Faculdade de Ciências e Tecnologias da Universidade de Coimbra

Departamento de Engenharia Informática

Ano letivo de 2018/2019

Compiladores

Projeto final: Compilador para a linguagem *deiGo*

Realizado por:

Frederico Cardoso, número 2012138904

Renato Santos, número 2015237457

I – Gramática re-escrita

Meta 1

Na meta 1 do projeto, começou-se por fazer uma análise lexical, isto é, dado um código de um programa ao analisador, este separa os *tokens* que encontra, de acordo com a especificação da linguagem. Para a correta deteção destes tokens, foi utilizada a gramática fornecida.

Para a deteção de erros lexicais e sintáticos, foram criadas as variáveis *line* e *col*, de forma a poder apresentar mensagens de erro com a localização do token que o originou, como exemplifica o enunciado. Para a meta 1 foi utilizada a ferramenta *LEX*, que analisa as entradas em forma de *tokens* e devolve as suas designações, desde que se chame o programa com a *flag –l*. À semelhança das posteriores metas, a invocação do programa sem *flag* apenas origina as mensagens de erro.

Meta 2

Após a análise lexical, para esta meta de análise sintática foi usada a ferramenta *YACC* que permite criar uma gramática sem ambiguidade, na qual os *tokens* entram de forma ordenada, garantindo a prioridade de operações. A estrutura *union* criada serve para ordenar em formato de lista ligada os *tokens* recebidos.

```
%union{
    char *val;
    struct node *node;
}
```

Após leitura do enunciado, reparámos que a gramática se encontra na notação *EBNF*. Como o *YACC* é um analisador sintático *top-down*, esta notação não é aceite pelo analisador, de forma a que tivemos que fazer algumas alterações, quando há problemas de recursividade à direita, que se podem transmitir em conflitos de *shift-reduce*. Uma alteração que foi feita para o prevenir foi criar estados responsáveis por serem recursivos à esquerda, de modo a poderem terminar sem gerar esse tipo de conflitos. Um exemplo disso é:

```
ExprList:
        ExprList COMMA Expr
    |   Expr
    ;
```

:

Outro problema encontrado foi que a gramática é ambígua, ou seja, o que faz com certas ordens de operações, por exemplo, sejam ignoradas. Desta forma, e de acordo com a especificação da linguagem *Go*, utilizámos as suas precedências. Retirámos também a associatividade dos *IF*, dos *ELSE* e dos *FOR*.

```
%left COMMA
%right ASSIGN

%left OR
%left AND
%left LT LE GT GE EQ NE
%left PLUS MINUS
%left STAR DIV MOD

%right NOT
%left LPAR RPAR LSQ RSQ

%nonassoc IF ELSE FOR
```

II – Estruturas de dados da AST e da tabela de símbolos

Para a segunda meta, tendo a gramática pronta a ser analisada sintaticamente, criámos então uma estrutura do tipo árvore, de forma a ser percorrida com o algoritmo *DFS*. Esta estrutura é composta por 4 elementos, o *token_type*, que armazena o tipo de variável que é (*VarDecl*, *Program*, *etc.*), o *token_value*, que nos diz qual o valor armazenado por esse tipo de variável, se existir (*Id(ex1*), *Int(123), etc.)*, e dois dados do tipo nó, que são apontadores para o filho desse mesmo nó e para o seu irmão. Esta árvore apenas é impressa se o programa for chamado com a *flag –t*. Desta forma, podemos percorrer e adicionar elementos à árvore, de acordo com as seguintes funções:

```
void print_ast(node *current_node, int npoints);

node *create_node(char *type, char *value);

void add_child(node *parent, node *child);
void add_first_child_varDecl(node *parent, node *new_child);
void add_brother(node *first, node *last);
int has_brother(node *child);
```

Em relação à meta 3, a última que fizemos, que consistia em criar tabelas indexadas das diferentes funções variáveis utilizadas no programa, criámos também uma estrutura um pouco mais complexa para armazenar toda a informação:

- ***int func:*** para verificar se é a tabela global ou se é a tabela de uma função.
- ***char name[]:*** nome da tabela.
- ***char type[]:*** tipo de retorno da função (*NULL* se for a tabela global).

- ***int n_params:*** número de parâmetros da tabela (se for a global, funções).
- ***char **params:*** os parâmetros no formato [int | 3 | id | ex].
- ***char param_str[]:*** parâmetros em string, para impressão destes na global.

- ***int n_vars:*** número de parâmetros de entrada da função.
- ***char **vars:*** as variáveis no mesmo formato dos parâmetros.

- ***struct table *next:*** um nó para a próxima tabela.

Antes de começar a inserir os dados nas tabelas, a árvore é percorrida uma primeira vez para identificar nomes de variáveis globais e de funções, e só depois, numa segunda travessia da árvore, é que as funções são expandidas. As funções utilizadas para esta meta, para criar os símbolos e as tabelas foram as seguintes:

```
/* Funções gerais para tabelas*/
void init_global_table();
table *init_func_table();

/* Funções 'utils' para tabelas */
void add_name_to_table(table *t, char *name);
void add_type_to_table(table *t, char *type);
void add_param_to_table(table *t, char *param, char *type);
void add_var_to_table(table *t, char *var, char *type);
void increaseParams(table *t);
void increaseVars(table *t);

/* Misc */
void print_tables();
```