Relatório Compiladores

Leonor Coelho, 2017254561 João Cardoso, 2017247458

2019/2020

1 Gramática

A gramática foi construída com base no enunciado, transformando tudo o que é ou opcional ,ou de zero ou mais repetições, em novos estados ou produções.

O que tem zero ou mais repetições tem novas produções associadas, como é o caso da primeira produção *Program* que, para distinguir entre *MethodDecl*, *FieldDecl*, SEMICOLON e *%empty* avança para uma nova produção *Program1*. Tudo o que seja declarações de variáveis ou de parâmetros separados por vírgulas tem uma produção à parte nomeadamente *CommaId* (para *FieldDecl*'s e *VarDecl*'s), *CommaTypeId* (para declarações de parâmetros) e *CommaExpr* (para *Expr*'s).

Também têm a sua própria produção *Statement*'s que possam ser *Expr*'s como são os casos *Assignment*, *MethodInvocation* e *ParseArgs*.

Nas *Statement*'s, são apresentadas 2 produções que causam um conflito de Shift/reduce:

- IF LPAR Expr RPAR Statement
- IF LPAR Expr RPAR Statement ELSE Statement

Quando o token ELSE é lido e se torna o token de lookahead, o parser está pronto para reduzir à primeira regra, mas também é possível fazer *shift* do ELSE, pois iria levar eventualmente, a uma redução através da segunda regra. Ou seja, no caso:

- if(ola) X = 1; if(adeus) x = 2; else x= -1;

O ELSE pode estar associado ao primeiro ou ao segundo IF. Para evitar este problema, declaramos um token REDUCE e o ELSE com *%nonassoc*, e utilizamos o *%prec* para dar prioridade ao IF sem ELSE, ou seja, no caso acima, o ELSE estaria associado ao primeiro IF.

As expressões causavam, igualmente, diversos conflitos de *shift/reduce* (como no caso acima). Para resolver os conflitos, e com auxílio da documentação disponibilizada, reescrevemos as produções e adicionamos outras 2:

- O *Assignment* é a operação que tem menos prioridade sobre as outras, logo, isolamos o mesmo numa produção (*Expr*).
- De seguida, vem uma produção que contém todas as operações. A linguagem Juc não permite que exista um *Assignment* antes ou depois dos operadores, logo, a expressão depois do operador apenas pode ser *Expr1* ou *Expr2*.
- Depois de um operador unitário, apenas pode vir um símbolo terminal, uma invocação de método, um *ParseArgs* ou uma Expressão dentro de parêntesis, não é permitido conter operadores na sua expressão, daí ter sido criada a produção *Expr2*.
- Todos os operadores têm precedência à esquerda, à exceção do NOT e do ASSIGN que reduzem primeiro as Expressões à direita e só depois as da esquerda.

2 Algoritmos e estruturas de dados da AST e da tabela de símbolos

Para conseguirmos reportar eventuais erros nesta parte da análise, tornou-se necessário conhecer a localização exata de cada token no ficheiro com o código fonte. Para isto utilizámos a estrutura do yacc, yylloc, que permite passar os valores da localização dos tokens de uma maneira fácil entre ficheiros lex e yacc.

2.1 Árvore Sintática Abstrata

Para o desenvolvimento da AST foram criadas 23 estruturas, declaradas no ficheiro *structures.h*, para acomodar todas as categorias da gramática, e funções de criação e inserção de nós na árvore, definidas no ficheiro *functions.c*.

Para o caso das *Statements*, e das *Expressions*, para conseguirmos abranger todos os tipos que estas duas categorias têm, usámos **unions**, desta maneira conseguimos ter apenas numa estrutura um ponteiro para cada tipo de statement ou expression que precisemos de adicionar ou aceder.

No que toca à criação da AST temos dois tipos de funções:

- **criação**: criar nós que mais tarde vão ser inseridos na árvore;
- **inserção**: inserir nós diretamente na árvore;

As funções de criação são utilizadas em maior parte no *Assignment*, no *ParseArgs* e nas *Calls*, pois estes nós podem ser inseridos como *Statement* ou como *Expression*. Assim, criamos primeiro uma estrutura, onde guardamos os valores necessários, que depois é passada a uma função de inserção que insere a estrutura na árvore como *Statement* ou como *Expression*.

No entanto nem todas as funções de inserção inserem diretamente na árvore. Aplica-se uma excepção às produções *CommaId*, *CommaTypeId* e *CommaExpr* que recolhem tudo o que for inserido e criam uma lista ligada, com inserção à cabeça, que depois é usada na criação dos nós que chamam estas produções.

No final do programa toda a memória alocada para a construção da árvore é libertada pela função cleanTree.

2.2 Tabela de símbolos

Para a construção desta tabela temos um conjunto de funções nos ficheiros *semantics.c* e *symbol_ table.c* que vão procurar a existência de erros semânticos aquando da inserção dos símbolos na tabela.

A tabela de símbolos é constituída por 3 tipos de estruturas: *TableHead*, *TableElement* e *MethodElement*. A primeira serve apenas para guardar a cabeça da tabela e contêm o nome da classe e um ponteiro para uma lista ligada de *TableElements*. Estes elementos podem ser *FieldDeclarations* ou *MethodDeclarations* e sendo estes do segundo tipo contêm um ponteiro para uma lista ligada de MethodElement's que guardam as declarações de parâmetros e as declarações de variáveis locais.

2.3 Árvore de sintaxe anotada

Para a anotação dos nós relativos a *Expressions*, percorremos a árvore sintática antes de imprimirmos a mesma, tendo em conta a documentação da linguagem Juc. Caso algum id de função não seja entrado na tabela de símbolos, não é anotado qualquer nó descendente a este. O mesmo acontece com os nós *Lshift*, *Rshift* e *Xor*.

As funções correspondes à anotação da árvore encontram-se no ficheiro *semantics.c*.

3 Geração de código

Meta não desenvolvida.