UNIVERSIDADE D COIMBRA

Relatório do Projeto de Compiladores 2019/20

Compilador para a linguagem Juc

Trabalho realizado por:
Carolina de Castilho Godinho - 2017247087 - uc2017247087@studente.uc.pt
Maria Paula de Alencar Viegas - 2017125592 - uc2017125592@student.uc.pt

1) Gramática Re-escrita

Para o projeto, fizemos um analisador lexical e sintático para a linguagem Juc e para tal foram utilizados os tokens de Java e as ferramentas LEX e YACC.

Em relação ao analisador lexical produzimos um código que detetasse os *tokens* indicados no enunciado que seguem a especificação da linguagem e que devolvem as suas designações desde que se use a *flag -l*. Caso não seja colocada nenhuma flag são apenas apresentadas as mensagens de erros existentes.

Na deteção de erros sintáticos e lexicais foram usadas as variáveis *col_count* e *line_count* para que a mensagem de erro consiga apresentar a localização do token que o originou.

Foi considerado para o analisador sintático uma gramática sem ambiguidade, respeitando as regras de associação dos operadores e as precedências. A estrutura *union* ordena os tokens recebidos por meio de uma lista ligada.

Para retirar a ambiguidade da gramática, foram precisos alguns ajustes na notação EBNF. Por exemplo, foram criados estados responsáveis pela recursividade à esquerda para conflitos de shift-reduce relacionados à recursividade à direita.

```
Expr:                   Expr1
                        |   Assignment
                        ;
Expr1:                  Expr1 PLUS Expr1




                        |   Expr1 MINUS Expr1
```

Também foram estabelecidas ordens e associatividades foram retiradas de determinadas operações. São elas:

```
%right ASSIGN
%left OR
%left AND
%left XOR
%left EQ NE
%left LT LE GT GE
%left LSHIFT RSHIFT
%left PLUS MINUS
%left STAR DIV MOD
%right NOT

%nonassoc IF
%nonassoc ELSE
```

2) Algoritmos e estruturas de dados da AST e da tabela de símbolos

Foi construída uma árvore de sintaxe abstrata que aparece caso não sejam encontrados erros sintáticos no código analisado e se use a *flag -t*.

Para a árvore criámos uma estrutura *no* que guarda o tipo da variável, o valor atribuído, caso exista, um ponteiro para o filho desse nó e outro ponteiro para um possível irmão desse nó.

```
typedef struct no{
    char *nome;
    char *valor;
    struct no* filho;
    struct no* irmao;

} No;
```

A árvore é percorrida com o algoritmo DFS de busca de profundidade e é construída pelas seguintes funções:

```
No* cria_no(char* nome, char* valor);
void add_irmao(No* n1, No* n2);
void add_filho(No* n, No* filho);
int check_irmao(No* n);
void tratamentoIDRep(No* pai, No* filho);
void print_tree(No* n, int nivel);
void free_tree(No* n);
```

Além de criar o nó, adicionar um irmão/filho e imprimir/destruir a árvore, temos as seguintes funções para específicos casos:

check_irmao() que verifica se um dado nó tem irmãos, utilizada para impedir a criação de nós supérfluos em *Statements.*

tratamentoIDRep() para ordenar uma sequência de *ID* em *VarDecl* e *FieldDecl* conforme o enunciado.

Também foram construídas tabelas de símbolos durante a análise semântica, sendo elas visualizadas com o uso da *flag -s*. Estas podem ser tabelas de símbolos globais e tabelas de símbolos dos métodos. A estrutura responsável por guardar as informações da tabela tem os seguintes parâmetros:

```c
typedef struct table {
    int isMethod; //se eh Global (0) ou se eh Method (1)
    char *nome; // nome do mettodo
    char *type; // tipo de retorno

    int n_params; // numero de parâmetros
    char **params; //array de parâmetros [n_param][nome_param] -linha a seguir- [n_param]
    char params_str[MAX_S * 10]; //Parametros para string (x,y,z)

    int n_vars; //numero de variaveis
    char **vars; //array de variaveis

    struct table *next;
} table;
```

Antes de inserir os dados nas tabelas, a árvore é percorrida para identificar os nomes das variáveis globais e dos métodos. Numa segunda travessia da árvore, os métodos são expandidos. As funções a seguir permitem a construção das tabelas:

```c
void init_global_table();
table *init_method_table();

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);

void print_tables();
```

init_global_table() e *init_method_table()* são responsáveis pela inicialização das tabelas de símbolos.

add_param_to_table() adiciona um parâmetro e o seu tipo ao array de parâmetros da tabela.

add_var_to_table() adiciona uma variável e seu tipo ao array de variáveis da tabela.

increaseParams() realoca espaço do array de parâmetros.

increaseVars() realoca espaço do array de variáveis,

print_tables() imprime as tabelas.