UNIVERSIDADE D COIMBRA

FACULDADE DE CIÊNCIAS E TECNOLOGIA DA UNIVERSIDADE DE COIMBRA

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Relatório de Projeto

Unidade curricular: Compiladores

2022/2023

Docente: Frederico Manuel Duarte Cerveira

Trabalho realizado por:

Carlos Miguel Oliveira Matos (2020245868)

Mariana Santos Magueijo (2020246886)

Turma: PL1

Índice

Introdução

Este projeto consiste no desenvolvimento de um compilador para a linguagem Juc, que é um subconjunto da linguagem Java de acordo com a especificação Java SE 9. Com o objetivo de percebermos melhor como fazer um compilador, o projeto foi dividido em quatros metas: análise lexical (meta 1), análise sintática (meta 2), análise semântica (meta 3), geração de código intermédio (meta 4).

Este relatório pretende demonstrar algumas decisões realizadas ao longo do desenvolvimento do projeto.

Gramática inicial em notação EBNF

Na meta 2 tivemos de fazer alterações de modo que, dado uma gramática ambígua inicial escrita em notação EBNF, se adaptasse com a análise lexical feita na meta 1, tendo sido reescrita para YACC.

Foi necessário ter em conta as regras de associação dos operadores e as precedências, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens Juc e Java.

Tendo em conta que poderiam existir símbolos terminais ou não terminais que 1) fossem opcionais [], 2) que apareciam zero ou mais vezes {} ou 3) dos quais apenas se pode escolher um (), tivemos de reescrever a gramática utilizando uma recursividade à direita. Como apresentado no exemplo a seguir, criamos um símbolo terminal auxiliar *Program2* para o *Program,* permitindo a resolução do problema 2) apresentado.

```
Program → CLASS ID LBRASE { MethodDecl | FieldDecl | SEMICOLON } RBRASE

Program: CLASS ID LBRACE Program2 RBRACE
       | CLASS ID LBRACE Program2 RBRACE error
       ;

Program2: /*empty*/
        | MethodDecl Program2
        | FieldDecl Program2
        | SEMICOLON Program2
        ;
```

No exemplo a seguir resolvemos os problemas 1) e 3).

```
MethodHeader → ( Type | VOID ) ID LPAR [ FormalParams ] RPAR

Methodheader: Type ID LPAR MethodHeader2 RPAR
            | VOID ID LPAR MethodHeader2 RPAR
            ;

Program2: /*empty*/
        | FormalParms
        ;
```

Para evitar que haja conflitos entre símbolos não terminais, definimos prioridades para cada um dos símbolos não terminais, resolvendo o problema das precedências:

```
%left ARROW
%right ASSIGN
%left OR
%left AND
%left XOR
%left EQ NE
%left GE GT LE LT
%left LSHIFT RSHIFT
%left PLUS MINUS
%left STAR DIV MOD
%right NOT UNARY
%left LPAR RPAR LSQ RSQ
%right ELSE
```

Árvore de sintaxe abstrata (AST)

Ainda na meta 2 foi desenvolvida uma árvore de sintaxe abstrata para ser possível fazer a análise sintática e semântica.

Com este objetivo, foi criada uma estrutura *tipo_no*, para a identificação dos diferentes tipos de nós que podem existir na nossa árvore.

```
typedef enum tipo_no{
    no_raiz,
    no_declaracao,
    no_metodos,
    no_statments,
    no_operadores,
    no_terminais,
    no_id
}tipo_no;
```

A nossa árvore é constituída por diversas estruturas no, cada um com inúmeras características.

Para a criação e inserção de um novo nó na árvore criámos a função *criar_no* , onde passamos o tipo de nó, o nome e o tipo como parâmetro, e inicializamos o pai, filho e irmão a NULL. Nas funções *adicionar_filho* e *adicionar_irmao* passamos o nó que queremos adicionar como filho ou irmão e o nó antigo que vai ficar como pai.

```
typedef struct no{
    no1 pai;
    no1 filho;
    no1 irmao;
    tipo_no no_tipo;
    char *valor;
    char *tipo;
}no;
```

Tabela de símbolos

Na meta 3 desenvolvemos código para implementar uma tabela de símbolos, em que a tabela principal continha um identificador para as variáveis globais e para os métodos definidos. Para além disso era criada uma tabela de símbolos para cada método.

Para tal criamos uma estrutura chamada *metodo*, que tinha um identificador para distinguir se o nó era um método ou uma variável global. Caso fosse método guardávamos o seu nome, o tipo que retornava, os parâmetros que tinha (com o respetivo nome e tipo) e as variáveis declaradas no método. Se fosse uma variável global, apenas guardávamos o seu nome e o tipo.

```
typedef struct metodo{
    char *tipo;
    char *nome_class;
    char *nome_funcao;
    char *tipo_retorna;
    char **tipo_param;
    char **nome_param;
    int num_param;
    char *variavel;
    struct metodo *proximo_metodo;
}metodo;
```

De modo a ser possível imprimir as tabelas consoante aparecem no código, criamos uma lista que continha um ponteiro para o primeiro método.

```
typedef struct lista{
    metodo *inicio;
}lista;
```

Para criar a tabela de símbolos usamos a função *criar_tabela* que retornava a lista de métodos para depois ser usada para imprimir as tabelas na função *imprime_tabela*.