UNIVERSIDADE D
COIMBRA

Projeto Compiladores

Uccompiler

Trabalho realizado por:
Marcelo Rodrigues Gomes nº2021222994
Pedro Miguel Brites Santos nº2021226319

Índice

Gramática Re-escrita

Começámos por analisar a gramática fornecida em notação EBNF e de seguida reescrevê-la em yacc, adicionando também os tokens necessários e os returns correspondentes no lex. Ao escrever a gramática em yacc, obtemos um grande número de conflitos “reduce-reduce” e “shift-reduce”. Assim, definimos regras que estabelecem a associatividade e precedência dos operadoes de forma a remover as ambiguidades da gramática. Tendo os operadores mais abaixo maior prioridade.

```
%nonassoc LOW
%nonassoc ELSE
%left COMMA
%right ASSIGN
%left OR
%left AND
%left BITWISEOR
%left BITWISEXOR
%left BITWISEAND
%left EQ NE
%left LE GE LT GT
%left PLUS MINUS
%left MUL DIV MOD
%left HIGH
%right NOT
```

Ainda fizemos algumas alterções na gramática para as listas e erros e também acrescentámos algumas regras tendo em conta o “opcional” e “zero ou mais repetições”.

```
DeclarationList:
Declarator
| DeclarationList COMMA Declarator
;

IfStatement: IF LPAR Expr RPAR Statement %prec LOW
| IF LPAR Expr RPAR Statement ELSE Statement
;

StatementList: StatementError
| StatementList StatementError
;

StatementError: error SEMI
| Statement
;

ExprList: Expr %prec HIGH
| ExprList COMMA Expr}
;
```

Algoritmo e Estruturas de dados

Structs usadas para a ast:

```
struct node {
    enum category category;
    char *token;
    int token_line, token_column;
    enum type type;
    struct node_list *children;
    int isExpr;
    char *p;
};
```

category: categoria do node
token: nome(token) associado ao node
children: lista dos filhos do node
token_line e token_column: usados para os erros semânticos
type: tipo do node
isExpr: variável que decide se o node deve ser anotado
p: parâmetros do node

```
struct node_list {
    struct node *node;
    struct node_list *next;
};
```

node_list: usada para guardar todos os nodes ligados, para depois poder percorrer

Structs usadas para as tabelas de símbolos:

```
struct symbol_list {
    char *identifier;
    enum type type;
    struct node *node;
    char *func_parameters;
    struct symbol_list *next;
};
struct symbol_list_list {
     char func_name[50];
     struct symbol_list *symbol_list;
     struct symbol_list_list *next;
};
```

A symbol list list é uma lista ligada de listas ligadas usada para a criação e ligação das tabelas de funções

Algoritmos e Estruturas de dados

Na análise semântica começamos por uma tabela global que armazena todas as declarações de variáveis globais e funções. Quando uma função é definida corretamente, é criada uma uma nova lista ligada à anterior com primeiramente o tipo de retorno, de seguida os parâmetros da função e por último, as variáveis criadas dentro dela. Para darmos print às tabelas pela ordem de declaração mesmo sendo definida depois, fizemos uma função que as mete na ordem comparando com um array de strings com os nomes das funções na ordem certa.

O algoritmo para decidir se um node é anotado é apenas verificar se os símbolos estão dentro de um *func body* (excluindo declarations e statements que não são expressions) alterando assim a variável isBody. Seguidamente, através da variável node→isExpr, todos os síbolos que sejam expressions serão anotados na função show da ast.

Quanto aos erros, fomos à gramática no yacc para poder retornar a linha e coluna dos nodes com símbolos terminais. Depois programámos os erros e foram colocados onde fazia sentido, como quando era criada uma variável void ou um símbolo já estava definido antes, apesar de as colunas ficarem todas com valores diferentes vindo do mesmo sítio e nas mesmas circunstâncias.