FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Projeto Compiladores

Compilador para a linguagem Juc

Licenciatura em Engenharia informática

Ano letivo 2022/23

Trabalho realizado por:

Leonor Reis, nº 2019210146

Ricardo Santiago, nº 2020219352

Índice:

1. Introdução

No âmbito da disciplina de Compiladores do ano letivo 2022/2023 foi pedido aos alunos o desenvolvimento de um compilador para a linguagem JUC, que é um subconjunto da linguagem java. Tendo em conta os conceitos lecionados nas aulas teóricas e práticas, o desenvolvimento deste projeto foi divido em 4 etapas: análise lexical, análise sintática, análise semântica e geração de código. A última etapa relativa a geração dá código não foi implementada e por isso não será abordada neste relatório.

Este relatório detalha de forma concisa as decisões técnicas relativas à transformação da gramática e aos algoritmos e estruturas de dados da AST e tabela de símbolos.

2. Transformação da Gramática

A gramática apresentada no enunciado é ambígua e está escrita em notação EBNF. Portanto, o primeiro passo foi a transformação da gramática para permitir a análise sintática ascendente com o yacc.

Na notação EBNF, [...] significa opcional, então criamos uma expressão em que os símbolos contidos nos […] apareciam e outra expressão em que em esses símbolos não apareciam. Deixamos a seguinte transformação numa das expressões regulares a título de exemplo:

Statement ⟶ IF LPAR Expr RPAR Statement [ELSE Statement]

```
Statement:   IF LPAR Expr RPAR Statement
           | IF LPAR Expr RPAR Statement ELSE Statement
```

Na notação EBNF, {...} significa “zero ou mais repetições”, então criamos um estado em que existe recursividade à direita. Ao passarmos para esse novo estado, podemos continuar nesse estado recursivamente ou parar (null). Deixamos a seguinte transformação numa das expressões regulares a título de exemplo:

VarDecl ⟶ Type ID { COMMA ID } SEMICOLON

```
VarDecl: Type ID VarDeclRep SEMICOLON


VarDeclRep: COMMA ID VarDeclRep
|
;|
```

Após estas alterações, tínhamos 321 *shift/reduce* conflitos. O próximo passo foi a remoção das ambiguidades da gramática dada, através da utilização de regras que estabelecem a precedência e a associatividade dos operadores. Definimos as regras de acordo com as regras aritméticas e de lógica, por exemplo, a multiplicação tem prioridade em relação à adição, a operação lógica AND tem prioridade em relação à operação OR e

a operação “==” tem prioridade à operação AND. O ELSE tem maior prioridade, seguido dos parêntesis.

Optou-se ainda pelo uso da keyword %prec que permite atribuir a uma regra da gramática a precedência associada ao operador especificado, ou seja, na imagem ilustrada utiliza-se a precedência do NOT para as produções, tal foi feito para evitar conflitos produzidos pela regra Expr→(MINUS|NOT|PLUS)Expr:

```
| MINUS Expr1   %prec NOT

| NOT Expr1

| PLUS Expr1      %prec NOT
```

Por fim, para evitar casos onde produções de Expr pudessem originar dois assignments seguidos optou-se por dividir essas produções com auxílio de um novo lado esquerdo da gramática (Expr1):

```
Expr: Assignment
| Expr1
;

Expr1:     Expr1 PLUS Expr1
```

3.Algoritimos e estruturas de dados da AST e da tabela de símbolos

Nesta fase do projeto, foi desenvolvido estruturas de dados com o intuito de representar a AST:

```
typedef struct node ast_tree;
struct node{
    token *token;
    char *type;
    ast_tree* filho;
    ast_tree* irmao;
    ast_tree* pai;
    char *anot;
};
```

```
typedef struct token{
    char *value;
    int num_linha;
    int num_coluna;
}token;
```

Estrutura ***token***: guarda o valor (caso ele exista) de um *token* enviado durante a análise lexical (yytext) através do atributo ***value***, bem como a linha e coluna onde foi identificado.

Estrutura ***ast_tree***, representa cada nó da AST através dos atributos: ***type***, guarda o tipo de dados de cada nó (“MethodDecl”,”Program”,etc); ***token***,, um ponteiro para a estrutura explicada anteriormente; ***filho, irmao e pai***, referências respetivamente para o primeiro filho do nó, para o seu irmão e para o seu pai; ***anot,*** tipo que vai ser anotado na AST (“int”,”undef”,etc).

O algoritmo foi desenvolvido através de funções como: ***create_node()***, permite criar um nó; ***adicionar_irmao()***, adiciona um nó como irmão; ***adicionar_filho()***, adiciona a um nó um novo filho (sendo este agora irmão dos nós que eram previamente filhos do nó pai); ***block_count()***, conta o número de irmãos de um nó, para saber se este será um novo do tipo “Block”; e ***print_tree(),*** itera recursivamente a arvore e imprime a mesma, respeitando a hierarquia entre variáveis globais e locais.

De seguida, deu-se início a construção da estrutura de dados *table_element*, cada elemento da tabela de símbolos:

```
typedef struct table_element {
    char *name;
    char *type;
    char *param_types;
    int num_param;
    int is_param;
    struct table_element *body;
    struct table_element *next_element;
    int line;
    int col;
    int alreadyUsed;
}table_element;
```

Caracterizada pelos atributos: ***name*** (nome da variável/método); ***type*** (tipo da variável ou tipo do valor que o método retorna); ***param_types*** (tipo dos argumentos recebidos pelo método); ***num_param*** (é igual ao número de argumentos caso seja um método e igual a -1 caso contrário); ***is_param*** (se igual a 1 é uma variável de um método e se igual a 0 é um método); ***body*** (caso se trate de um método aponta para os elementos que são variáveis do próprio método, caso se trate de uma variável é igual a NULL); ***nextElem*** (aponta para o próximo elemento que é um método); ***line*** e ***col*** (número da linha e coluna onde se localiza);e ***alreadyUsed*** usada para fazer verificações.

O algoritmo foi desenvolvido através de funções como: ***show_table()***,imprime a tabela de símbolos globais e as tabelas de cada método; ***check_two_member_op()*** ,analisa o tipo dos membros da operação e consoante a operação, anota o nó correspondente à mesma e ***check_one_number_op()***, para operações com um membro, ***add_new_value()***, responsável por fazer as anotações nos nós correspondentes aos RealLiit, BooLit, DecLit e StrLit , verifica também se os números estão dentro dos limites do JUC; ***check_call()***, verifica se existe algum método com o nome, tipo e argumentos iguais ao que foram dados no "Call" , caso não haja, é feita outra verificação tendo em conta o facto de um *int* pode ser "considerado" um *double* em JUC.