termos mais do que uma variável.

No âmbito das precedências, de modo a obter para cada string uma e uma só árvore de derivação possível, adicionamos as seguintes precedências, ordenadas por prioridade decrescente e precedência crescente:

```
%nonassoc NO_ELSE
%nonassoc ELSE
%left COMMA
%right ASSIGN
%left OR
%left AND
%left LT GT EQ NE LE GE
%left PLUS MINUS
%left STAR DIV MOD
%right NOT
%right precedencia
%nonassoc preced
```

Com associatividade à esquerda devido à recursão também à esquerda na árvore. O resto da gramática manteve-se igual à que nos foi fornecida no enunciado.

# Árvore AST

A estrutura de dados da AST é possui as estruturas base para a elaboração da mesma, como a criação de nós, ASTtree* createNode(char* type, char* value), que tem como parâmetro um Type (nome do nó) e um value (se tiver um valor associado como Declit ou Strlit por exemplo).

Adição de um nó filho ou nó irmão, que percorrem todos os nós filho ou irmão de um determinado nó, e adicionam o nó pretendido ao final da lista de nós que existe nessa categoria e que estão associados ao nó “node” que é passado como parâmetro (adiciona filho ou irmão passado por parâmetro ao nó passado por parâmetro também), sendo estes métodos appendChild(ASTtree* child, ASTtree* node) e appendBrother(ASTtree* brother, ASTtree* node).

O método de print na árvore, imprime em Post order. Com o número de pontos requeridos no enunciado e ignorando alguns nós nulos criados ao longo da árvore.

# Tabela de Símbolos

O método ast_to_symbol_table comeca por percorrer a AST à procura de 3 nós em particular, “FieldDecl”, “MethodDecl” e “VarDecl” dentro do “MethodBody”, para criar a tabela Main (começa com FieldDecl), tabela das funções (começa com MethodDecl) e tabela de variáveis em cada função (VarDecl).

À medida que percorre os vários nós da arvore AST com os vários métodos, - create_symbol, add_sym_to_table, create_table, add_table- , junta um par Id-value-param que será introduzido na tabela de símbolos. A tabela de símbolos é composta por um head(nome da tabela), lista de simbolos e parametros da função.

Os tabs na tabela são introduzidos no método de print e não associados já aos nós, a possibilidade de haver um parâmetro ou não na função é associado ao sym_table_node, nomeadamente à variável flag, a que depois é associado o tab no print, como é feito anteriormente.

# Conclusão

Através da realização deste trabalho, foi-nos possível aprofundar conhecimento nomeadamente no que diz respeito a como ocorre a análise lexical e sintática “dentro” de um compilador, e como se pode criar várias linguagens diferentes através da sua gramática e dos tokens que são aceites por essa mesma linguagem. Também ajudou a compreender como é que ocorre a derivação de uma gramática e como resolver vários problemas em linguagens LALR(1), especificamente os problemas shift/reduce e reduce/reduce.