Projeto de Compiladores 2020/2021

Compilador para a linguagem C

Estudantes:
Dinis Carvalho, nº 2018278118
João Teixeira, nº 2018278532

Introdução

Neste projeto foi nos solicitado o desenvolvimento de um compilador para a linguagem UC, um subconjunto da linguagem C (de acordo com o standard C99).

Este trabalho foi subdividido em quatro partes: Analisador Lexical; Analisador Sintático; Analisador Semântico; Geração de código intermédio. As mesmas correspondem às metas 1 a 4, respetivamente.

Este relatório tem duas secções, *gramática reescrita* e *algoritmos e estruturas de dados da AST e da tabela de símbolos*. A última secção *geração de código* não será apresentada visto não termos realizado essa meta.

Secção 1. Gramática reescrita

Um dos nossos maiores problemas na execução da Meta 2 - Analisador Semântico, foi a ambiguidade da gramática. Tivemos, portanto, de alterar a gramática para resolver as ditas ambiguidades (tendo o cuidado de não alterar a linguagem resultante) (Fig. 1 e Fig. 2).

```
Program: FunctionsAndDeclarations
;
```

```
FunctionsAndDeclarations: FunctionDefinition FunctionsAndDeclarationsOP
        | FunctionDeclaration FunctionsAndDeclarationsOP
        | Declaration FunctionsAndDeclarationsOP
;
```

```
FunctionsAndDeclarationsOP: FunctionsAndDeclarations
        |
;
```

```
FunctionDefinition: TypeSpec FunctionDeclarator FunctionBody
;
```

```
FunctionBody: LBRACE DeclarationsAndStatementsOp RBRACE
;
```

```
DeclarationsAndStatementsOp: DeclarationsAndStatements
        |
;
```

```
DeclarationsAndStatements: DeclarationsAndStatements Statement
        | DeclarationsAndStatements Declaration
        | Statement
        | Declaration
;
```

```
FunctionDeclaration: TypeSpec FunctionDeclarator SEMI
;
```

```
FunctionDeclarator: ID LPAR ParameterList RPAR
;
```

```
ParameterList: ParameterDeclaration ParameterListOp
;
```

```
ParameterListOp: COMMA ParameterDeclaration ParameterListOp
        |
;
```

```
ParameterDeclaration: TypeSpec
        | TypeSpec ID
;
```

```
Declaration: TypeSpec Declarator DeclarationOp SEMI
        | error SEMI
;
```

```
TypeSpec: CHAR
        | INT
        | VOID
        | SHORT
        | DOUBLE
;
```

```
Declarator: ID
        | ID ASSIGN Expr
;
```

```
StatementAux: StatementError
        | StatementError StatementAux
;
```

```
StatementError: Statement
        | error SEMI
;
```

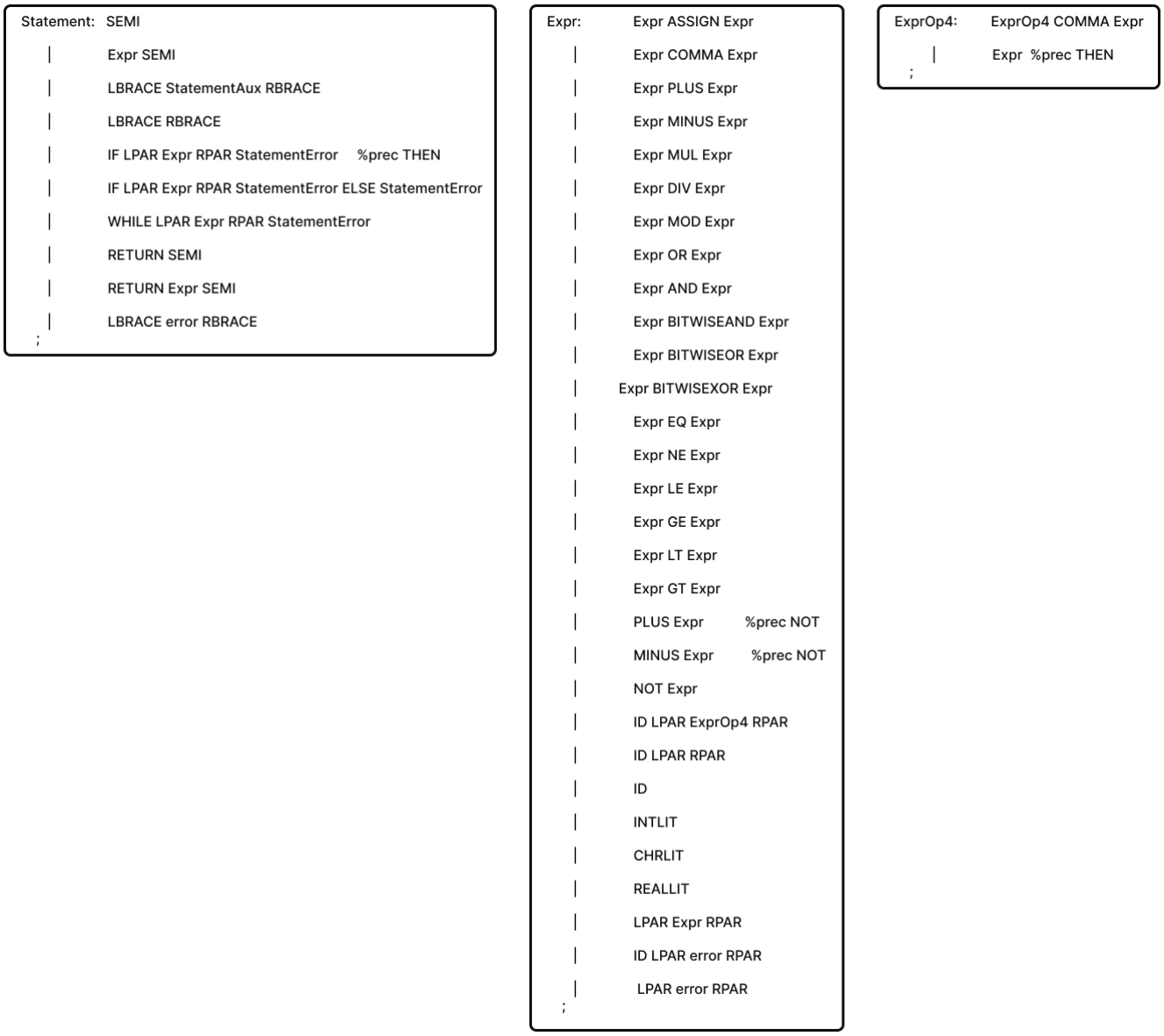

Fig. 1 e Fig. 2: Gramática

Utilizamos, portanto, os símbolos %left, %right e %nonassoc (Fig. 3), para especificar a precedência de operações à esquerda, à direita ou a usa inexistência, onde a última definição listada tem a maior precedência. Para isto recorremos à documentação do C. Para resolver algumas ambiguidades da gramática utilizámos o "comando" %prec, de forma a atribuir uma determinada precedência a regras específicas da gramática onde se encontravam ambiguidades. Um exemplo disto é no "*IF LPAR Expr RPAR StatementError %prec THEN*", onde *THEN* é um símbolo sem associatividade. Deste modo, asseguramos que *"x if y if z"* é um erro sintático.

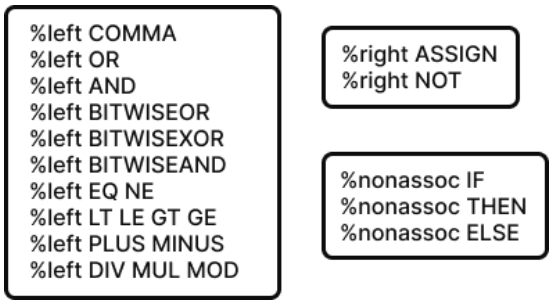

Fig. 3: %left; %right e %nonassoc

No caso dos ciclos criamos expressões auxiliares, como por exemplo, em "*Statement*" temos "*LBRACE StatementAux RBRACE*" que subdivide em "*StatementError*" ou "*StatementError StatementAux*" (Fig. 4).

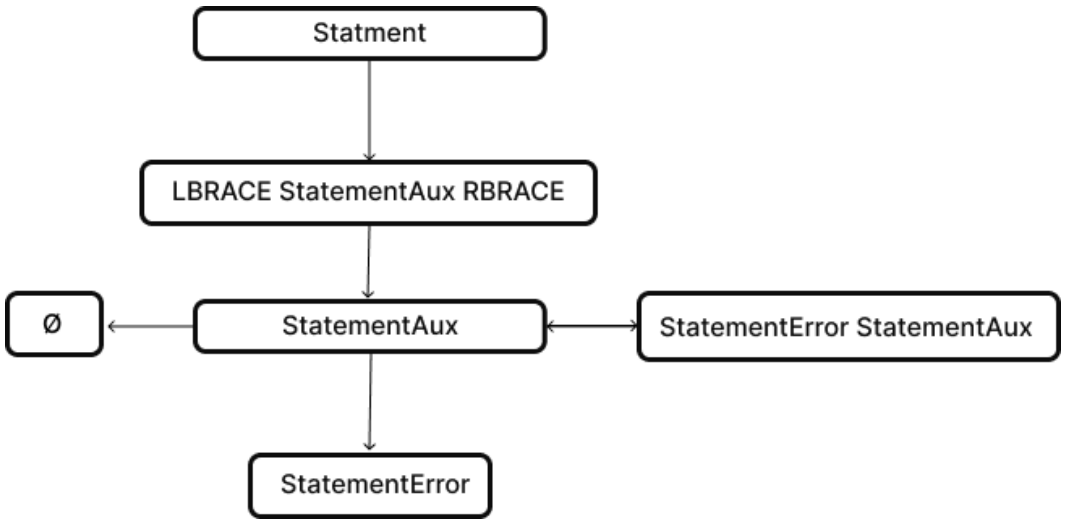

Fig. 4: Ciclo *"Statement Aux"*

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

A estrutura de dados AST é composto por uma lista ligada onde cada nó está ligado ao seu pai, irmãos e filhos. Enquanto o seu valor e tipo são definidos na criação de num novo nó, múltiplos parâmetros, tais como, número de filhos, o nó do pai e o nó do irmão, são iniciados a NULL. Além disse, o node tem uma *array list* com os nodes dos seus filhos.

Ao longo do programa, duas funções, "*addchild*" e "*addbro*" são chamadas para completar a árvores com nova informação.

De modo a conseguir identificar na tabela de símbolos o seu tipo, criamos um parâmetro no nó denominado de "*note*" e uma nova função "*annotate*" que percorre a AST recursivamente modificando a anotação correspondente. Na maioria dos casos é suficiente buscar o seu "*type*" e alterar o "*note*", no entanto, isto não acontece para certos "*ID's*" onde temos de procurar a variável na tabela de função correspondente. Caso não seja encontrada, temos de procurar na tabela global, com a função "*findTableAndAnnotate*".

Por sua vez, esta função faz uso da estrutura da tabela de símbolos (Fig. 5 e Fig. 6).

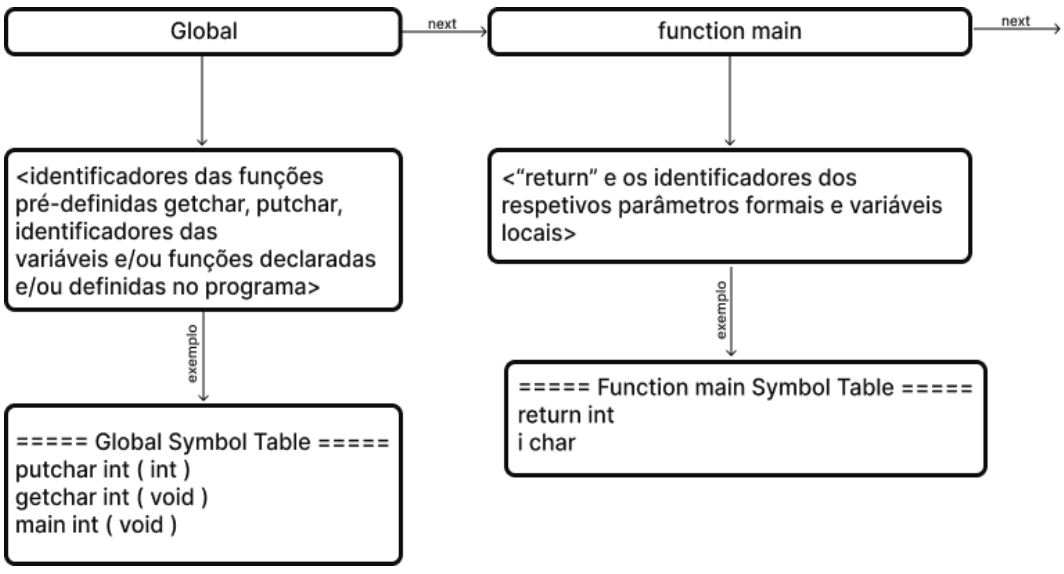

Fig. 5: Estrutura de dados da tabela de símbolos

Temos dois tipos de nós (Fig. 6), *nodetg* e *nodetf*, num formato de lista de listas. Os *nodetg* são a lista principal (ligam as diferentes tabelas), e os *nodetf* representam o conteúdo das tabelas. Tomando como exemplo a Fig. 3, o "Global", o "function main" e outras funções que viriam com o next seriam do tipo "*notetg*" enquanto os seus conteúdos seriam do tipo "*nodetf*".

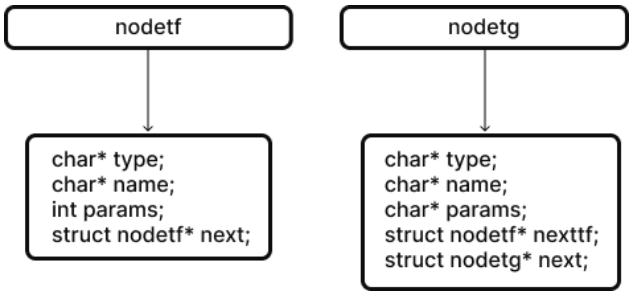

Fig. 6: Estrutura dos nodes global e função

A tabela de símbolos é gerada a partir da AST, onde cada vez que encontramos uma função ou uma variável adicionamos à tabela. Nesta operação verificamos também se já se encontra inserida de modo a não ter variáveis globais ou funções repetidas.

Nota:

Gostaríamos de adicionar que, por equivoco, não colocamos os nossos nomes na submissão da Meta 2 - Analisador Semântico no Mooshak, um dos requisitos estabelecidos no enunciado do projeto: “O ficheiro lex entregue deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada membro do grupo.”.