Departamento de Engenharia Informática
Faculdade de Ciência e Tecnologia da Universidade de Coimbra

Compiladores
2019/2020

Projeto Prático

Ricardo Tavares – 2014230130 – uc2014230130@student.uc.pt

Gramática

O analisador semântico “jucompiler.y” especifica a sintaxe da linguagem Juc, lendo os tokens fornecidos pelo analisador “jucompiler.l”. A gramática está descrita no analisador semântico, resolvendo os problemas de ambiguidade existentes na notação inicial fornecida.

Para resolver essa ambiguidade é especificada a precedência dos operadores e a sua prioridade:

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

(1)

É necessário definir a associatividade dos operadores usando “%left” e “%right” para o yacc saber como resolver expressões como 5 + 5 – 9, que por ex. pode resultar em (5+5) - 9 ou 5 + (5-9).

A keyword “%prec” é também utilizada para mudar a precedência da gramática para ficar igual à do token que a segue. Foi necessário recorrer a ela para resolver conflitos com os operadores MINUS NOT E PLUS.

A prioridade dos operadores está listada de ordem do que tem menor prioridade para o com maior prioridade.

Referências

(1) Definição baseada em https://introcs.cs.princeton.edu/java/11precedence/

Estruturas

As estruturas estão definidas nos ficheiros “jucompiler.l” “node.h” e “symtable.h”. Para compilar o código foi criado um script “script.sh” com as flags e “linkagem” necessárias.

Position:

Estrutura auxiliar para o lex, guarda a posição para a impressão dos erros.

```
struct Position {
        int column;
        int line;
    };
struct Position commentTracker;
struct Position stringTracker;
```

%union:

Estrutura utilizada para fazer a passagem dos tokens entre o lex e o yacc.

```
%union{
    char* string;
    struct node* node;
}
```

Node:

Estrutura usada para a criação dos nós que compõem a AST.

```
typedef struct node{
    NodeType type;
    struct node* father;
    struct node* brother;
    struct node* child;
    char* value;
    SymbolType primitiveType;
}Node;
```

O type é uma enum com todos os tipos de nós, sendo usado depois um conversor para os passar para char* na impressão da árvore.

Tem 3 ponteiros para os nós pai/filho/irmão.

O campo value guarda o valor do nó caso seja um nó folha (terminal).

O primitiveType foi adicionado na meta 3 para guardar o tipo primitivo do nó, é também uma enum usada na tabela de símbolos.

Symbol:

Estrutura que representa cada símbolo da tabela.

```
typedef struct symbol{
    char* name;
    SymbolType symType;
    int varMethod; // 0 = var 1 = method
    int isParam;
    struct symbol* brother;
    struct table* tablePointer;
} Symbol;
```

O name vai guardar o valor do nó da AST.
O symType é uma enum com os tipos primitivos.
varMethod guarda se o símbolo é uma variável ou método para alterar a forma como se imprime a tabela de símbolos.
isParam é usado para o mesmo propósito do campo varMethod, indica se o símbolo representa um parâmetro de uma função.
A estrutura symbol possui um ponteiro para o seu irmão e também um ponteiro para uma tabela de símbolos, definido se for uma declaração de métodos.

Table:

Estrutura que representa a tabela de símbolos.

```
typedef struct table{
    char* name;
    int type; // 0 = class 1 = method
    struct table* brother;
    struct symbol* child;
} Table;
```

O name guarda o nome da tabela (Class ou Method) e o type identifica-o.
Tem um ponteiro para a sua tabela irmã e o seu filho.

A criação destas estruturas é feita através dos nós da AST, normalmente pegando num nó e apontando para o filho, percorrendo os seus irmãos.