Relatório sobre o compilador para a linguagem deiGo

Compiladores 2021/2022

Trabalho realizado por:

- Liu Haolong 2018288018
- João Dionísio

Índice

Meta 2, transformação da gramática e estrutura de dados da AST

Transformações na gramática

Nesta etapa começámos por transformar a gramática dada no enunciado, em notação EBNF, para a notação BNF, porque a primeira é ambígua e permite a possibilidade de 0 ou mais produções.

Isto resultou em novas produções: no caso em que o símbolo é opcional, foi necessário criar dois resultados da mesma produção, um incluindo o símbolo e outro excluindo-o; no caso em que o símbolo se repete 0 ou mais vezes, foi preciso uma nova produção resultando nos símbolos a repetir e no símbolo λ, com a própria produção em recursão, resultando numa repetição de 0 ou mais vezes dos símbolos. É importante realçar também que para resolver ambiguidades é necessário definir as precedências.

Alguns casos de transformação de produções opcionais:

- Vars and Statements:

VarsAndStatements ⟶ VarsAndStatements [VarDeclaration | Statement] SEMICOLON | ϵ

```
varsandstatements:  varsandstatements statement SEMICOLON
    |             varsandstatements vardeclaration SEMICOLON
    |                varsandstatements  SEMICOLON
    |
    ;
```

- Program:

Program ⟶ PACKAGE ID SEMICOLON Declarations

```
program:            PACKAGE ID SEMICOLON declarations
    |               PACKAGE ID declarations
    ;
```

Alguns SEMICOLON são opcionais(nos casos implementados no lex), o que resulta também numa produção com dois resultados, um com SEMICOLON e outro sem.

Casos com 0 ou mais repetições:

- Declarations:

Declarations ⟶ {VarDeclaration SEMICOLON | FuncDeclaration SEMICOLON}

```
declarations:       declarations vardeclaration SEMICOLON
    |               declarations funcdeclaration SEMICOLON
    |       /*vazio*/
    ;
```

Nesta produção todos os símbolos se repetem, o que resulta na recursão da mesma produção;

- VarSpec:

$$\text{VarSpec} \longrightarrow \text{ID } \{\text{COMMA ID}\} \text{ Type}$$

```
varspec:            ID varspec_comma type
    ;

varspec_comma:           varspec_comma COMMA ID
    |           /*vazio*/
    ;
```

- Parameters:

$$\text{Parameters} \longrightarrow \text{ID Type } \{\text{COMMA ID Type}\}$$

```
parameters:         ID type parameters_comma

parameters_comma:        parameters_comma COMMA ID type
    |               /*vazio*/
    ;
```

- Statements:

$$\text{Statement} \longrightarrow \text{LBRACE } \{\text{Statement SEMICOLON}\} \text{ RBRACE}$$

$$\text{Statement} \longrightarrow \text{IF Expr LBRACE } \{\text{Statement SEMICOLON}\} \text{ RBRACE [ELSE LBRACE } \{\text{Statement SEMICOLON}\} \text{ RBRACE]}$$

$$\text{Statement} \longrightarrow \text{FOR [Expr] LBRACE } \{\text{Statement SEMICOLON}\} \text{ RBRACE}$$

```
statement:          ID ASSIGN expr
|               LBRACE statement_semicolon RBRACE
|               IF expr LBRACE statement_semicolon RBRACE ELSE LBRACE statement_semicolon RBRACE
|               IF expr LBRACE statement_semicolon RBRACE
|               FOR expr LBRACE statement_semicolon RBRACE
|               FOR LBRACE statement_semicolon RBRACE
|               RETURN expr
|               RETURN
|               funcinvocation
|               parseargs
|               PRINT LPAR expr RPAR
|               PRINT LPAR STRLIT RPAR
|               error
;

statement_semicolon:         statement SEMICOLON statement_semicolon
|                /*vazio*/
;
```

- FuncInvocation:

FuncInvocation ⟶ ID LPAR [Expr {COMMA Expr}] RPAR

```
funcinvocation:     ID LPAR RPAR
    |             ID LPAR expr funcinvocation_comma RPAR
    |               ID LPAR error funcinvocation_comma RPAR
    ;

funcinvocation_comma:    funcinvocation_comma COMMA expr
    |                 funcinvocation_comma COMMA error
    |
    ;
```

Estas produções listadas exigiam 0 ou mais repetições de alguns símbolos, o que se resolveu com a solução referida anteriormente: recursão das novas produções.

Declarada a gramática, é possível a análise da sintaxe dos tokens provenientes da fase anterior, gerando uma estrutura de dados.

Estrutura de dados implementada na gramática

Foi implementada nesta fase uma árvore de sintaxe abstrata para armazenar os elementos encontrados na fase da análise sintática, que se resume numa estrutura com poucos detalhes sobre a sintaxe real, que apenas mostra os detalhes estruturais do conteúdo.

No nosso caso cada elemento é inserido como um nó, estando cada um deles a apontar para um filho e um irmão e guardando dados essenciais para a sintaxe. Se existir recursão é adicionado ao nó irmão. Para os casos de produção vazia e error, o topo da pilha é definido como NULL.

```
typedef struct node {
    int invalid;
    char* symbol;
    char* annotation;
    char* key;
    char* tipo;
    int line;
    int column;
    class tipoclass;
    struct node* irmao;
    struct node* filho;
}node;
```

Originalmente, apenas os atributos irmao, filho, tipoclass(categoria de produção do nó) e symbol(símbolos terminais como intlit e ID) pertenciam à estrutura node. Os outros atributos foram adicionados para facilitar a implementação da meta 3.

A impressão da árvore é feita recursivamente, passando por cada nó, com prioridade no nós irmao, imprimindo pontos representando a profundidade, seguido do symbol/tipoclass.

```
typedef enum { program, decl_list, func_head, func_param, funcbody, func_decl, vardec, varsandstatements,
factor, expression, parseargs, statement, funcinvocation, block, blocky }class;
```

Meta 3, implementação da tabela de símbolos

Esta fase está dividida essencialmente em três partes: construção das Tabelas de Símbolos, acréscimo das anotações a imprimir em conjunto com a árvore e verificação de erros semânticos. Para a criação das Tabelas de Símbolos criámos quatro estruturas de dados, uma com a lista das tabelas, outra duas com cada tabela e uma com elementos da tabela.

```
typedef struct table_element {
    int is_func;
    char name[32];
    basic_type typeenum;
    //daqui
    char *symbol;
    char *tipo;
    char *key;
    char *optionalkey[MAX_PARAM_SIZE];
    char *optional[MAX_PARAM_SIZE];
    int cur_param_index;
    int is_param;
    //ate aqui
    struct table_element* irmao;
}table_element;
```

```
typedef struct h_table {
    char *name;
    int is_defined;
    struct table_element *head;
    struct table_element *last;
} h_table;
```

```
typedef struct l_node {
    struct h_table *table;
    struct l_node *irmao;
} l_node;
```

```
typedef struct symbol_table_list {
    struct l_node *first;
    struct l_node *last;
} symbol_table_list;
```

Depois de criar as estruturas criamos algumas funções para percorrer a árvore: uma função check_program, que chama a função check_vardec; outra função check_funcdecl, que por sua vez chama a função check_expression cada vez que encontra um nó do tipo expression. Desta forma, começamos por criar uma tabela “Global” no check_program e depois, consoante o tipo de nós, criamos uma nova tabela (se for uma declaração de função) ou criamos um novo elemento com os seus atributos na tabela respetiva. Para isto usamos algumas funções, como a put_hash_table_and_params, que verifica se o elemento a inserir na tabela é uma função e quais os seus parâmetros, a add_symbol_table, que junta uma tabela já criada a uma lista de tabelas e a create_table, que cria uma tabela.

```
int check_program (node * p){
    table_list = create_symbol_table_list();
    add_symbol_table(table_list, create_table("Global",1));
    global = table_list->first->table;

    int error_count = 0;

    error_count += check_decl_list(p->filho);

    return error_count;
}
```

```
int check_funcdecl(node* iv, int body){

    int erro_counter = 0;

    if(!body){
        char* type_spec = put_hashtable_and_params(iv,1);
        char buffer2[BUFFER_SIZE];
        strcpy(buffer2,"(");
        int first = 1;

        for(int i = 0; i < get_hashtable(global, iv->key)->cur_param_index; ++i) {
            if(first != 1)
                strcat(buffer2,",");
            first = 0;
            strcat(buffer2,get_hashtable(global, iv->key)->optional[i]);
        }
        strcat(buffer2,")");
        char buffer3[BUFFER_SIZE];
        strcpy(buffer3,"Function ");
        strcat(buffer3,(char*)strdup(iv->key));
        strcat(buffer3,buffer2);
        add_symbol_table(table_list, create_table(strdup(buffer3),1));
        tableaux = table_list->last->table;

        if(iv->tipo != NULL){
            put_hashtable(tableaux, "return", type_spec , 0,0);
        }else{
            put_hashtable(tableaux, "return", "none" , 0,0);
        }
        for(int i = 0; i < get_hashtable(global, iv->key)->cur_param_index; ++i) {
            put_hashtable(tableaux, get_hashtable(global, iv->key)->optionalkey[i], get_hashtable(global, iv->key)->optional[i] , 1,0);
        }
```

```
void put_hashtable(h_table *table, char *key, char *type, int is_param,int is_func) {
    if(table != NULL) {
    //table_element *lasnode = table->last;
    table_element *temp = (table_element *)malloc(sizeof(table_element));
    temp->is_func = is_func;
    temp->key = key;
    temp->tipo = type;
    temp->is_param = is_param;
    temp->cur_param_index = 0;
    temp->irmao = NULL;
    if(table->last == NULL) {
        table->head = temp;
        table->last = table->head;
    } else {
        table->last->irmao = temp;
        table->last = table->last->irmao;
    }
    }
}
```

A função mais complexa é a “check_expression” e foi nesta parte que percebemos que deviamos guardar o tipo, a key e o symbol do node em variáveis diferentes. Esta recebe um nó correspondente a uma expressão e depois dependendo da “key” ou do “symbol” do nó, faz algo diferente. Foi nesta fase que sentimos que teriamos de criar mais funções complementares como o “check_expression_operation” que será chamado na “check_expression” quando no nó corresponde a uma operação. Depois, dentro do “check_expression_operation” existem mais casos específicos que garantem que os nós são lidos corretamente e as annotations são bem guardadas. No “check_expression” temos, essencialmente, as operações, as variáveis (intlit, realit, strlit) e casos mais específicos. Estes casos mais específicos são o id, a call, o if, o block, o for, o print e o return. Muitos

destes casos estão ligados entre si, casos como as operações e o id, o block e o if, o block e o for, etc.

```
if(!strcmp(n->key,"Print")){

    if(n->filho != NULL){
        if(check_expression(n->filho, tableaux)){

            erro_counter = 1;
        }
    }

    if(!strcmp(n->filho->annotation,"undef")){

        print_incompatible_types(n, n->filho->annotation,symbolforop(n->key),0);

    }
    return erro_counter;
}
```

```
if(!strcmp(n->key,"Return")){

    for(table_element* naux = tableaux->head; naux != NULL; naux = naux->irmao){
        if(!strcmp(naux->key, "return")){
            typeaux = naux->tipo;
        }
    }

    if(n->filho != NULL){

        if(check_expression(n->filho, tableaux)){

            erro_counter = 1;
        }
        if(strcmp(typeaux,n->filho->annotation)){
            char buffer[BUFFER_SIZE];
            sprintf(buffer,"%c%s",lower(n->key[0]),n->key+1);
            print_incompatible_types(n, n->filho->annotation,strdup(buffer),0);
            erro_counter = 1;
        }
    }

    return erro_counter;
}
```

Para tratar dos erros, temos funções que imprimem o erro do caso em questão. A maior dificuldade nesta parte, para além de descobrir todas as regras e exceções da linguagem DeiGo, é a impressão correta das colunas. A solução que encontramos foi alterar o yyval do lex de forma a este devolver uma struct e não uma só variável. Nesta struct podemos guardar os valores corretos do número da coluna, o que seria muito mais difícil de implementar da forma como tínhamos antes.

Meta 4, geração do código

Este projeto não tem implementado a meta 4, dado à impossibilidade de conciliar o tempo fora de aula com a meta de entrega.