COMPILADOR DE DEIGO

Projeto de Compiladores

2021/2022

Edgar Filipe Ferreira Duarte 2019216077, Rodrigo Francisco Ferreira 2019220060

1- Introdução

No âmbito da cadeira de compiladores, foi desenvolvido um compilador de deiGo, um subconjunto da linguagem Go. Foram implementadas todas as metas propostas, sendo que a meta 4 ficou muito incompleta.

2- Gramática Reescrita

Foi necessário reescrever a gramática apresentada no enunciado uma vez que não existe forma direta de tratar de tokens opcionais e ou repetentes. Assim, para solucionar este problema, decidimos que uma boa opção seria criar estados adicionais aos apresentados. O critério para a criação de um novo estado foi o seguinte:

Caso exista um estado e/ou tokens opcionais numa produção realizamos uma de duas opções:

- Se apenas existisse um estado ou token opcional como por exemplo "*Statement −→ RETURN [Expr]*" criamos no estado atual 2 produções, uma com o estado/token opcional e outra produção sem.
- Nos casos onde os estados e/ou tokens opcionais ocorreram em produções mais complexas, como por exemplo "*FuncInvocation −→ ID LPAR [Expr {COMMA Expr}] RPAR*", procedemos à criação de um estado adicional onde existe uma produção com os estados e/ou tokens opcionais e uma produção que se encontra vazia.

Nos casos de produções possuírem estados e/ou tokens que se possam repetir, como por exemplo "*Parameters −→ ID Type {COMMA ID Type}*", criamos um estado onde existe uma produção que possui os tokens e/ou estados a repetir e no final é chamado o próprio estado para criar um loop. O estado possui também uma produção vazia para permitir a saída do loop.

Nos estados onde as produções possuem uma possibilidade de combinações, são criadas produções para cada um dos possíveis casos. Um exemplo deste caso "*Expr −→ Expr (OR | AND) Expr*".

3-Algoritmos e estruturas de dados da AST e tabela de símbolos.

Para a criação da árvore AST foi, primeiramente, necessário fazer a árvore sintática. Esta árvore, na realidade, é uma lista de listas ligadas. Os nós da árvore estão definidos da seguinte forma:

```
struct token{
    char *symbol;
    int line, column;
    param first_param;
};


struct node{
    enum class_name class;
    char *type;
    struct token *token;
    struct node_list *children;
    int parent_is_call;
};

struct node_list{
    struct node *node;
    struct node_list *next;
};
```

É de notar que a *struct node_list* é redundante, visto que se passarmos o campo **next* para a *struct node* obtemos o mesmo resultado. Não foi corrigimos pois já tínhamos muito código e dar *refractor* iria dar muitos problemas.

Cada nó tem: um tipo (*float, int* , etc.); tem uma *struct token* que contem o seu símbolo (valor associado), as linhas e colunas onde aparece no input e, caso seja uma função, um ponteiro para o seu primeiro parâmetro; e tem um *enum* que atribui uma "propriedade" ao nó (usado em *switch* cases).

Criação da árvore sintática:

Cada nó pode ter ligação para um irmão e/ou para um filho. Cada produção, quando reduzida, devolve, para a produção "pai", o nó que criou. Este depois utiliza o nó e junta-o à sua lista de filhos (*node->children*). Se um "pai" receber mais do que 1 filho, ele encaminha o próximo filho para a próxima posição disponível no *array* de filhos (*node->children->next* no caso de ser o segundo filho). É assim criada a árvore.

Criação da árvore anotada:

Primeiramente precisamos de criar tabelas de símbolos. As estruturas escolhidas foram as seguintes:

```
struct parameter{
    char *params;

    param next;
};

struct element_table{
    char *value;
    char *type;
    int is_param;
    char is_used;
    int has_been_passed;
    int line;
    int column;
    int variable_value;
    int previous_variable_value;
    int was_used_assembly;
    int is_global;


    param first_param;
    elem_table next;
};
```

```
struct table{
    char *name;
    int current_index_variables;
    param first_param;
    tab next;
    elem_table first_elem;
};
```

A ideia é a mesma que nos nós da árvore visto que é uma lista de listas. Existe uma lista de nodos *table*, onde cada *table* tem uma lista de nodos *element_table* e uma lista de nodos *parameter*. Cada elemento tem um valor e um tipo associado, sendo que existem inúmeras *flags* para resolver pequenos problemas.

Para as preencher, passamos pela árvore sintática toda e sempre que encontramos um VARDEC acrescentamos uma entrada na tabela correta.

Passamos agora à fase de criação da AST. Começamos por percorrer a árvore em pré-ordem até aparecer uma declaração de uma função. Até aqui atribuímos tipos a variáveis globais. A partir daí começamos a percorrer a árvore da "direita para a esquerda", visto que isto trás muitas vantagens, uma vez que, quando aparece uma operação de, por exemplo, adição, os seus IDs filhos já têm um tipo associado. Quando sairmos da função voltamos a percorrer a árvore em pré-ordem até aparecer uma nova função.

No fim verificamos ainda se existem parâmetros que não estão a ser acedidos.

4- Geração de Código

Esta meta ficou incompleta. Foi implementado a de declaração e atribuição de variáveis e print das mesmas e, parcialmente, expressões (só o básico).

Primeiramente precisamos de guardar qualquer *string* que apareça no programa numa variável global em LLVM. Para o efeito, percorremos a árvore inteira e sempre que aparece um nó *string* adicionamos o registo em variável global e adicionamos o valor da variável numa lista ligada de *strings* para, no momento da geração de código dentro das funções, podermos chamar a variável.

```
struct string_list{
    tree_node node;
    string_glob next;
    int value;
};
```

Seguidamente, percorremos de novo a árvore e procuramos por variáveis globais, visto que também têm de ser imprimidas no topo do script de LLVM e vemos se existe alguma função de print. Se existir temos de por como variáveis globais todos os seus argumentos possíveis.

Com as variáveis globais declaradas, passamos às variáveis "internas" das funções. Percorremos os nós das funções em pré-ordem e sempre que aparece um nó VARDEC alocamos memória para a nova variável, através da função *aloca*. Sempre que aparece um nó ASSIGN temos de atribuir um valor ao nó a ser *assigned* através da função *load*.

Em nós de PRINT utilizamos a função *printf()* do C, convertida para LLVM.

Sobre expressões só foi implementado contas algébricas básicas de inteiros e *floats*. Calcula-se em C o resultado da expressão, ao percorrer os nós da operação da "direita para a esquerda", seguindo-se um *store* do resultado para a variável desejada.