Judson Santos Santiago

Exercícios e Revisão

Compiladores

Exercício

1. Escreva uma gramática para representar declarações de classes, como a do exemplo abaixo:

```
class Ponto
{
private:
    int x;
    int y;
public:
    Ponto();
};
```

Solução

```
class Ponto
{
private:
    int x;
    int y;
public:
    Ponto();
};
```

```
program     -> class id { class_decls };

class_decls -> class_decl class_decls
             | ϵ

class_decl  -> access : decls

access      -> private
             | public
             | protected

decls       -> decl decls
             | ϵ

decl        -> type id;
             | id();
```

Exercício

2. Mostre como representar a precedência entre operadores lógicos e relacionais.

```
expr && expr
expr || expr
!expr

expr > expr
expr >= expr
expr < expr
expr <= expr
expr != expr
expr == expr
```

Exemplos:

```
x && y > 6

y > x > 8

x > 2 || !(y < 2) && y > z
```

Solução

```
expr &&
expr
expr ||
expr
!expr

expr > expr
expr >=
expr
expr < expr
expr <=
expr
expr !=
expr
expr ==
expr
```

```
log -> log &&
neg
     | log ||
neg
     | neg

neg -> ! rel
     | rel
```

```
rel -> rel > exp
     | rel >=
exp
     | rel < exp
     | rel <=
exp
     | rel !=
exp
     | rel ==
exp
     | exp

exp -> id
     | num
     | (log)
```

Exercício

3. Construa um esquema de tradução para converter expressões aritméticas da notação pós-fixada para a notação infixada.

```
expr -> expr expr +
      | expr expr -
      | expr expr *
      | expr expr /
      | digi

digi -> 0
      | 1
      | ...
      | 9
```

Solução

```
expr -> { print('('); } expr  { print('+'); } expr +
{ print(')'); }
      | { print('('); } expr  { print('-'); } expr -
{ print(')'); }
      | { print('('); } expr  { print('*'); } expr *
{ print(')'); }
      | { print('('); } expr  { print('/'); } expr /
{ print(')'); }
      | digi

digi -> 0 { print('0'); }
      | 1 { print('1'); }
      | ...
      | 9 { print('9'); }
```

Exercício

4. Remova a recursão à esquerda da gramática abaixo para a construção de um analisador sintático preditivo:

```
expr -> expr expr +
      | expr expr -
      | expr expr *
      | expr expr /
      | digi

digi -> 0
      | 1
      | ...
      | 9
```

Solução

```
expr -> expr expr +
      | expr expr -
      | expr expr *
      | expr expr /
      | digi

digi -> 0
      | 1
      | ...
      | 9
```

```
expr -> digit oper
oper -> expr + oper
      | expr - oper
      | expr * oper
      | expr / oper
      | ϵ

digi -> 0
      | 1
      | ...
      | 9
```

Resumo

- Um **processador de linguagem** pode ser obtido:
  1. Escrevendo uma gramática apropriada
  2. Criando um analisador léxico
  3. Criando um analisador sintático
     - Um analisador preditivo requer:
       - Remoção da recursão à esquerda
       - Garantia de que os conjuntos FIRST são disjuntos
  4. Criando uma tabela de símbolos
  5. Gerando código intermediário*