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

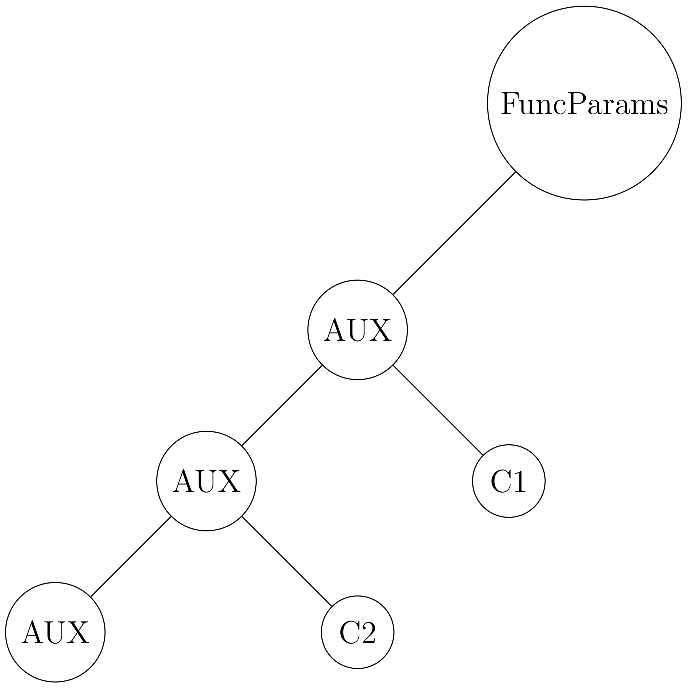
```
VarSpec : IDENTIFIER StarCommaId Type
    {
    $$ = new_node(AUX, NULL);
    struct node *new_decl = new_node(VarDecl, NULL);
    add_child(new_decl, $3);
    add_child(new_decl, new_node(Identifier, $1));
    add_child($$, new_decl);
    add_child($$, $2);
    }
    ;
```

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 elements and repetitions more effectively. These nodes help in managing repeated elements by acting as a container for multiple instances of a particular non-terminal. For example, in the VarSpec rule we use an AUX node to handle mutiple vairable declarations. 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 anaasys we perform as DFS transversal to append the AUX nodes children to their respective parent nodes.
- **Locates**:
  DESCRITO NA FOTOGRAFIA
- **StarCommaId as an example**:
- **RemoveAux**:
- **nodes**:
- **structs**:
- **Table Symbols**:
- **node children**:
- **Syntax Error Handling**:



4 Geração de Código