· U C · Compiladores
2021/2022

João António da Silva Melo | 2019216747

Miguel António Gabriel de Almeida Faria | 2019216809

Relatório

Projeto de Compiladores - GoCompiler

Gramática re-escrita

De modo a remover problemas de ambiguidade, durante a escrita da gramática da linguagem foram tomadas algumas decisões de maneira a ser possível a sua implementação no *yacc* sem conflitos de *shift-reduce* e *reduce-reduce*.

- O primeiro método utilizado para remover a ambiguidade foi a implementação da associatividade e precedência em alguns operadores, de acordo com a linguagem Go. No *yacc* a associatividade é implementada com recurso às keywords "%left" e "%right", sendo que, por exemplo, o operador '=' (igual) terá associatividade à direita, a = b = c => a = (b = c). Já o operador '+' (soma) terá associatividade à esquerda, a + b + c => (a+b)+c. A precedência no *yacc* é dada de acordo com a ordem das declarações de associatividade: a primeira declaração será a com menor precedência. Assim, para seguir as regras de Go, foi, por exemplo, declarada primeiro a associatividade do operador '+' que a do operador '*' (multiplicação). Também foi utilizada a keyword %prec que permite atribuir a uma regra da gramática a precedência associada ao operador especificado. Assim, utilizámos a precedência de '!'(negação) (a mais elevada) nas seguintes regras:

```
| MINUS Expr %prec NOT
| PLUS Expr  %prec NOT
```

- Na escrita da nossa gramática utilizamos recursividade à direita, pois nos casos em que existe "zero ou mais repetições" conseguimos garantir mais facilmente as regras da gramática.
  - Por exemplo, na criação de ciclos que permitam produzir listas de declarações, variáveis, parâmetros, expressões.

- A estrutura *union* que usamos aceita apenas 2 tipos de dados: "infoT", que é destinada a *tokens* em que é necessário guardar algumas das suas informações, e "node", que é utilizada nos *types* e que vai ser usado como cada nó da árvore. Estas estruturas vão ser abordadas novamente.

Algoritmos e estruturas de dados

Árvore de Sintaxe Abstrata

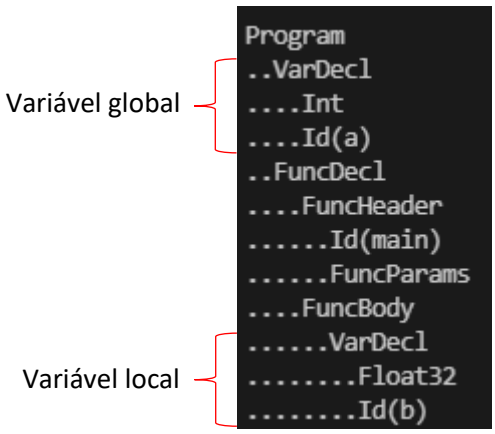
Estrutura "infoToken": estrutura que permite guardar o "value" de um token, recebido por *yytext* ("id", "intlit", "realit", …), bem como a informação acerca da linha e da coluna em que se encontra ("num_linha" e "num_coluna").

```
typedef struct infoToken{
    char *value;
    int num_linha;
    int num_coluna;
}infoToken;
```

Estrutura "nodeAst": estrutura que representa cada nó da árvore, possuindo o tipo de dado ("type") desse nó recebido pelo *return* ("Id", "IntLit", Mul", "If", "Int", "VarDecl", "ParamDecl", …), a estrutura "infoT" já mencionada, o tipo "exprNotation" que vai ser especificado na árvore ("int", "float32", "undef", …), o nó filho ("child") e o nó irmão ("sibling").

```
typedef struct nodeAST {
    char *type;
    struct infoToken *infoT;
    char *exprNotation;
    struct nodeAST *child;
    struct nodeAST *sibling;
}nodeAst;
```

Para implementar a árvore de sintaxe abstrata, foi utilizada a struct "nodeAst" para facilitar a sua construção, usufruindo dos elementos "child" e "sibling" de um nó para permitir a ligação entre todos os nós. Assim, definimos que haveria uma hierarquia consoante a sua especificidade na gramática. Por exemplo, a declaração de uma variável local será de uma hierarquia mais baixa do que a de uma variável global:



De uma maneira geral, o nosso algoritmo baseia-se no seguinte diagrama:

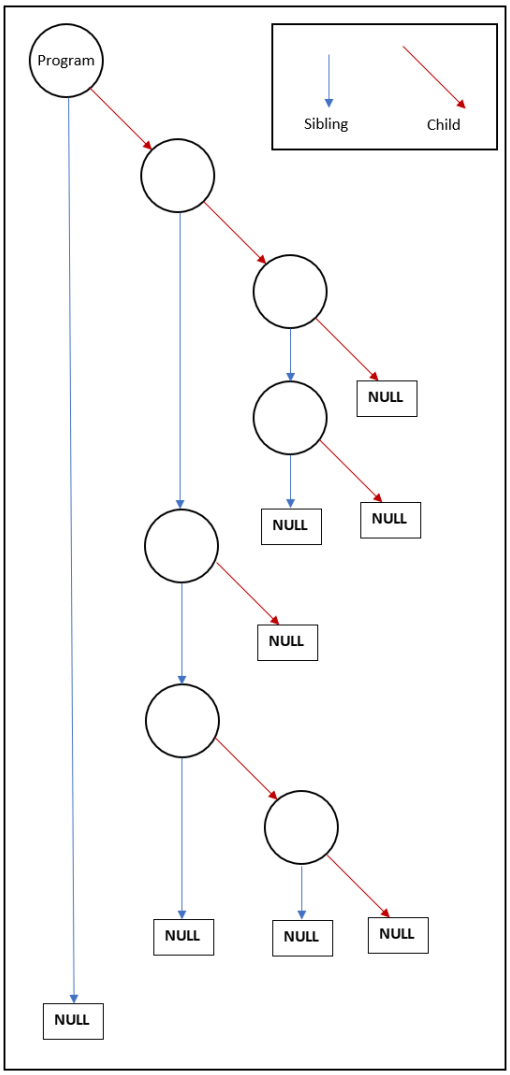


Tabela de Símbolos

Estrutura "elementST": estrutura que representa cada elemento da tabela, possuindo o nome do elemento ("name") (que pode ser variável ou função), os parâmetros ("paramTypes") caso seja uma função, o tipo da variável ou função ("type"), uma variável auxiliar para o print da tabela "param", uma variável auxiliar "isVar" que permite saber se o elemento é função ou variável, o elemento "body" que é usado caso se encontre uma função, o elemento "nextElem" que indica a nova variável (global) ou função, a variável "alreadyUsed" que indica se as variáveis (locais) já foram usadas e o número da linha e coluna ("line" e "col").

```
typedef struct elementST {
    char *name;
    char *paramTypes;
    char *type;
    char *param;
    int isVar;
    struct elementST *body;
    struct elementST *nextElem;
    int line;
    int col;
    int alreadyUsed;
}elementST;
```

De modo a garantir que uma variável não deve ser redefinida e que estando definida, deve ser utilizada, são usados algoritmos de travessia de listas que verificam se o elemento já se encontra na tabela e para mudar nele a variável "alreadyUsed" caso já tenha sido usado.

Para criar a AST anotada, utilizamos um algoritmo de recursão de cabeça para ser mais fácil e eficiente a anotação dos nós, dado que, com esta recursão, nos casos em que é necessário saber a anotação dos nós filho, podemos analisar diretamente o valor que estes possuem e atribuir um valor de acordo ao nó em questão.

O seguinte esquema representa o algoritmo de criação da tabela: