Projeto de Compiladores 2024/25

Compilador para a linguagem deiGO

Trabalho realizado por:
Miguel Castela uc2022212972
Nuno Batista uc2022212972

1 Introduction

Este relatório descreve o desenvolvimento de um compilador para a linguagem deiGO, realizado no âmbito do projeto da disciplina de Compiladores do ano letivo 2024/25. O objetivo deste projeto é aplicar os conhecimentos adquiridos ao longo do curso na construção de um compilador funcional.

2 Grammar

Our implementation focuses on resolving ambiguities by explicitly defining operator precedence and associativity rules.
Non-termials where simplified by introducing auxiliary nodes to handle the optional elements and repititions.

- **Handling of Optional and Repeated Elements**:
  Optional and repeated elements are handled using specific grammar rules that provide alternatives for their presence or absence. By defining optional elements explicitly, we also ensure that the parser can handle cases where certain elements are missing. This is crucial for maintaining the correctness of the AST later on. By using optional nodes, we can easily append new declarations to the existing list.

Listing 1: StarCommaId Rule

```
StarCommaId:StarCommaId COMMA IDENTIFIER
    {
        $$ = $1;
        struct node *new_decl = new_node(VarDecl, NULL);
        add_child(new_decl, new_node(Identifier, $3));
        add_child($$, new_decl);

    // The type is added in the VarSpec rule
    }
    |
    {
        $$ = new_node(AUX, NULL);
    };
```

This rule, designed to manage repeated identifiers separated by commas, is particularly useful for handling multiple variable declarations in a single statement. For this, the rule employs recursion, where each node represents a variable declaration.

- **Recursive case**: When an identifier is followed by a comma and another identifier, the existing node (StarCommaId) serves as the "parent" or "father" node. A new VarDecl node is created for the subsequent identifier, with the identifier as its "child" or "son." This new VarDecl node is then added as a "child" to the parent node (StarCommaId).
- **Base case**:If no preceding identifiers exist, a new AUX node is created to serve as the root or starting "father" node for the list of identifiers.

By organizing the identifiers with this structure, the rule facilitates the processing and representation of variable declarations as a tree,

- **Optimization and Readibility** :
  Our grammar rules are desgined to be clear and readable, making it easier to understand the parsing process. We define operator precedence and associativity explicitly to resolve ambiguities. This ensures that expressioons are parsed correctly according to the deiGO language specifications.
  Examples of the transcription of the initnal grammar in EBNF notation to the Bison format:

Declarations – VarDeclaration SEMICOLON — FuncDeclaration SEMICOLON

```
Declarations : Declarations FuncDecl SEMICOLON
    {
        $$ = $1;
        add_child($$, $2);
    }
    | Declarations VarDecl SEMICOLON
    {
        $$ = $1;
        struct node *var_declarations = new_node(AUX, NULL);
```

```
        add_child($$, var_declarations);
        add_child(var_declarations, $2);
    }
    |
    {
        $$ = new_node(AUX, NULL);
    }
    ;
```

VarSpec – IDENTIFIER COMMA IDENTIFIER Type

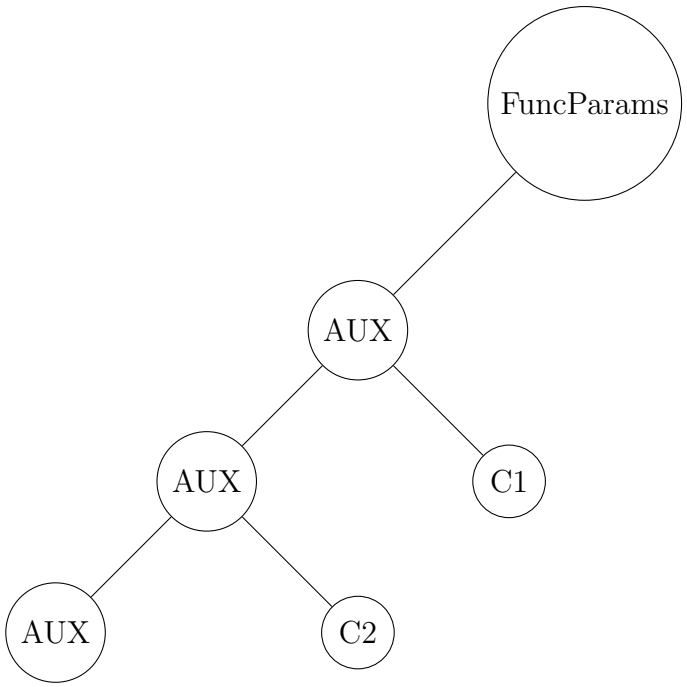
```
VarSpec : IDENTIFIER StarCommaId Type
    {
    $$ = new_node(AUX, NULL);
    struct node *new_decl = new_node(VarDecl, NULL);
    add_child(new_decl, $3);
    add_child(new_decl, new_node(Identifier, $1));
    add_child($$, new_decl);
    add_child($$, $2);
    }
    ;
```

FuncDeclaration – FUNC IDENTIFIER LPAR [Parameters] RPAR [Type] FuncBody

```
FuncDecl : FUNC IDENTIFIER LPAR OptFuncParams RPAR OptType FuncBody
{
    $$ = new_node(FuncDecl, NULL);
    struct node *new_func_header = new_node(FuncHeader, NULL);
    add_child($$, new_func_header);
    add_child(new_func_header, new_node(Identifier, $2));
    add_child(new_func_header, $4);
    add_child(new_func_header, $6);
    add_child($$, $7);
}
;
```

3 AST/Symbol Table Algorithms and Data Structures

- **Auxiliary AST nodes**:
  In our grammar, we use auxiliary(AUX) nodes to store the children of nodes with an undefined number of children. This approach helps in managing optional and repeated elements by acting as a container for multiple instances of a particular non-terminal. This function has the main goal of maintaining a clear and organized AST. By grouping related nodes under an AUX node, we can ensure that the tree structure is easy to traverse and understand. As this is a temporary container at the end of the syntax analasys we perform as DFS transversal to append the AUX nodes children to their respective parent nodes, using que RemoveAux function.EXMPLO (varSpec(?))



- **Locates**:
  This macro serves as a way to store the lines and columns of a specific node. In this way we can identify the nodes that are creating a semantic error.

- **AST Structs**:
  Firstly the node struct:

```
struct node {
    enum category category;
    char *token;
    int token_line, token_column;
    enum type type;
    char *parameter_list;
     struct node_list *children;
    };
```

Parameter	Description
category	An enum representing the category of the node.
token	A string representing the lexical token associated with the node.
token_line and token_column	Integers storing the line and column numbers of the token, useful for error reporting.
type	An enum representing the type of the node.
parameter_list	A string representing the list of parameters for function nodes.
children	A pointer to a **node_list struct**, representing the children of this node.

Table 1: Description of the node struct fields

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

This struct is used to store a list of children nodes, where each node is linked to the next node in the list.

- **Symbol Table Structs**:

```
        struct symbol_list {
    char *identifier;
    enum type type;
    struct node *node;
    int is_parameter;
    struct symbol_list *next;
    int was_used;
    int is_function;
    char *function_parameters;
};
```

Parameter	Description
identifier	A string representing the name of the symbol.
type	An enum representing the type of the symbol.
node	A pointer to the AST node associated with the symbol.
is_parameter	An integer indicating if the symbol is a function parameter.
next	A pointer to the next symbol_list element.
was_used	An integer indicating if the symbol was used.
is_function	An integer indicating if the symbol is a function.
function_parameters	A string representing the parameter types of the function.

Table 2: Description of the symbol struct fields

```
struct scopes_queue {
    struct symbol_list *table;
    struct scopes_queue *next;
    struct node *func_body; // The function body node is stor
    char *identifier;
};
```

This struct stores a queue of symbol tables for managing program scopes. Each scope includes a symbol table, a pointer to the next scope, a pointer to the function body, and the function's identifier.

- **Table Symbols**:

- **node children**:

- **Syntax Error Handling**:

4 Geração de Código