Compilador DeiGo

UNIVERSIDADE D
COIMBRA

Compiladores 2021/2022
Licenciatura em Engenharia Informática

João Filipe Guiomar Artur, 2019217853
Sancho Amaral Simões, 2019217590

18 de dezembro de 2021

Índice

1 Introdução

No contexto da parte prática desta unidade curricular, foi proposto o desenvolvimento de um compilador para a linguagem DeiGo, uma variante da linguagem de programação Go. O percurso de desenvolvimento encontra-se divido em quatro etapas, sendo elas Análise Lexical (meta 1), Análise Sintática (meta 2), Análise Semântica (meta 3) e Geração de Código (meta 4).

O presente relatório tem como objetivo explicar as decisões referentes à rescrita da gramática, estrutura da árvore de sintaxe abstrata (AST), inseridas no contexto da etapa de Análise Sintática, e a estrutura das tabelas de símbolos, construídas durante a meta da Análise Semântica. Por não ter sido alcançada, a etapa correspondente à Geração de Código não será abordada.

2 Gramática reescrita

Como ponto de partida para esta etapa, foi fornecida, em notação EBNF, uma gramática ambígua que descreve a linguagem DeiGo. O objetivo foi reescrever a gramática inicial em suporte Yacc, permitindo, assim, a integração da análise lexical efetuada na primeira etapa.

A reescrita da gramática involveu contornar alguns aspetos da notação EBNF, nomeadamente a existência de símbolos, terminais ou não terminais, que podiam ocorrer zero ou mais vezes (assinalados como {...} na notação referida). Para este efeito, foram criadas novas regras que permitiam obter os seguintes cenários: (i) zero repetições; (ii) uma repetição; (iii) duas ou mais repetições, tendo sido adotada recursividade à direita nesta última regra.

Como exemplo ilustrativo, considerem-se as seguintes regras:

```
Program:        PACKAGE ID SEMICOLON Declarations
Declarations:   {VarDeclaration SEMICOLON | FuncDeclaration SEMICOLON}
```

que ficam:

```
Program:        PACKAGE ID SEMICOLON Declarations | PACKAGE ID SEMICOLON
Declarations:   VarDeclaration SEMICOLON
                | VarDeclaration SEMICOLON Declarations
                | FuncDeclaration SEMICOLON
                | FuncDeclaration SEMICOLON Declarations
```

Outro aspeto a salientar na notação EBNF é a existência de símbolos terminais ou não terminais opcionais (assinalados como [...]) e símbolos terminais ou não terminais em que apenas se pode escolher um deles (assinalados como (...)). À semelhança da situação descrita anteriormente, optou-se por criar novas regras que representem todas as possibilidades.

Exemplificando, considere-se a seguinte regra e a sua reescrição:

```
Statement:      RETURN [Expr]
```

```
Statement:      RETURN | RETURN Expr
```

Destaque-se a existência de duas exceções em que, devido à complexidade de conceber regras, seguindo a filosofia até aqui descrita, que traduzam corretamente os resultados esperados, se recorreu a regras vazias (assinaladas como epsilon).

Dada a ambiguidade da gramática inicial, foi necessária a definição da precedência dos operadores. No yacc, as precendências funcionam de baixo para cima, isto é, os tokens (neste caso, os operadores) com maior precedência estão definidos em baixo e os com menor em cima. Além disso, right e left definem se a precedência é calculada a partir da direita ou da esquerda, respetivamente.

As preferências do operadores definidas, por ordem, foram:

- %right ASSIGN (atribuição)
- %left OR (Or lógico)
- %left AND (And lógico)
- %left LT (menor que), GT (maior que), LE (menor ou igual), GE (maior ou igual), EQ (igual), NE (diferente)
- %left PLUS (mais), MINUS (menos)
- %left MULT (multiplicação), DIV (divisão), MOD (módulo)
- %right NOT (negação)

Em algumas regras foi usado o operador %prec para modificar localmente a precedência de um determinado operador. Verifica-se este cenário quando os operadores + e - são aplicados, de forma unária, a um token numérico.

```
Expr:           NOT Expr
                | MINUS Expr %prec NOT
                | PLUS Expr %prec NOT
```

No exemplo acima, os operadores + e - são unários, em vez de binários, pelo que têm a sua precedência modificada localmente para mesma que o operador !, isto é, passam a ter precedência unária.

3 Algoritmos e estruturas de dados

3.1 Árvore de sintaxe abstracta (AST)

Na segunda etapa, foi construída uma árvore de sintaxe abstracta, sendo anotada posteriormente na terceira meta, correspondendo à Análise Sintática e Análise Semântica, respetivamente.

Para que fosse possível a integração da análise lexical e a passagem dos dados obtidos durante a mesma, recorreu-se a uma estrutura *union*, onde são armazenados o valor do ID do *token*, a linha e a coluna.

Na arquitetura da AST são considerados dois tipos de nós: (i) nó da AST; (ii) nó de lista ligada. Um nó da AST é constituído pelos seus atributos (ID, anotação, linha e coluna), *flags* para efeitos de deteção de erros e por duas listas ligadas ligadas, correspondentes aos filhos e aos irmãos do nó. Por sua vez, o segundo tipo de nó contém um nó da AST e a referência para o próximo nó da lista ligada.

Durante a construção da árvore, dada a sua natureza *bottom-up*, um nó criado é guardado no topo da pilha e quaisquer outros nós criados na mesma regra são adicionados aos seus irmãos. Esta construção depende da inicialização dos atributos do nó criado, que depois é inserido nos filhos ou irmãos do nó que esteja no topo da pilha.

```
ID Type Parameters_1                                {$$ = create_node(0, 0, A_PARAM_DECL, "ParamDecl");
                                                    push($$->children, $2);
                                                    push($$->children, create_node($1->line, $1->column, A_ID, $1->id));
                                                    push($$->siblings, $3);}
```

3.2 Tabelas de símbolos

Existem duas categorias de tabelas de símbolos, tabela global e tabela local (representativa de uma função), sendo cada uma constituída pelo nome e por uma lista ligada de entradas e, no caso da segunda categoria, uma entrada especial, que contém o tipo de dados devolvido.

Cada entrada global pode conter uma variável, em conjunto com os seus atributos (nome e tipo/classe), ou uma tabela local, isto é, a tabela de símbolos de uma função. A nível local, uma entrada pode representar um parâmetro da função ou uma variável local e os respetivos atributos (nome, tipo/classe e distinção entre parâmetro ou variável). No caso de ser uma variável, a entrada tem ainda um ponteiro para a zona da AST onde a variável foi declarada.

A construção das tabelas é feita em duas etapas, uma para a construção da tabela global e uma segunda para a construção das várias tabelas locais. Assim, permite-se que, dentro de uma dada função, sejam invocadas quaisquer variáveis globais ou funções, independentemente da posição relativa da sua declaração.

Na primeira passagem são criadas entradas para todas as variáveis globais e inicializadas as tabelas locais. A inicialização compreende a definição do nome, do tipo de dados devolvido e os parâmetros da função. Na segunda passagem, são completadas as tabelas locais com entradas de variáveis declaradas localmente.