FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Departamento de Engenharia Informática

Projeto de Compiladores

2019/2020 - 2.º Semestre

Compilador para a linguagem Juc

jucompiler

The best compiler for simple java programs

Sérgio Manuel Carvas Machado 2017265620

Renato Miguel Francisco de Matos 2015257602

Índice

1 - Gramática re-escrita

Fizemos algumas alterações à gramática inicial indicada no enunciado de modo a remover a sua ambiguidade e tratar das situações com produções **opcionais** ou com **zero ou mais repetições**.

Nos casos em que uma símbolo da produção seja opcional, como por exemplo em

Statement → RETURN [Expr] SEMICOLON

dividimos em duas produções diferentes. Neste caso

Statement → RETURN Expr SEMICOLON

e

Statement → RETURN SEMICOLON

Já nos casos em que existam produções com várias repetições, como em **Program → CLASS ID LBRACE { MethodDecl | FieldDecl | SEMICOLON } RBRACE** são várias novas produções, em que o símbolo a partir do qual esta resulta está presente em ambos os lados da produção. Assim, esta acaba por se comportar como uma lista que podem conter um, vários ou nenhum dos símbolos de entre aqueles que poderiam ser repetidos ou nem sequer aparecer. Assim, com base no caso dado, a nossa solução foi a seguinte

Declarations → EPSILON
Declarations → Declarations MethodDecl
Declarations → Declarations FieldDecl
Declarations → Declarations SEMICOLON

Para a Expr, devido a conflitos *shift reduce*, decidimos separar em duas produções diferentes, o que resolveu o conflito:

Expr -> Expr2
Expr -> Assignment

Deste modo, as restantes produções em que do lado esquerdo tinham Expr passam a ser geradas a partir de Expr.

Estas foram as precedências que definimos, com base na documentação da linguagem Java:

%right ELSE
%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
%nonassoc LPAR RPAR LSQ RSQ

Há casos específicos em que, apesar de indicarmos que um símbolo tem precedência sobre outro, precisamos que tal não se verifique em determinada regra de uma produção. Para isso, a seguir à produção, indicamos %prec seguido do símbolo pretendido. Tal resulta em tanto a regra quanto o símbolo indicado terem a mesma precedência.

Usamos este método no Statement, em

IF LPAR Expr RPAR Statement %prec ELSE

e nas Expressões (Exp2), em

MINUS Expr2 %prec NOT

e

PLUS Expr2 %prec NOT

Também foram acrescentadas produções com o símbolo terminal **error** com o intuito de haver produções de recuperação caso houvesse um erro na análise semântica e assim evitar uma paragem no parsing.

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

2.1 - AST

Durante a Meta 3 foi necessário rever a forma como os dados dos tokens passavam do LEX para o YACC, devido a ser necessário apresentar corretamente a linha e coluna destes nos erros semânticos (da forma como tínhamos anteriormente, a coluna era sempre a do último token da produção da gramática). Tentando manter o máximo inalterado, de forma a não criar novos bugs, passámos a usar a seguinte estrutura de forma a passar os dados (id, linha e coluna) dos tokens em que isso representava um problema (fomos testando para ver quando isso ocorria):

```
typedef struct identifier{
	char* id;
	int line;
	int col;
}Identifier;
```

Inicialmente tentámos adaptar as estruturas de dados da AST presentes nas fichas práticas ao nosso projeto (com listas ligadas). No entanto encontrámos algumas dificuldades, tendo optado por estruturas que nos pareceram mais fáceis de implementar:

```
typedef struct ASTREE* AST_pointer;
typedef struct ASTREE{
   char * nome;
   char* valor;
   char* anotar;
   char* return_id;
   int is_call;
   int linha, coluna;
   AST_pointer pai;
   AST_pointer filho;
   AST_pointer irmao;
}ASTREE;
```

Todos os nós da AST foram implementados segundo esta estrutura. Assim, cada nó tem um ponteiro para o nó pai, filho e irmão, e várias variáveis onde armazena os dados

necessários (nome, valor, anotar (anotação), assim como flags e número de linha e coluna, necessários para uma correta implementação).

No que toca a algoritmos, o facto de as estruturas pelas quais optámos serem mais genéricas facilitou-nos um pouco a implementação, principalmente na criação da AST. De forma muito resumida, registando-se a produção “Program” é criada a raíz da árvore. Os restantes nós são criados à medida que as produções vão sendo verificadas e são adicionadas à árvore com recurso a funções que indicam a relação dos nós uns com os outros (**irmão** ou **filho**).

2.2 - Tabela de símbolos

As tabelas de símbolos foram criadas segundo uma lista ligada de estruturas **Table**, com cada uma dessas estruturas a conter o seu nome (de forma a poder identificá-la tanto quando se pretende verificar se já existe uma tabela com aquele nome, quanto para o header quando se pretender imprimir as tabelas), uma lista ligada dos símbolos correspondentes (incluindo ponteiro para último adicionado) e ponteiros para a próxima tabela e a última adicionada.

O primeiro elemento dessa lista corresponde à tabela global (contendo os parâmetros globais (da classe) e declaração de funções e variáveis globais). O resto dos elementos dessa lista contêm os símbolos declarados dentro de cada uma das funções. Deste modo, consegue-se diferenciar o scope dos símbolos declarados nos vários métodos.

No que toca aos símbolos, estes podem ser variáveis declaradas, funções (métodos) ou argumentos da função, e contêm os expectáveis campos de nome, tipo e linha e coluna. Os parâmetros têm também uma estrutura própria, que também funciona como lista ligada.

```
//Tipo de parametros
typedef struct _param* param_pointer;
typedef struct _param{
   char *tipo;
   char *nome;
   param_pointer next;
   param_pointer last_added;
}Param;
```

```
//Simbolo
typedef struct _symbol* symbol_pointer;
typedef struct _symbol{
   char* nome;
   char* tipo;
   int line, col;
   int data_type;  //0 - VAR, 1 - function, 2 - param
```

```
   param_pointer params;
   symbol_pointer next;
}Symbol;
```

```
//Tabela com nome para definir o tipo
typedef struct _table * table_pointer;
typedef struct _table{
   char* nome;
   symbol_pointer next_sym;
   symbol_pointer last_added;
   table_pointer next;
   table_pointer last_table;
}Table;
```

A anotação da AST e criação das tabelas de símbolos são feitas através do mesmo algoritmo, tal como os erros semânticos. A AST é percorrida duas vezes, uma primeira para a criação da tabela de símbolos globais e das tabelas de símbolos de cada método, e uma segunda para anotar a árvore.

No entanto, esta abordagem pode ter dificultado a tarefa de tratamento de erros no caso de símbolos (e métodos) repetidos, por já termos de percorrer a árvore mais que uma vez. Ainda tentámos implementar uma solução, mas acabámos por não conseguir resolver este problema.

3 - Geração de código

Acabámos por não conseguir dedicar-nos à Meta 4, principalmente devido a falta de tempo e por nos termos alongado na Meta 3. Ficando pelo início, mesmo assim já estávamos a ter alguns outputs corretos no que toca a declarações de variáveis e alguns cálculos com elas. Assim, mantivemos aquilo que implementámos nesta parte do projecto (**code_generator.c**) na entrega final do trabalho, apesar de o seu código não ser corrido.