Teoria da Computação

Alunos: Fábio Melo, Luigge Lena, Thuane França, Elton Vinicius, Valber Moreira.

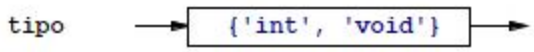
Projeto 2

Questionário

1. Quais tipos estão disponíveis na linguagem? Mostre a produção (ou produções) da gramática onde é possível ver isso.

Os tipos disponíveis são **“int”** e **“void”**, como mostrado nessa produção:

```
tipo    :       'int'
        |       'void'
        ;
```



2. É possível ter um bloco de comandos vazio? (ou seja, apenas as chaves { }). Justifique mostrando a produção da gramática que determina isso.

Sim, pois a chamada de ‘comando’ na regra função está encapsulada por um asterisco, que permite qualquer quantidade de execuções da regra comando.

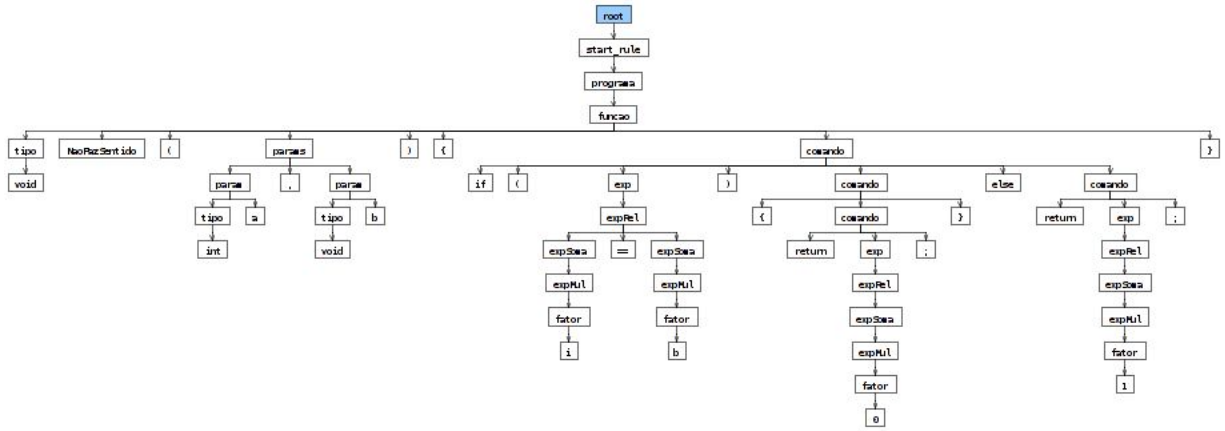
```
funcao :        tipo ID '(' params? ')' '{' comando* '}';
```

3. O analisador sintático de um compilador não faz verificação de tipos; isso é feito por etapas posteriores (em geral na análise semântica). Crie um exemplo de programa que está correto de acordo com a gramática mas cujos tipos não fazem sentido (usando as regras da linguagem C como base).

O Programa abaixo está correto, apesar dos tipos não fazerem sentido:

```
void NaoFazSentido(int a, void b){

if( a == b ){

return 0;} else return 1; }
```

Árvore de Execução:



4. Adicione a operação de subtração na gramática. Crie um programa que teste a nova operação e mostre a árvore de análise do programa testado.

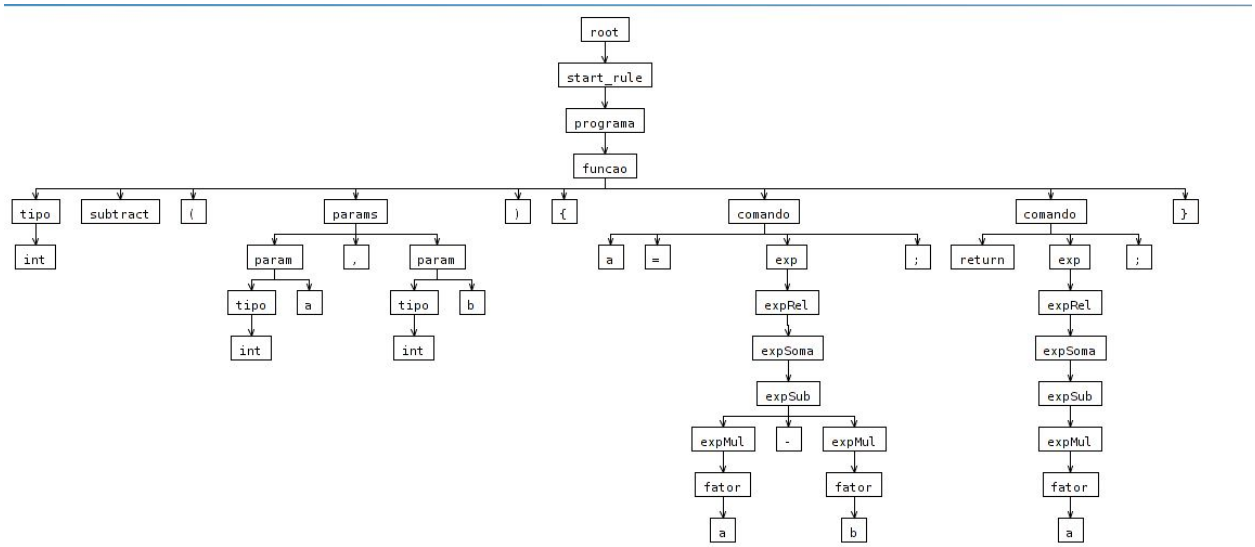
Para manter a ordem dos resultados, adicionamos a produção de subtração (expSub) entre as produções expSoma e expMul.

```
expRel  :       expSoma ('<' expSoma | '==' expSoma)*       ;

expSoma :       expSub ('+' expSub)*

        ;

expSub  :       expMul ('-' expMul)*  //função subtração

        ;

expMul  :       fator ('*' fator)*

         ;
```

para testar a produção, utilizamos este programa:

```
int Subtract(int a, int b){
a = a - b;
return a;
```

Cuja execução retorna a árvore abaixo:



5. Adicione a operação de divisão na gramática. Crie um programa que teste a nova operação e mostre a árvore de análise do programa testado.

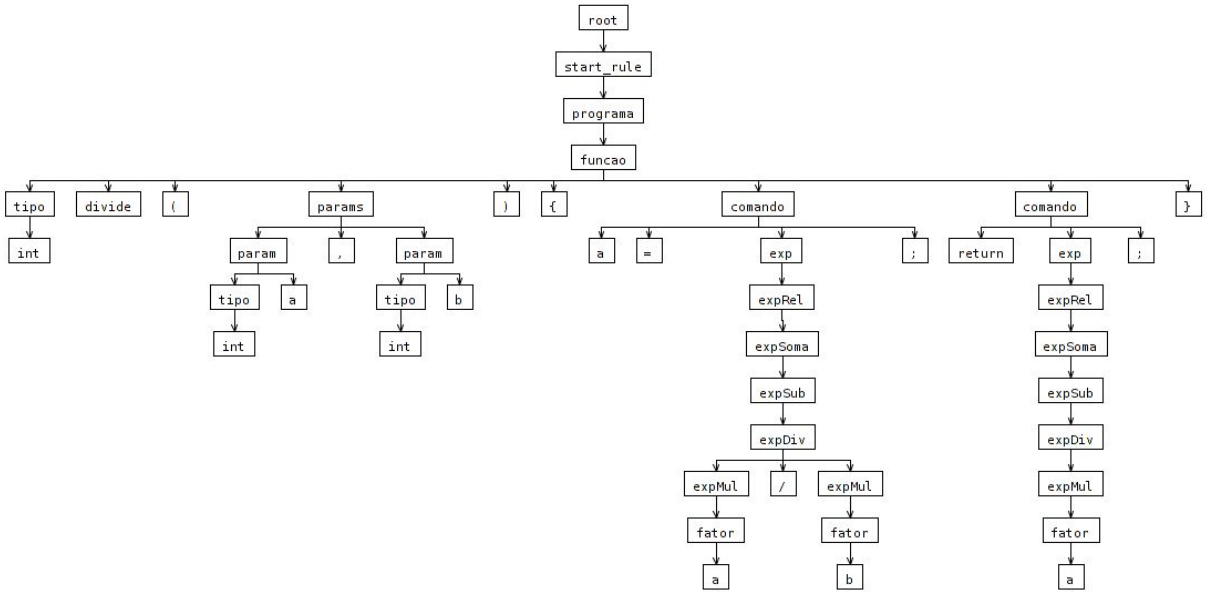
Adicionamos a produção de divisão expDiv, entre as produções expSub e expMul

```
expSub  :       expDiv ('-' expDiv)*
        ;
expDiv  :       expMul ('/' expMul)*
        ;
expMul  :       fator ('*' fator)*
        ;
```

Para testar a produção, utilizamos este programa:

```
int Divide(int a, int b){
a = a / b;
return a;
```

Na qual execução retorna a árvore abaixo:



6. Seguindo essa gramática, é possível definir funções com tipo de retorno void, mas não é possível chamar essas funções em um comando sozinho (por exemplo, uma chamada a printf, se a função printf estivesse definida). Isso acontece porque só existe chamada de função nas produções para expressões (não-terminal fator). Adicione uma produção de chamada de função ao não-terminal comando, permitindo a chamada de funções que não retornem nada. Crie um exemplo testando isso e mostre a árvore de análise do seu exemplo.

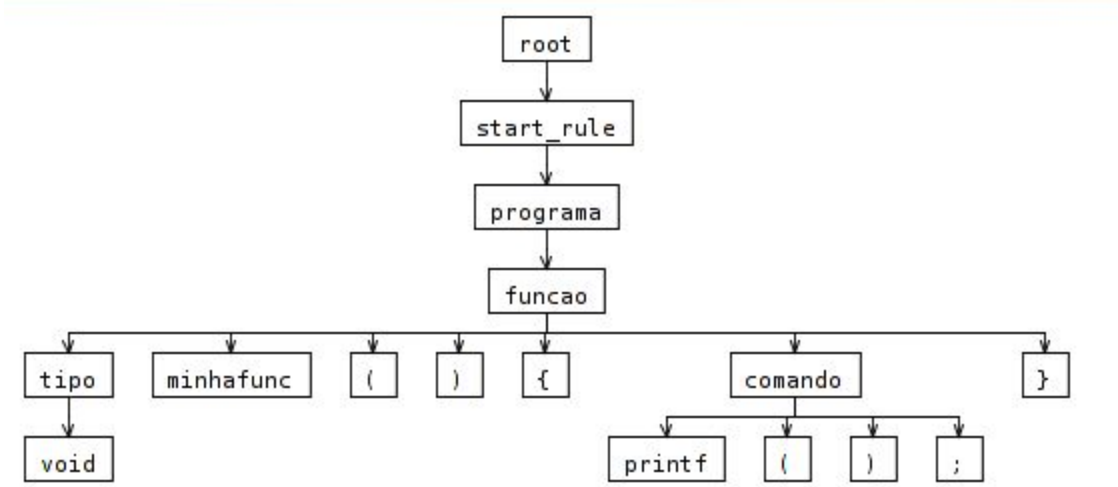
Para conseguirmos tal funcionalidade, adicionamos uma regra que permite a chamada de funções:

```
comando:      ID '=' exp ';'                          // atribuicao
        |     'return' exp ';'                        // return
        |     'if' '(' exp ')' comando 'else' comando  // condicional
        |     '{' comando+ '}'                        // bloco de comandos
        |     ID '(' params? ')'  ';'  // chamada de funcao
        ;
```

Função de Teste:

```
void minhafunc(){
printf();

}
```

Árvore retornada:



7. A gramática inicial não permite um if sem o else. Adicione uma produção que permite essa construção. Teste a nova gramática e mostre a árvore do programa de teste.

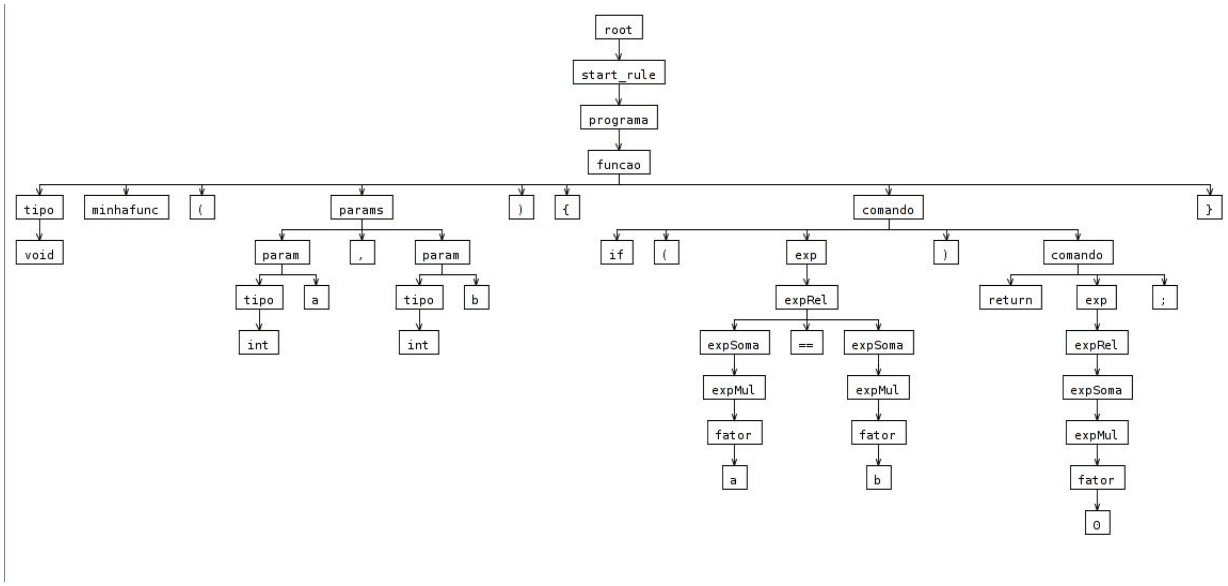
Para permitir que o ‘if’ execute sem o ‘else’, encapsulamos o terminal ‘else’ entre parênteses com o sinal de asterisco.

```
comando:      ID '=' exp ';'                          // atribuicao
        |     'return' exp ';'                        // return
        |     'if' '(' exp ')' comando ('else' comando)*  // condicional c/else encapsulado
        |     '{' comando+ '}'                        // bloco de comandos
        |     ID '(' params? ')'  ';'  // chamada de funcao
        ;
```

Código Testado

```
void minhafunc(int a, int b){
if (a == b)
return 0;
}
```

Árvore Resultante



8. A gramática inicial não tem operações lógicas como 'or' ou 'and'. Como podemos testar o OU lógico de duas condições sem ter o operador 'or'? Como testar o E lógico?

Com esta gramática, podemos testar os operadores OR e AND utilizando o retorno das funções das quais queremos comparar. por exemplo, uma operação de comparação que retorna 1, caso seja verdade, ou retorna 0 caso seja falso.

Partindo deste pressuposto, no caso do AND podemos montar um programa que verifica se o retorno de duas operações são iguais a 1, e do OR, um programa que verifica se alguma (ou ambas) das operações retornou 1.

Exemplo de código:

```
int meuAND(int a, int b){
if (a == b == 1){
return 1; } else return 0;
```

```
int meuOR(int a, int b){
if(a == 1){
return 1;} else if (b == 1) { return 1;}
else return 0;
```

exemplo de chamada de função:

```
meuOR(menorque(a, b), maiorque(c, d)) //verifica a primeira função OU a segunda função.
meuAND(igual(a,b), menorque(c,d) //verifica a primeira função E a segunda função.
// retornam 1 se verdadeiro, 0 se falso
```

9. Adicione os operadores || (OU lógico) e && (E lógico) na gramática. Isso deve ser feito com a criação de um novo não-terminal expLog, de precedência mais baixa que expRel. expLog deve chamar expRel, e na definição do não-terminal exp deve ser trocado expRel por expLog (ou seja, exp deve chamar expLog e não expRel como está no arquivo). Teste a nova gramática e mostre a árvore do teste.

Para conseguirmos tal resultado, criamos esta regra:

```
exp     :       expLog
        ;
expLog  :       expRel ('||' expRel | '&&' expRel)*
        ;
```

E a testamos com essa função:

```
void testAndOr(){
if (a == b || c == d && b == d)
return 0;
}
```

Resultando nesta árvore de teste: