

Departamento de Engenharia Informática

Relatório do Projeto

**Compilador para a linguagem deiGo**

**Compiladores (COMP)**

Marco Pais Nº 2019218981

Tiago Oliveira Nº 2019219068

**Reescrever a Gramática**

Foi nos fornecida, inicialmente, uma gramática da linguagem em notação EBNF, sendo esta ambígua, necessitando assim de várias modificações para tornar possível implementar, corretamente, a linguagem na ferramenta YACC. As ambiguidades inicialmente existentes foram identificadas através da identificação de diversos conflitos manifestados na execução do programa, sendo estes conflitos “*shift-reduce*” e “*reduce-reduce*”.

A primeira fonte de ambiguidades a ser identificada e corrigida foi encontrada nas produções relacionadas com expressões (**Expr**), sendo necessário adicionar regras de determinação de associatividade e de precedência de operadores. As regras de associatividade podem ser obtidas no YACC através do uso dos comandos **%left**, **%right** e **%nonassoc**. Já para a precedência foi necessário organizar as declarações de associatividade, pois a precedência aumenta de baixo para cima.

Tendo, ainda, em conta a precedência de operadores da linguagem foi ainda necessário acrescentar uma regra relativa às *if-else clauses*, que não possuem regras de associatividade definidas. No YACC isso pode ser obtido através do comando **%nonassoc**.

A imagem seguinte é referente às regras de precedência implementadas.

Uma imagem com texto

Descrição gerada automaticamente

De seguida, foram efetuadas várias modificações na gramática relativas:

* À implementação de ciclos que produzem listas de declarações, variáveis e/ou expressões. Para isso foram implementadas produções recursivas.
* À implementação de regras que identificam e lidam com casos em que uma dada construção da gramática é opcional.
* À criação de uma nova produção (**IDaux**), devido à necessidade adicionar, algumas vezes, o *token* **ID** à Árvore de Sintaxe Abstrata (AST).

Por fim, foram utilizados comandos do YACC, **%destructor**, que permitem limpar a *stack* caso existam erros sintáticos na execução do programa.

# **Algoritmos e Estruturas de dados**

No desenvolvimento das metas 2 e 3 foram implementados, maioritariamente, diversos algoritmos que permitem a pesquisa de elementos nas diversas estruturas de dados e/ou permitem percorrer essas mesmas estruturas de dados.

As estruturas de dados implementadas foram árvores, listas ligadas e alguns vetores.

A primeira estrutura a ser implementada foi a Árvore de Sintaxe Abstrata (AST). Esta estrutura tem nós que são criados assim que o YACC encontra uma sequência de carateres que corresponde a uma regra definida na gramática. A estrutura guarda o valor do *token* lido numa variável (char \* valor), nos casos em que esse valor exista (STRLIT, INTLIT …). É ainda guardado o tipo do *token* lido para mostrar ao imprimir a AST. Para o desenvolvimento da Meta 3 foi ainda necessário guardar na estrutura as linhas e colunas em que o *token* aparece, para mostrar nos casos em que há erros de semântica, e ainda um **char \* nota** que guarda a anotação feita, correspondente ao tipo de variável (*int*, *float32* …).

A imagem seguinte representa a estrutura usada para criar a AST, contendo todas as informações referidas anteriormente.

Uma imagem com texto

Descrição gerada automaticamente

Nesta estrutura considera-se um nó “irmão” aquele que se encontra no mesmo nível do nó acedido. Por sua vez, um nó “filho” é aquele que se encontra num nível inferior ao nó em questão.

A tabela de símbolos, implementada através de 3 listas ligadas, sendo a estrutura relativa à tabela de símbolos globais (variáveis globais e definições de funções) a primeira tabela a ser criada. Nesta primeira lista os dados (*globalTable*) foram guardados os nomes das funções (char \*name), o tipo da função (char \*type), duas flags utilizadas para verificar se estávamos a guardar uma variável global ou uma função e uma para verificar se já estavam declaradas (*int func* e *int declared*, respetivamente). Esta lista possuía ainda um ponteiro para cada uma das outras 2 listas ligadas.

As outras 2 listas ligadas têm uma estrutura idêntica à anterior, com algumas alterações, pois as estruturas dizem respeito a uma lista ligada que guarda todas as variáveis definidas dentro de uma função e a outra todos os parâmetros definidos na declaração da função.

Por fim, foi ainda criada uma lista ligada para guardar todos os erros de semântica que podiam ser gerados e os mesmos serem impressos por ordem crescente de linha e coluna, podendo esta ordem ser alterada no código consoante a ordem de apresentação pretendida.

Uma imagem com texto

Descrição gerada automaticamente

# **Geração de código**

Nesta etapa pouco foi feito, tendo a mesma ficado incompleta e apenas funcionando para o caso de teste ***factorial.dgo***.

# **Bibliografia**

<https://go.dev/ref/spec>