FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Compiladores
2022/2023

Compilador para a linguagem Juc

Trabalho elaborado por:
-Bruno Sequeira nº 2020235721

-Rui Santos nº 2020225542

Objetivos

Este projeto consiste em desenvolver um compilador para a linguagem Juc, esta que é um subconjunto da linguagem Java de acordo com a especificação Java SE 9.

Para realização deste projeto, foi-nos pedido a realização em 4 metas:

1. Análise Lexical
2. Análise Sintática
3. Análise Semântica
4. Geração de Código.

Das 4 metas, realizamos com sucesso a meta 1, a meta 2, e metade da meta 3. A meta 4 não conseguimos realizar a tempo.

Para este relatório foi nos pedido para documentar concisamente as opções técnicas relativas:

- à gramática re-escrita,
- aos algoritmos e estruturas de dados da AST e da tabela de símbolos e,
- à geração de código (sendo que este não poderemos documentar porque não desenvolvemos a meta4)

Secção 1 - Gramática re-escrita

Na meta 2 foi realizada a análise sintática, sendo que foi feita com base numa gramática ambígua inicial em notação EBNF, que foi rescrita para yacc de modo a fazer uma ligação com a análise lexical realizada na meta 1.

Para rescrição da gramática, seguimos os princípios básicos de yacc, mas existiam algumas restrições como por exemplo, regras de associação dos operadores e precedências.

Visto que na gramática inicial, existiam símbolos que podiam aparecer zero ou mais vezes, (na notação EBNF {...} representa zero ou mais vezes, [...] representa que é opcional).
Esses problemas foram contornados pelo yacc através de recursividade à direita, criando assim um símbolo terminar auxiliar, permitindo assim ser possível que estes símbolos apareçam 0 ou mais vezes ({...}), ou ser opção aparecer ([...]).

Mostramos agora um exemplo de casos:

1. {...}

Program ⟶ CLASS ID LBRACE { MethodDecl | FieldDecl | SEMICOLON } RBRACE

Este foi contornado da seguinte forma:

```
Program:        CLASS ID LBRACE declaration RBRACE
```

```
declaration:
                        MethodDecl declaration
                |       FieldDecl declaration
                |       SEMICOLON declaration
                |       /* null */
```

2. [...]

Statement ⟶ [(MethodInvocation | Assignment | ParseArgs)] SEMICOLON

Este foi contornado da seguinte forma:

```
Statement:
                |       MethodInvocation SEMICOLON
                |       Assignment      SEMICOLON
                |       ParseArgs SEMICOLON
                |       SEMICOLON
```

Neste exemplo, temos os parênteses que significa que o que está dentro dele, só pode aparecer um símbolo, neste caso, ou “MethodInvocation” ou “Assignment” ou “ParseArgs”.

Tivemos também de ter em conta as precedências para evitar que se criem conflitos entre símbolos não terminais. Para isso, usamos o 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
```

Com estas precedências conseguimos resolver conflitos respetivamente às operações. Um exemplo que podemos documentar é a multiplicação terá sempre prioridade em relação à adição e à subtração.

Nos Statements, são apresentadas 2 produções, que podem causar um conflito de Shift/Reduce:

```
IF LPAR Expr RPAR Statement %prec  ELSE
```

```
IF LPAR Expr RPAR Statement ELSE Statement
```

Quando se lê o ELSE, este torna-se o token de lookahead, sendo que está pronto a pronto a reduzir à primeira regra, como também é possível fazer shift do ELSE levando a uma redução através da segunda regra. Um exemplo seria:

- ☐ if(Teste1) a = 3; if(Teste2) a = 3; else a = 20;

O else tanto pode estar associado ao primeiro ou ao segundo “if”. Para evitar esse conflito utilizamos o %prec para dar prioridade ao IF sem ELSE, ou seja, no caso acima, o ELE estaria associado ao primeiro “if”.

A criação da produção Expr2 deu-se pelo facto de que após um operador unitário, apenas pode vir um símbolo terminal, uma invocação de um método, um ParseArgs ou uma Expressão dentro de parênteses, não sendo permitido conter operadores na sua expressão.

Todos os operadores têm precedência à esquerda, à exceção do NOT, ASSIGN e do ELSE, sendo que reduzem em primeiro as Expressões à direita e só depois as da esquerda.

Secção 2 - algoritmos e estruturas de dados da AST e da tabela de símbolos

Esta secção corresponde às metas 2 e 3 onde nos foi pedido para fazer a análise semântica e sintática e desenvolver uma árvore de sintaxe e uma tabela de símbolos

Para o desenvolvimento da AST foi criada uma struct node (definida em functions.h) que vai ser uma estrutura comum a todos os tipos de nó. Além disso, foram criadas funções para criar e inserir nós na árvore outras auxiliares (definidas em funcions.c). Para criar um novo nó usamos a função CriaNo() onde temos então de indicar o nó, o valor do nó (ambos podem ser NULL/"" em alguns casos) e por último o tipo de nó (MethodBody, ParamDecl, Id,...). Depois adicionamos um novo nó ao nó criado(será o primeiro filho) com a função AdicionaNo() e, se necessário adicionar mais algum filho usamos a função AdicionaIrmao().

Para a criação da tabela de símbolos usamos duas estruturas (definidas em symbol_table.h), uma representa os elementos da tabela de símbolos, onde estão definidos todos os elementos que vão ser impressos tal como o nome, o tipo, os parâmetros(e os seus tipos) e outra representa a tabela de símbolos onde inclui o nome, os parâmetros(e número de parâmetros) . Para preencher a tabela de símbolos usamos muitas funções definidas em semantics.c e em symbol_table.c que utilizamos para inicialização e inserção na tabela, procurar elementos na tabela, determinar erros e algumas auxiliares.

De forma a preencher a tabela de símbolos e anotar a AST, chamamos a função verifica() onde fazemos uma primeira passagem pela árvore onde

criamos todas as tabelas de símbolos e adicionamos à tabela o return type e os parâmetros que recebem e depois fazemos uma sequência de passagens usando a função check_ast() onde procedemos tanto ao preenchimento das tabelas de símbolos, como à adição de anotações na AST, criando uma árvore de sintaxe anotada.

Secção 3 - algoritmos e estruturas de dados da AST e da tabela de símbolos

Não realizamos esta secção.