Licenciatura em Engenharia Informática

Compiladores

2019/2020 – 2º Semestre

Compilador para a linguagem Juc

Gonçalo Manuel Oliveira e Sousa *2016244072*
João Filipe Carnide de Jesus Nunes *2017247442*

Introdução

Para este trabalho prático foi pedido o desenvolvimento de um compilador para a linguagem Juc, linguagem esta que é um subconjunto da linguagem Java. Foram pedidas a realização das diversas etapas da construção de um compilador, análise lexical (meta 1), análise sintática (meta 2), análise semântica (meta 3) e geração de código (meta 4).

Este relatório pretende mostrar como foram tomadas certas decisões no que diz respeito à rescrição da gramática na meta 2 (secção 1), a implementação da árvore sintática abstrata (AST) e da tabela de símbolos nas metas 2 e 3 (secção 2), respetivamente, e à geração de código da meta 4 (secção 3). Contudo este relatório não apresentará a última secção sobre a explicação da geração de código pois não foi desenvolvida em termos práticos.

Secção 1 – Gramática rescrita

Na meta 2 foi pedido para realizar a análise sintática, segundo passo na realização do compilador para a linguagem Juc. Esta análise foi feita com base numa gramática ambígua inicial em notação EBNF e, a partir desta, rescrita para yacc de modo a ser feita a ligação com a análise lexical realizada na meta 1.

A rescrição da gramática foi feita através dos princípios básicos do yacc, contudo, existiam algumas restrições mediante as regras de associação dos operadores e precedências, entre outros aspetos.

Um dos aspetos que teve de ser contornado na rescrição da gramática foi o facto de existirem símbolos terminais ou não terminais que podem aparecer zero ou mais vezes ({...} na notação EBNF). Isto foi contornado no yacc através da recursividade à direita, criando um símbolo terminal auxiliar de modo a ser possível estes símbolos poderem aparecer numa árvore de derivação uma ou mais vezes.

Por exemplo, Program ⟶ CLASS ID LBRACE { MethodDecl | FieldDecl | SEMICOLON } RBRACE foi contornado da seguinte forma:

```
Program: CLASS ID LBRACE ProgramScript RBRACE
        ;
ProgramScript: /*empty*/
                | MethodDecl ProgramScript
                | FieldDecl ProgramScript
                | SEMICOLON ProgramScript
                ;
```

Outros dois aspetos, semelhante ao anterior, foi a existência de símbolos terminais e não terminais opcionais ([…] na notação EBNF) e símbolos que só pode ser escolhido um deles ((…) na notação EBNF). Estes aspetos foram contornados criando outras regras na gramática para um certo símbolo terminal ou criando símbolos terminais auxiliares, dependendo da complexidade da situação.

Por exemplo, MethodHeader ⟶ (Type | VOID) ID LPAR [FormalParams] RPAR foi rescrito da seguinte maneira:

```
MethodHeader: Type ID LPAR MethodHeader2 RPAR
            | VOID ID LPAR MethodHeader2 RPAR
            ;
MethodHeader2: /*empty*/
            |  FormalParams
            ;
```

Também teve de se ter em conta as precedências, ou seja, prevenir que se crie conflitos entre certos símbolos não terminais. Isto foi contornado no yacc identificando a prioridade de cada símbolo não terminal:

```
%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
%left LPAR RPAR LSQ RSQ
%right ELSE
```

Assim não existirão conflitos respetivamente às operações, por exemplo, a multiplicação terá sempre prioridade em relação à adição.

Contudo foi ainda necessário mudar o nível de precedência em regras da gramática específicas para não existirem outro tipo de conflitos.

A dado exemplo temos a regra Expr ⟶ (MINUS | NOT | PLUS) Expr que foi contornada usando %prec nas regras do MINUS e do PLUS.

Secção 2 – AST e Tabela de Símbolos

Nas metas 2 e 3 foi pedido o desenvolvimento de uma árvore de sintaxe abstrata (AST) e de uma tabela de símbolos, respetivamente, de modo a ser feita as análises sintáticas e semânticas.

No que diz respeito à implementação da AST, mediante à sua estrutura de dados, foi realizada uma enumeração em C para a identificação dos diferentes tipos de nós que podem existir na árvore: nós raiz, nós de declaração de variáveis, nós de definição de métodos, nós de statements, nós dos operadores, nós terminais e nós dos diferentes tipos de id's. Foi também implementada uma estrutura em C com a as características de cada nó como valor, tipo, número de nós, pai, filho, irmão e um *char* type_tab que representa o tipo que será identificado a partir da tabela de símbolos.

Mediante aos algoritmos que foram implementados na AST temos a criação de um novo nó, onde todos os valores são inicializados a 0, NULL ou string vazia, dependendo das variáveis em causa; a adição de um novo nó, onde esse nó é considerado como filho de um nó previamente existente; a adição de um irmão; uma função de contador de irmãos e uma função para imprimir a AST (usando a flag -t) e a AST anotada (usando a flag -s).

Quanto à implementação da tabela de símbolos, no que diz respeito à estrutura de dados foram implementadas duas estruturas em C, uma referente ao nó da tabela onde estão presentes o que vai ser impresso em cada linha da tabela, como o Name, ParamTypes, Type e param, a outra estrutura é referente à tabela em si, onde estão presentes as variáveis como o tipo, nome, array de parâmetros, e número de parâmetros, bem como um ponteiro para a estrutura do nó da tabela para associar essa informação a um dado parâmetro ou método.

Quanto aos algoritmos, foram implementadas diversas funções de verificação para os diversos símbolos terminais da gramática realizada no ficheiro yacc como o Program, FieldDecl, MethodDecl, entre outros ou para a AST, de modo a serem úteis para a implementação da AST anotada. Para a realização da tabela de símbolos também foram implementadas funções de inserção e procura de elementos na tabela, assim como funções de inicialização, tanto da classe como dos métodos presentes nesta e uma função para imprimir a tabela de símbolos da forma que é referido no enunciado.