Universidade Federal de Santa Catarina (UFSC)
Departamento de Informática e Estatística (INE)
INE5426 - Construção de Compiladores

Freedom-- :
Análises Léxica e Sintática

16104059 - Bruno Izaias Bonotto
16100725 - Fabíola Maria Kretzer
16105151 - João Vicente Souto

Florianópolis
2018

1. Descrição da Linguagem

A linguagem *Freedom--* (*Freedom Less Less* - Liberdade Menos Menos) é inspirada nas linguagens c e c++. No entanto, possui posicionamento pré-estabelecido do código, de modo que force o desenvolvedor seguir certas regras de padronização de código. Deve-se a essa característica de "engessamento" que definiu-se o nome da linguagem.

Esta linguagem possui operações aritméticas, chamada de funções, definição de classes, estruturas if then else, while, for e switch case, não podendo definir classes e funções sem a sua respectiva implementação. Como as linguagens base, também é fortemente tipada, possuindo como tipos primitivos: int, double, float, short, char, bool e void. Ainda, permite a criação de variáveis tanto estáticas quanto dinâmicas, seguindo as regras gramaticais estabelecidas para tal.

Freedom-- oferece uma abordagem fraca de orientação a objetos, permitindo apenas a criação de classes (*struct* em c) contendo métodos e atributos públicos e (ou) privados. Entretanto, por motivos de simplificação *Freedom--* não possui polimorfismo de tipo, logo, não oferece herança.

2. Especificação Formal

a. Gramática:

```
grammar FreedomLessLess;

program_def: import_def? class_def* function_def* main_def? ;
import_def: (IMPORT STRING)+ ;
class_def: CLASS ID OPEN_KEY class_members_def CLOSE_KEY ;
class_members_def: private_def | public_def private_def? ;
public_def: PUBLIC class_scope_def ;
private_def: PRIVATE class_scope_def ;
class_scope_def: (attribute_def SEMICOLON)* function_def* ;
attribute_def:
        type_def ID (ASSIGN valued_expression_def)? (COMMA ID (ASSIGN
        valued_expression_def)?)* |
        type_def ID (OPEN_BRAK INT CLOSE_BRAK)+ (ASSIGN valued_expression_def)?
```

(COMMA ID (OPEN_BRAK INT CLOSE_BRAK)+ (ASSIGN valued_expression_def)?)* |
type_def MULT+ ID (ASSIGN valued_expression_def)? (COMMA MULT+ ID (ASSIGN
valued_expression_def)?)* ;

valued_expression_def:
value_def operation | function_call_def operation |
(MULT | REF) OPEN_PAR valued_expression_def CLOSE_PAR |
ID (((ASSIGN | auto_assign_op) valued_expression_def) | auto_increm_op | OPEN_BRAK
INT CLOSE_BRAK)? operation ;

operation: ((logical_op | arithmetic_op) valued_expression_def)* ;

function_call_def:
DELETE ID | FREE OPEN_PAR ID CLOSE_PAR |
NEW ID OPEN_PAR argument_def? CLOSE_PAR |
MALLOC OPEN_PAR valued_expression_def CLOSE_PAR |
SIZEOF OPEN_PAR type_def (MULT+ | (OPEN_BRAK INT CLOSE_BRAK)+)? CLOSE_PAR |
(ID ('.' | ARROW))? ID OPEN_PAR argument_def? CLOSE_PAR (('.' | ARROW) ID
OPEN_PAR argument_def? CLOSE_PAR)* ;

argument_def: valued_expression_def (COMMA valued_expression_def)* ;

function_def:
type_def (MULT+ | (OPEN_BRAK INT CLOSE_BRAK)+)? ID OPEN_PAR param_def?
CLOSE_PAR block_def ;

param_def:
type_def MULT+ ID (COMMA param_def)* |
type_def ID (OPEN_BRAK INT CLOSE_BRAK)* (COMMA param_def)*;

block_def: OPEN_KEY (valueless_expression_def SEMICOLON | struct_def)* CLOSE_KEY ;

valueless_expression_def:
BREAK | CONTINUE | attribute_def | function_call_def | RETURN valued_expression_def |
(MULT OPEN_PAR ID CLOSE_PAR | ID) ((ASSIGN | auto_assign_op)
valued_expression_def | auto_increm_op) ;

struct_def: if_def | for_def | while_def | switch_def ;

if_def: IF OPEN_PAR valued_expression_def CLOSE_PAR block_def (ELSE block_def)? ;

for_def:
FOR OPEN_PAR valued_attribute_def (COMMA valued_attribute_def)* SEMICOLON
valued_expression_def SEMICOLON valued_expression_def (COMMA
valued_expression_def)* CLOSE_PAR block_def ;

valued_attribute_def:
(type_def | CLASS ID) (MULT* ID | ID (OPEN_BRAK INT CLOSE_BRAK)+) ASSIGN
valued_expression_def ;

while_def: WHILE OPEN_PAR valued_expression_def CLOSE_PAR block_def ;

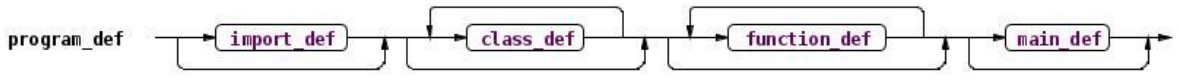
```
switch_def:
        SWITCH OPEN_PAR valued_expression_def CLOSE_PAR OPEN_KEY switch_case_def*
        switch_default_def CLOSE_KEY ;
switch_case_def:
        CASE value_def TWOPOINTS (valueless_expression_def SEMICOLON | struct_def)+
        BREAK SEMICOLON ;
switch_default_def:
        DEFAULT TWOPOINTS (valueless_expression_def SEMICOLON | struct_def)* BREAK
        SEMICOLON ;
main_def:
        VOID_T MAIN OPEN_PAR INT_T ID COMMA CHAR_T MULT MULT ID CLOSE_PAR
        block_def ;
type_def:
        INT_T | UNSIGNED_T | SHORT_T | FLOAT_T | DOUBLE_T |
        CHAR_T | BOOL_T | VOID_T | CLASS ID ;
value_def: INT | CHAR | STRING | INTEGER | FLOATING | BOOLEAN | NULL ;
logical_op: LESS | BIGGER | LESS_EQ | BIGGER_EQ | EQUALS | NOT_EQUALS | AND | OR ;
arithmetic_op: PLUS | MINUS | MULT | DIV ;
auto_assign_op: AUTOPLUS | AUTOMINUS | AUTOMULT | AUTODIV ;
auto_increm_op: INCREM | DECREM ;

//! Primitive types
INT_T           : 'int';
UNSIGNED_T      : 'unsigned';
FLOAT_T         : 'float';
DOUBLE_T        : 'double';
SHORT_T         : 'short';
CHAR_T          : 'char';
BOOL_T          : 'bool';
VOID_T          : 'void';

//! Some reserved words
IMPORT          : 'import';
CLASS           : 'class';
PUBLIC          : 'public' TWOPOINTS;
PRIVATE         : 'private' TWOPOINTS;
MAIN            : 'main';

//! Primitive structs
```

```
IF              : 'if';
ELSE            : 'else';
FOR             : 'for';
WHILE           : 'while';
SWITCH          : 'switch';
CASE            : 'case';
BREAK           : 'break';
CONTINUE        : 'continue';
DEFAULT         : 'default';
RETURN          : 'return';

//! Memory Allocation
NEW             : 'new' ;
FREE            : 'free' ;
MALLOC          : 'malloc' ;
DELETE          : 'delete' ;
SIZEOF          : 'sizeof' ;

//! Operations and operators
ASSIGN          : '=';
PLUS            : '+';
MINUS           : '-';
MULT            : '*';
DIV             : '/';
REF             : '&';
ARROW           : '->';
INCREM          : '++';
DECREM          : '--';
AUTOPLUS        : '+=';
AUTOMINUS       : '-=';
AUTOMULT        : '*=';
AUTODIV         : '/=';
LESS            : '<';
BIGGER          : '>';
LESS_EQ         : '<=';
BIGGER_EQ       : '>=';
EQUALS          : '==';
NOT_EQUALS      : '!=';
AND             : '&&';
```

```
OR              : '||';

//! Control tokens
OPEN_PAR        : '(';
CLOSE_PAR       : ')';
OPEN_KEY        : '{';
CLOSE_KEY       : '}';
OPEN_BRAK       : '[';
CLOSE_BRAK      : ']';
COMMA           : ',';
SEMICOLON       : ';';
TWOPOINTS       : ':';

//! Another types
NULL            : 'null' ;
INT             : NUMBER+ ;
INTEGER         : '-'? INT ;
BOOLEAN         : 'true' | 'false' ;
STRING          : '"' ~('"')* '"' ;
CHAR            : '\'' ~('\'') '\'' ;
FLOATING        : INTEGER ? '.' INT ;
ID              : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

//! Desconsidered text
COMMENT             : ('/*' .*? '*/') -> channel(HIDDEN) ;
WS                  : ( ' ' | '\t' | '\r' | '\n') -> channel(HIDDEN) ;
LINE_COMMENT        : ('//' ~('\n'|'\r')* '\r'? '\n') -> channel(HIDDEN) ;

//! Auxiliary datas
fragment NUMBER     : '0'..'9' ;
fragment ESC        : '\\' ('b' | 't' | 'n' | 'f' | 'r') ;
```

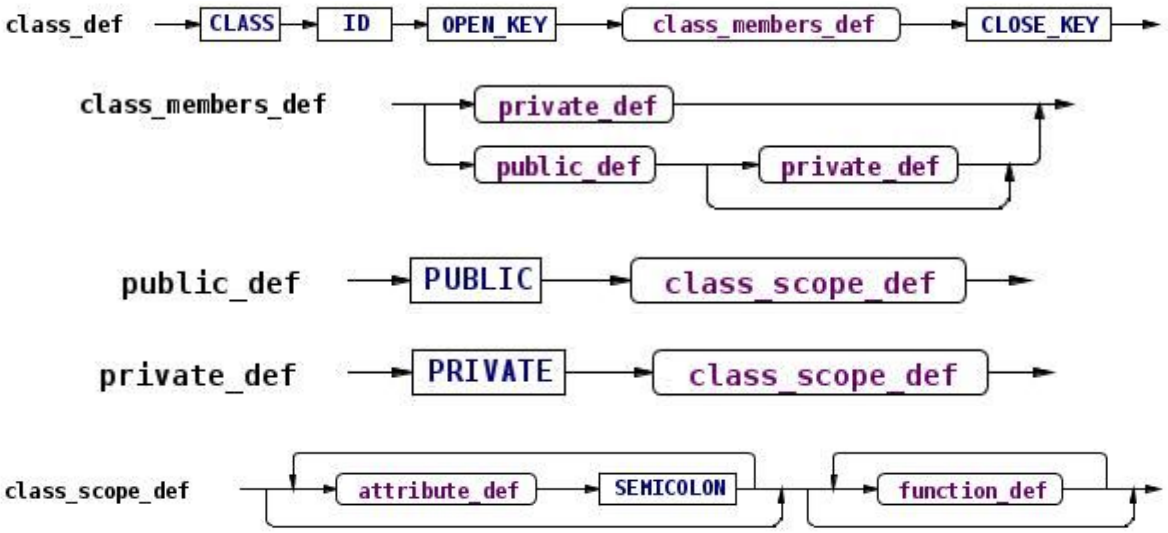
b. Grafos EBNF

i. Estrutura:

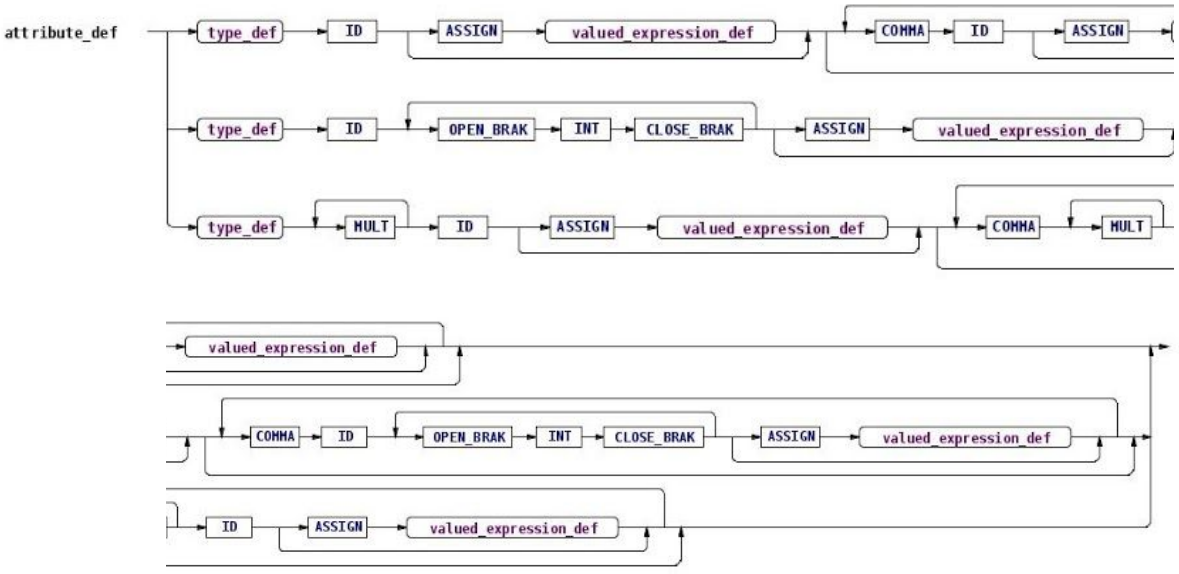


ii. Importação de arquivos externos:

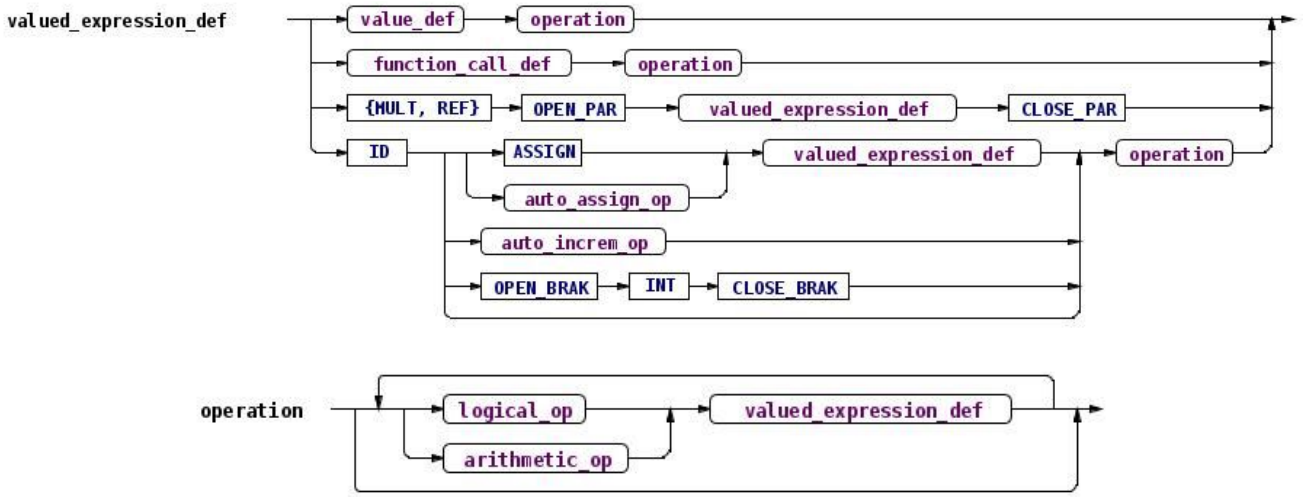
iii. Classes:



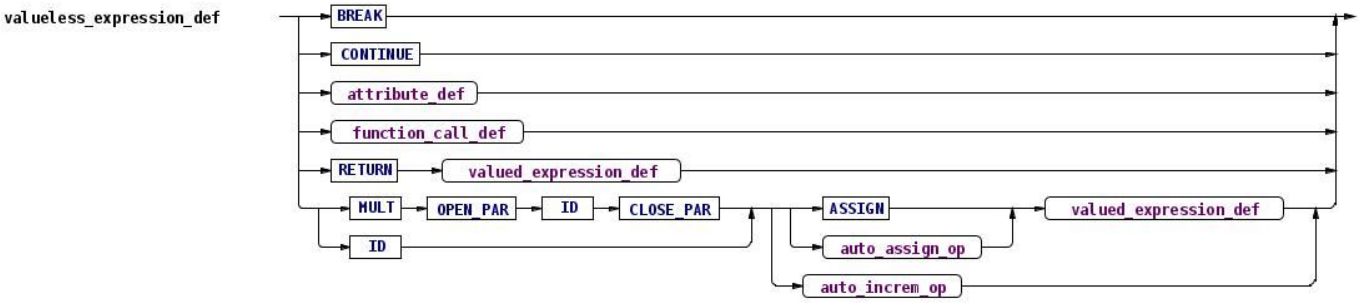
iv. Atributos:



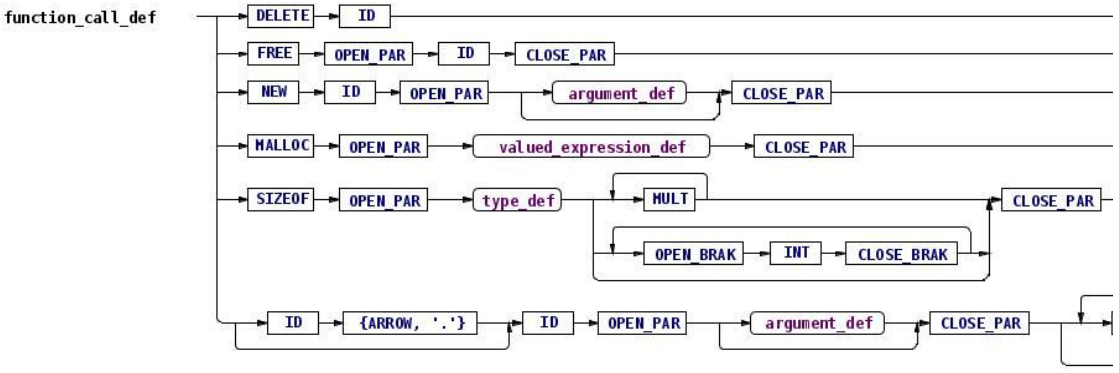
v. Expressão valorada:

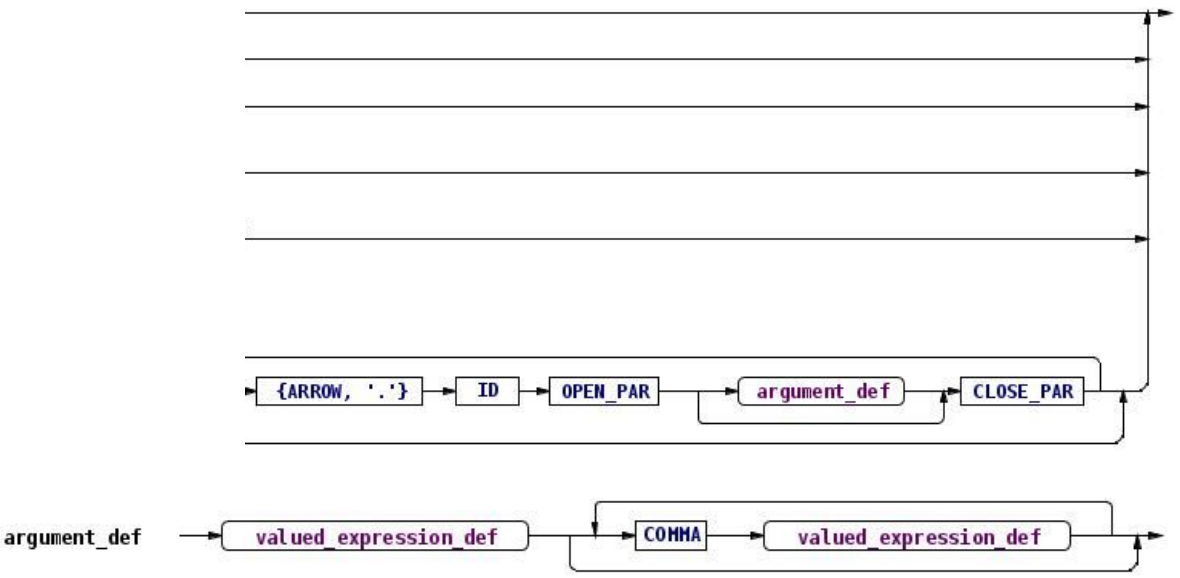


vi. Expressão não valorada:

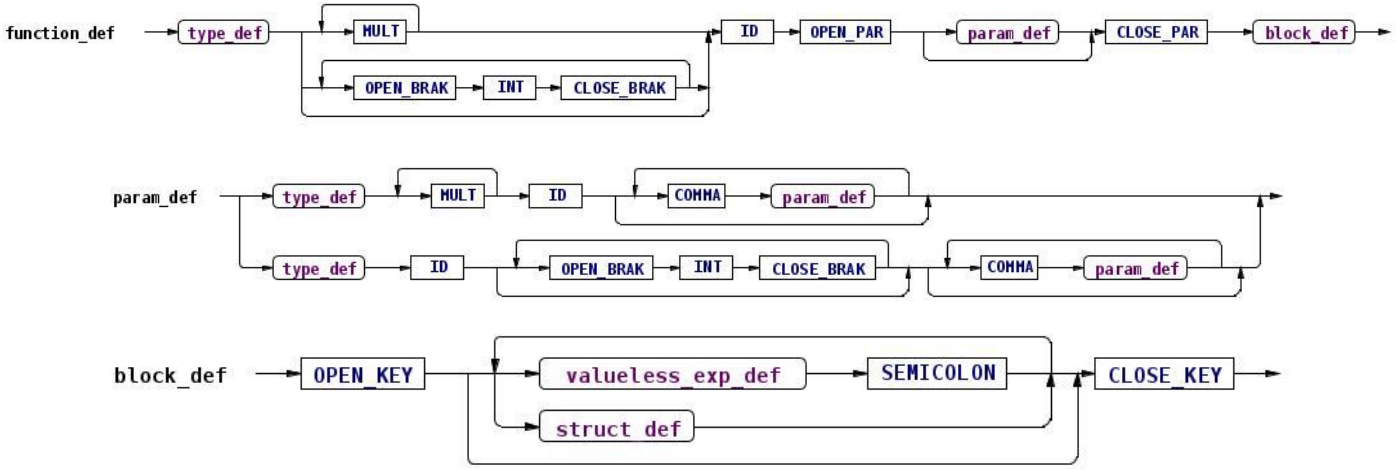


vii. Chamada de função:

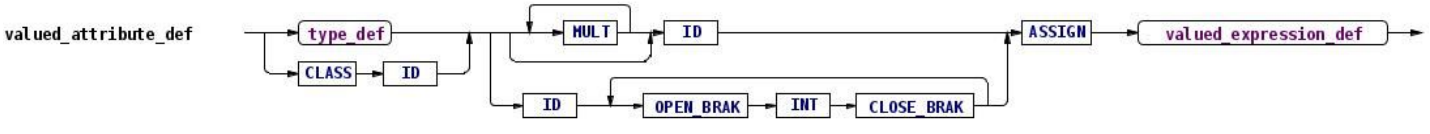




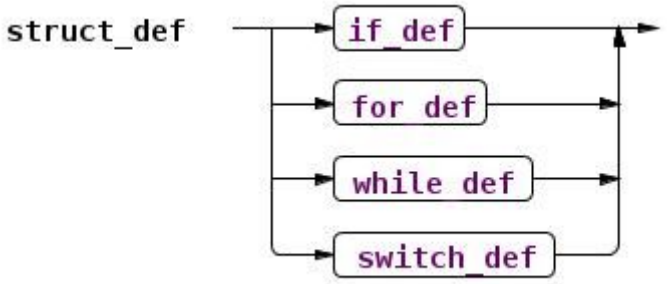
viii. Funções:

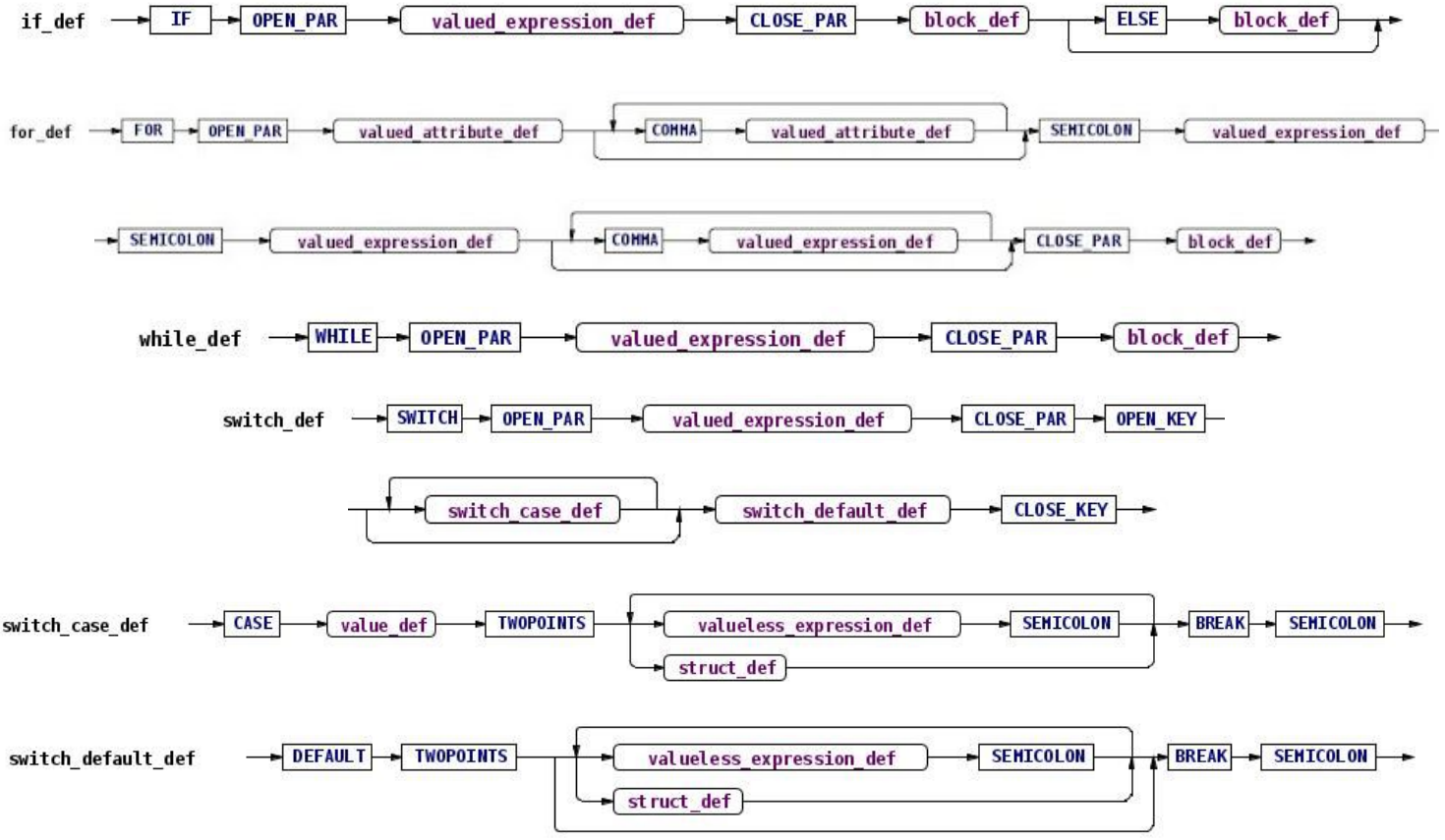


ix. Atributo valorado:

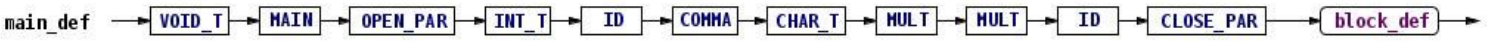


x. Estruturas de seleção e repetição:





xi. Função principal:



3. Analisador Léxico

A implementação do analisador léxico para a linguagem *FreedomLessLess* foi realizada automaticamente através da ferramenta *ANTLR*. Assim, bastou apenas a descrição da gramática utilizando a sintaxe do *ANTLR* e o mesmo gerou automaticamente o analizador léxico para ela.

4. Analisador Sintático

Assim como o analisador léxico, o analisador sintático foi construído automaticamente pela ferramenta *ANTLR.* A ferramenta utilizada gera analisadores sintáticos descendentes recursivos e constrói subrotinas mutuamente recursivas para cada uma das regras de produção definidas na gramática. Assim, a estrutura do programa gerado é um tanto quanto semelhante à especificação formal da gramática.

a. Reconhecimento Através do Analisador Sintático

O processo de reconhecimento de um programa gerado através da linguagem proposta inicia-se através da função (*program_def()*) associada ao símbolo inicial da gramática. Conforme os *tokens* vão sendo reconhecidos são realizadas verificações de acordo com o definido previamente pela gramática. Ainda, enquanto as produções vão sendo identificadas, o analisador sintático vai construindo a árvore sintática do programa.

Cada função cria um nó da árvore de sintaxe do programa. Cada nó é implementado por classes distintas, as quais encapsulam o contexto no qual as funções estão sendo executadas, armazenando nós terminais e (ou) não terminais.

O pseudocódigo a seguir ilustra o comportamento da função *program_def()*:

```
Program_defContext program_def() {

    /*cria o nó que encapsulará o contexto */
    Program_defContext _localctx = ctx();
    try {
        if (exist imports)
            import_def();
        while (exist classes)
            class_def();
        while (exist functions)
            function_def();
        if (exist main)
            main_def();

    } catch (Error &e) {
        /* Tratamento do Erro Sintático */
    }
```

```
        return _localctx;
}
```

b. Emissão e Captura de Erros

A emissão de erros sintáticos ocorre quando o analisador falha ao avaliar uma sequência de *tokens*. Isso significa que não é possível combinar (dar *match*) o *token* sob verificação com o próximo, uma vez que o próximo token não faz parte do que é esperado após o que está sendo avaliado, o que possível utilizando-se as regras de produções da função que está sendo executada.

Os principais erros emitidos durante a análise sintática são:

- *NoViableAltException*: indica que o analisador não pôde decidir qual dos dois ou mais caminhos deve prosseguir para continuar a verificação do restante da entrada. Portanto, é possível que analisador sintático rastreie o *token* inicial da entrada incorreta e também saiba onde estava dentre os vários caminhos onde ocorreu o erro.
- *InputMismatchException*: é aplicado sobre qualquer tipo de exceção onde a entrada (*token*) não é compatível com nada do que é esperado (a sequência não pertence à nenhuma produção da gramática).
- *FailedPredicateException*: utiliza-se este tipo de erro quando um predicado semântico falhou durante sua validação.
- *RecognitionException*: erro base para os demais, assim, engloba erros de previsão, predicados falhos e também entradas incompatíveis.

Todas as funções que implementam uma regra da gramática possuem uma estrutura *try / catch* que capturam as exceções emitidas e avisam o analisador para que seja executado o tratador de erros. O código a seguir exemplifica a captura de uma exceção.

```
try { … }
catch (RecognitionException &e) {
        //! Atualiza estado do tratamento de erros do analisador
        _errHandler->reportError(this, e);

        //! Pega a exceção atual
        _localctx->exception = std::current_exception();
```

```
        //! Executa o tratador de erros
        _errHandler->recover(this, _localctx->exception);
    }
```

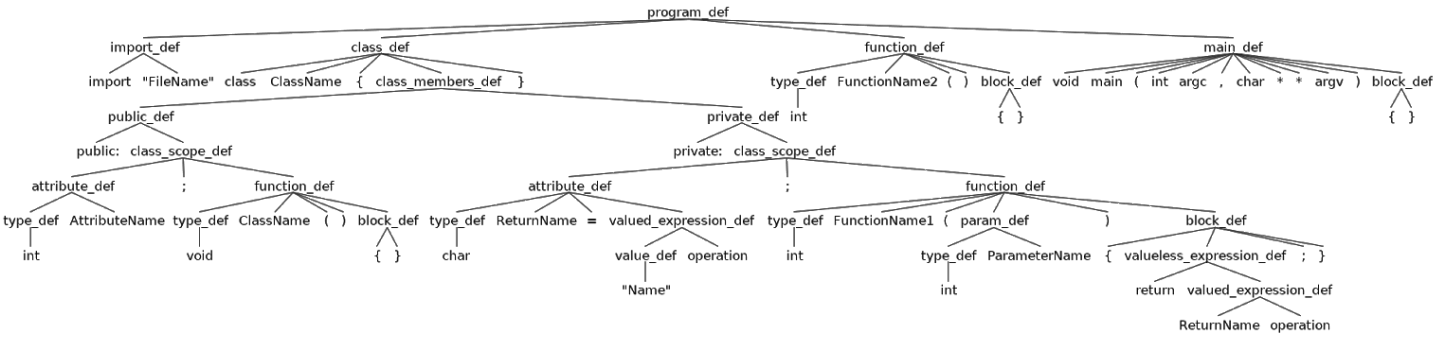
c. Tratamento de erros

Ao identificar o primeiro erro sintático, o tratador de erros entra em modo de recuperação de erros. Nesse modo, ele executa a política de ignorar todos os *tokens* e erros subsequentes. Isso acontece até que seja possível reconhecer um *token* que pertença ao conjunto *Follow* do não-terminal onde o erro foi detectado. Após o reconhecimento do primeiro *token* válido, a análise considera que a regra onde ocorreu o erro foi reconhecida corretamente e retoma o reconhecimento como anteriormente. Caso novos erros ocorram, a mesma estratégia para recuperação de erros é utilizada.

5. Exemplos de Programas Corretos

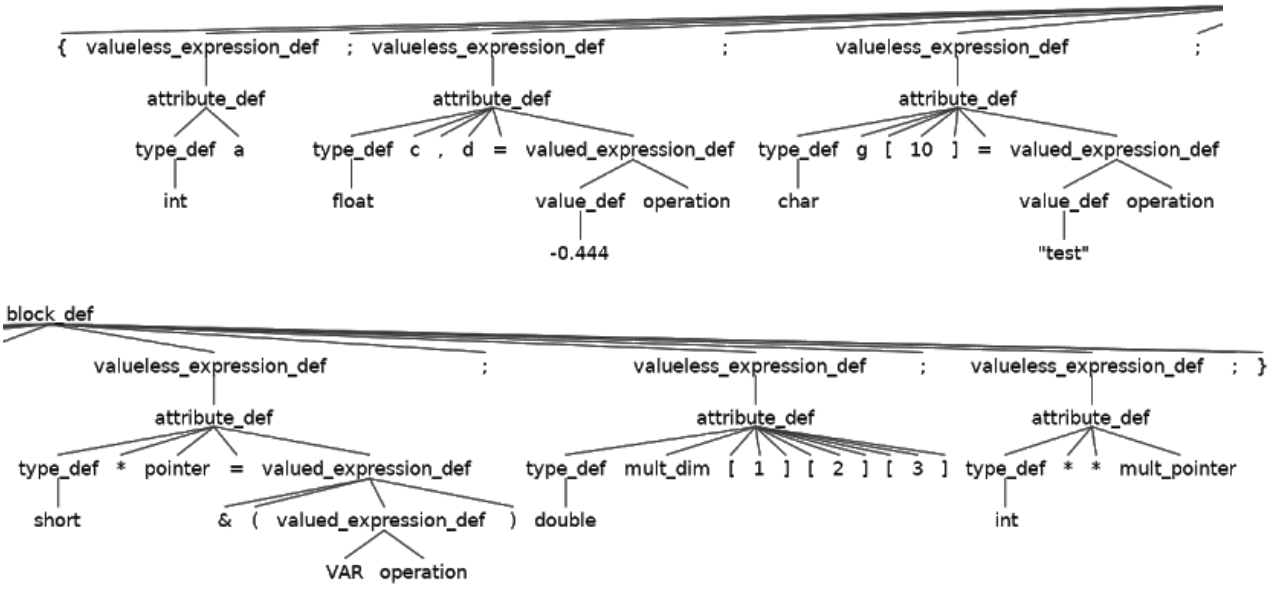
a. Estrutura

```
import "FileName"
class ClassName {
   public:
        int AttributeName;
        void ClassName () {}
   private:
        char ReturnName = "Name";
        int FunctionName1 (int ParameterName) { return ReturnName; }
}
int FunctionName2 () {}
void main (int argc, char **argv) {}
```



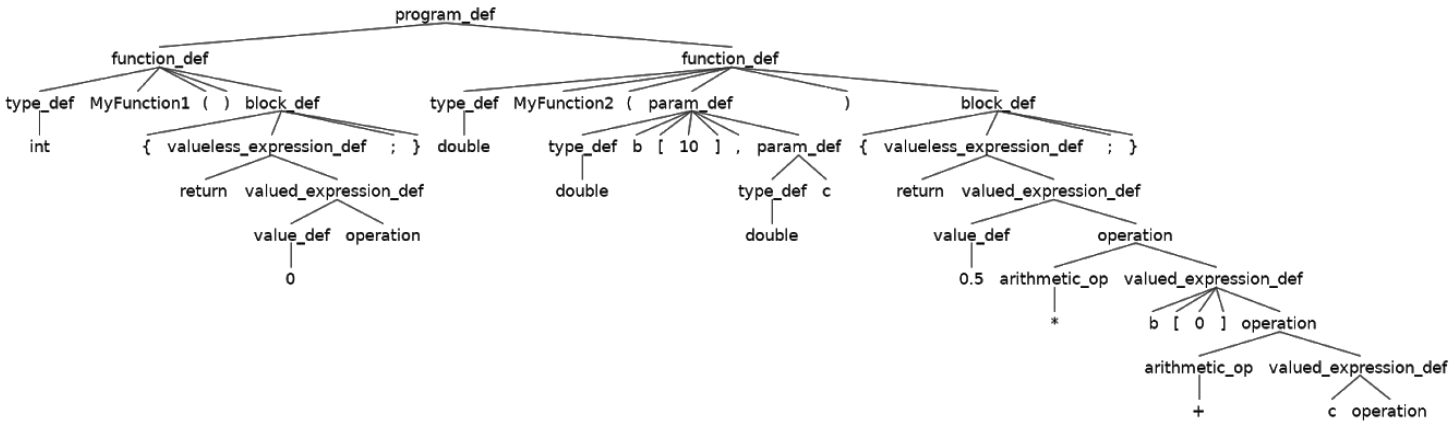
b. Declaração de variáveis e atributos

```
{
    int a;
    float c, d = -0.444;
    char g[10] = "test";
    short * pointer = &(VAR);
    double mult_dim[1][2][3];
    int ** mult_pointer;
}
```



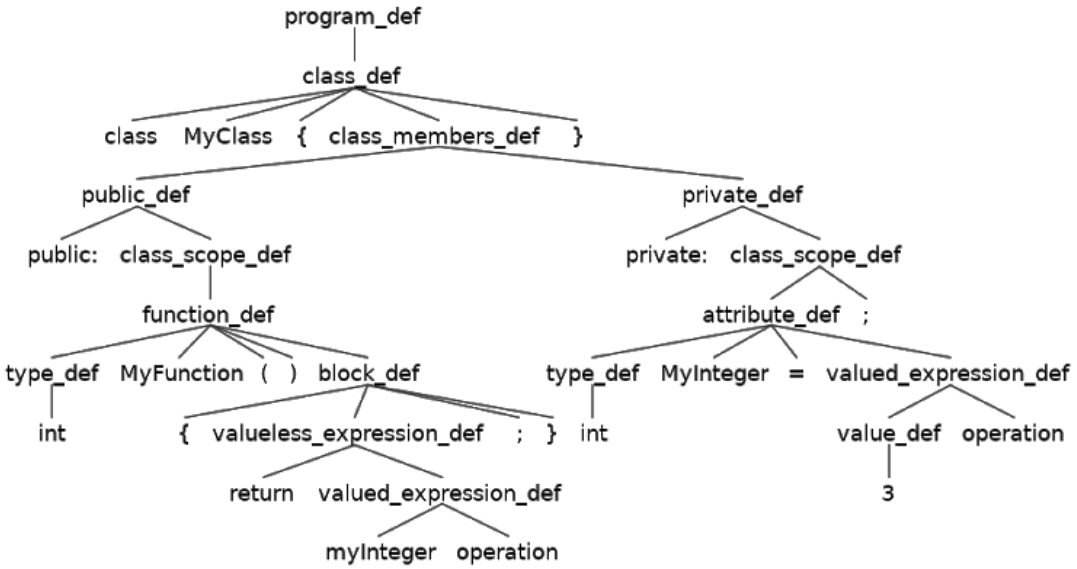
c. Declaração de funções

```
int MyFunction1 () { return 0; }
double MyFunction2 (double b[10], double c) { return 0.5 * b[0] + c; }
```



d. Definição de classes

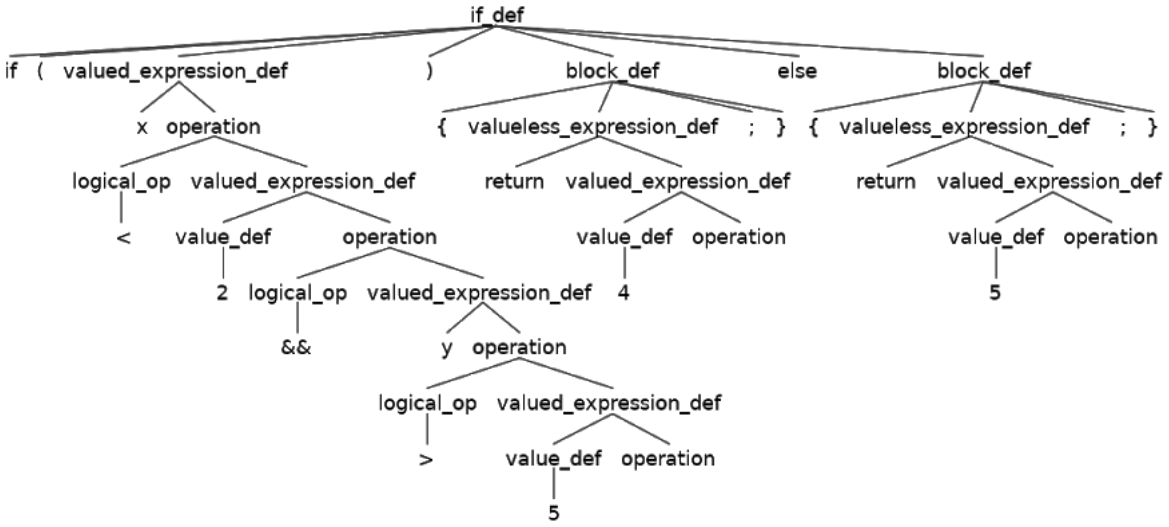
```
class MyClass {
    public:
        int MyFunction () { return myInteger; }
    private:
        int MyInteger = 3;
}
```



e. Estrutura de seleção if

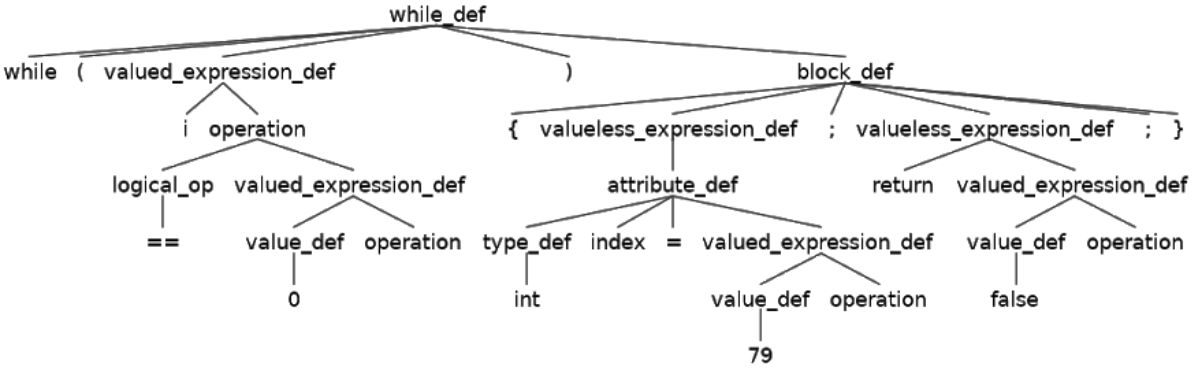
```
if (x < 2 && y > 5) {
    return 4;
```

```
} else {
    return 5;
}
```



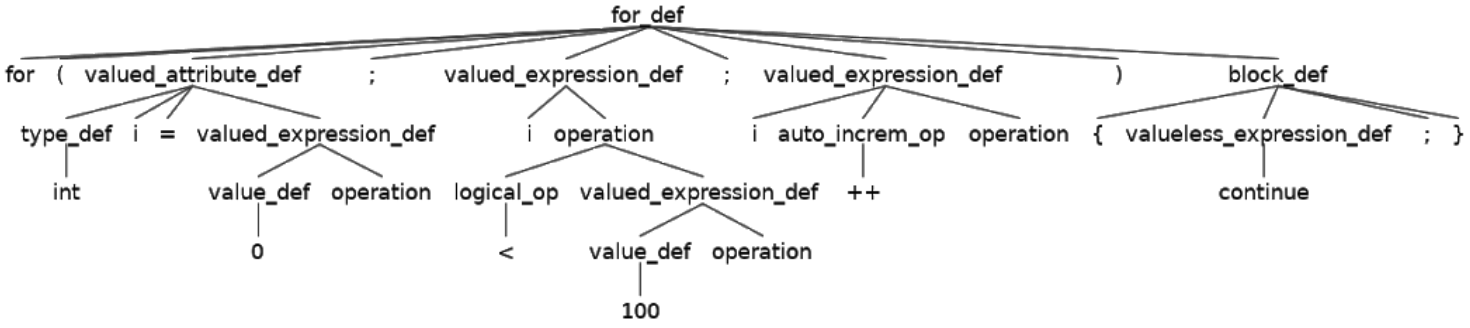
f. Estrutura de seleção while

```
while (i == 0) {
    int index = 79;
    return false;
}
```



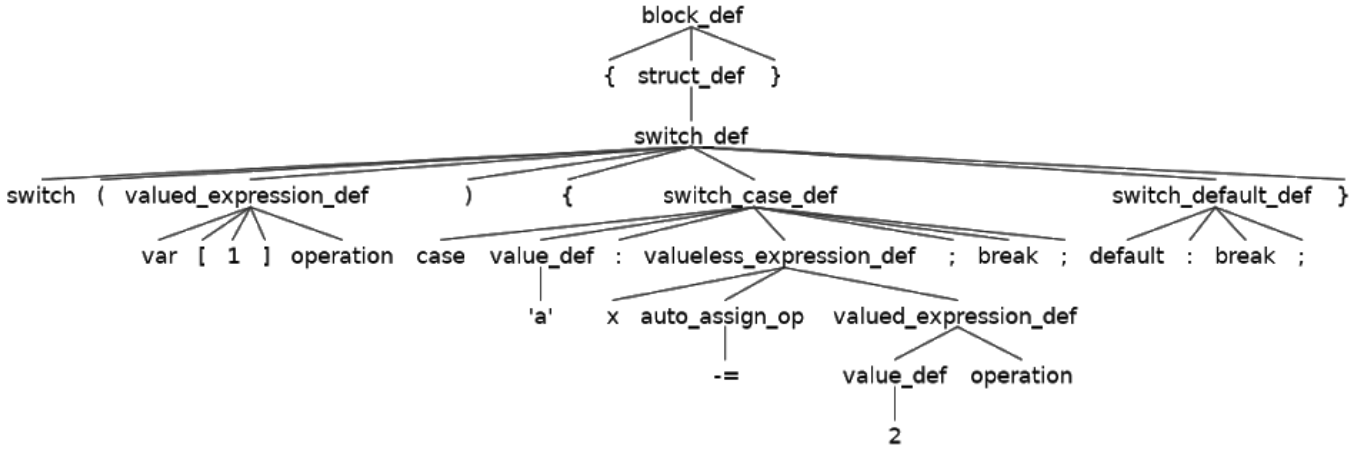
g. Estrutura de seleção for

```
for (int i = 0; i < 100; i++) { continue; }
```



h. Estrutura de seleção switch case

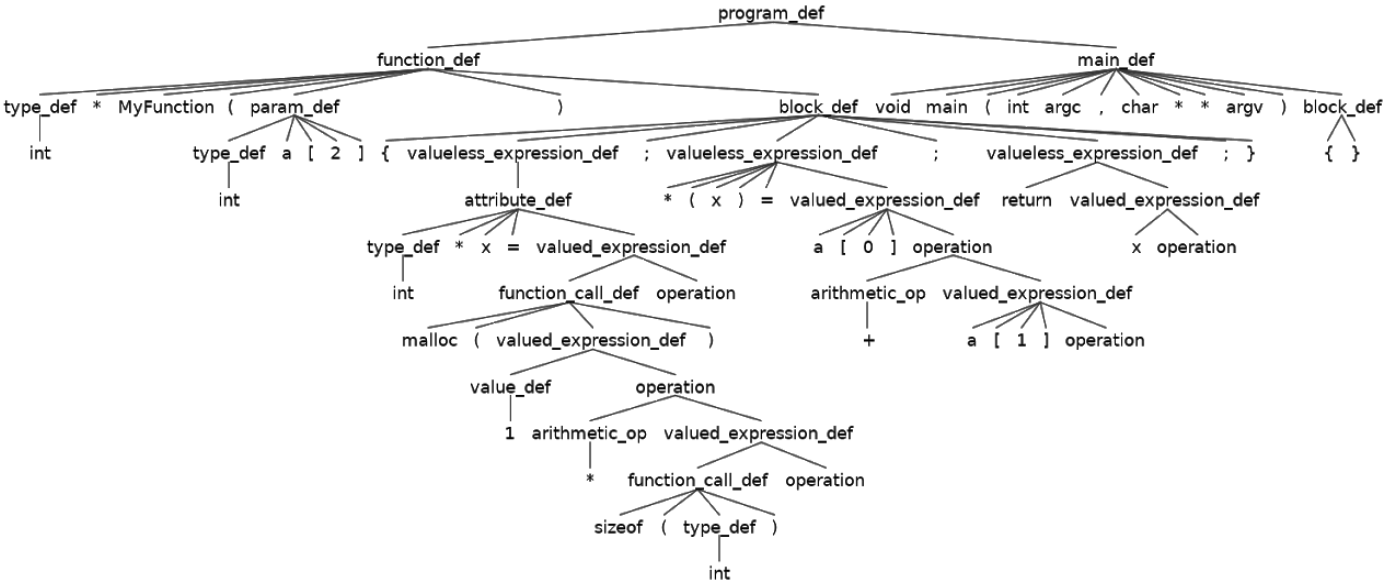
```
{
    switch (var[1]) {
            case 'a': x -= 2;
                    break;
            default:
                    break;
    }
}
```



i. Ponteiros

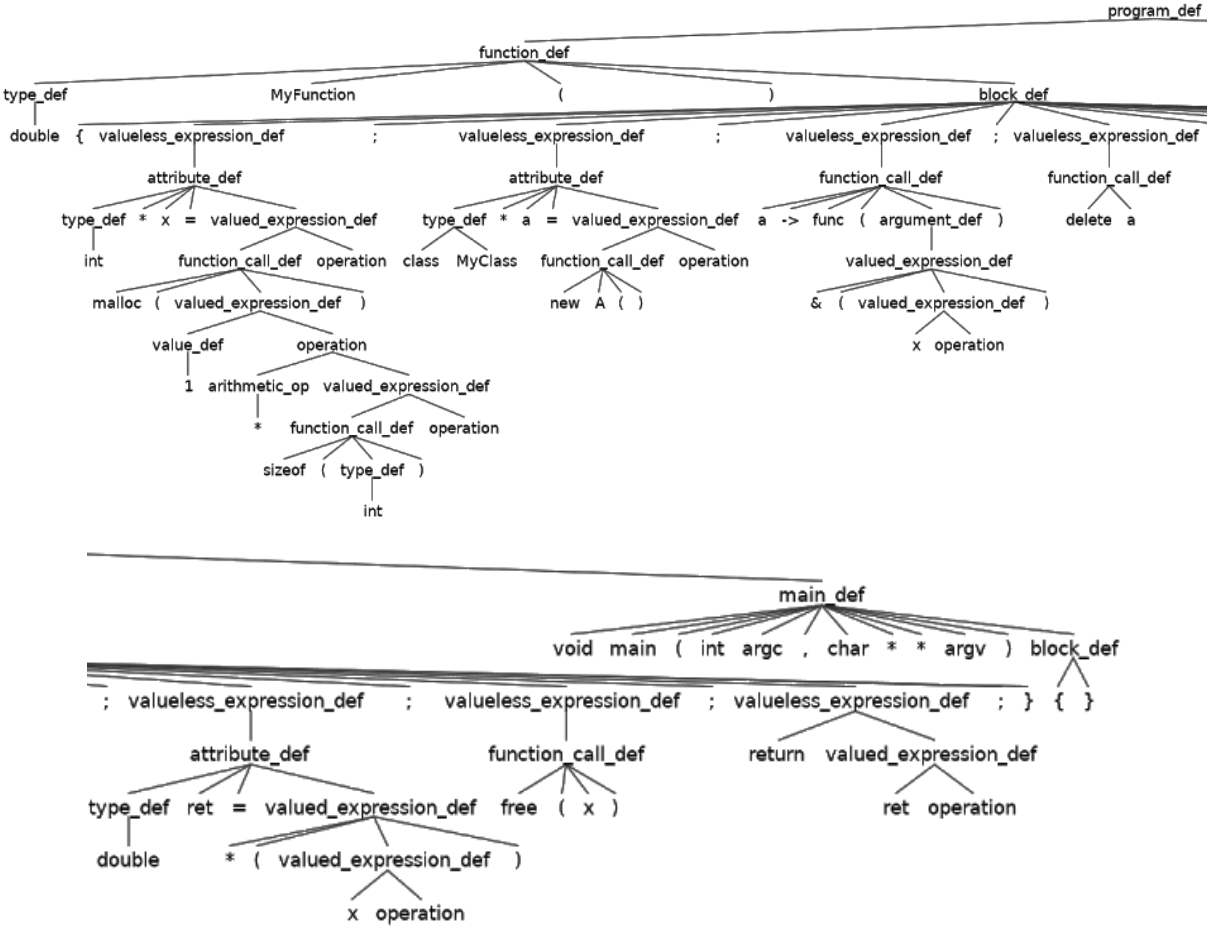
```
int * MyFunction (int a[2]) {
        int * x = malloc(1 * sizeof(int));
        *(x) = a[0] + a[1];
        return x;
}
```

```
void main (int argc, char **argv) {}
```



j. Alocação dinâmica de memória

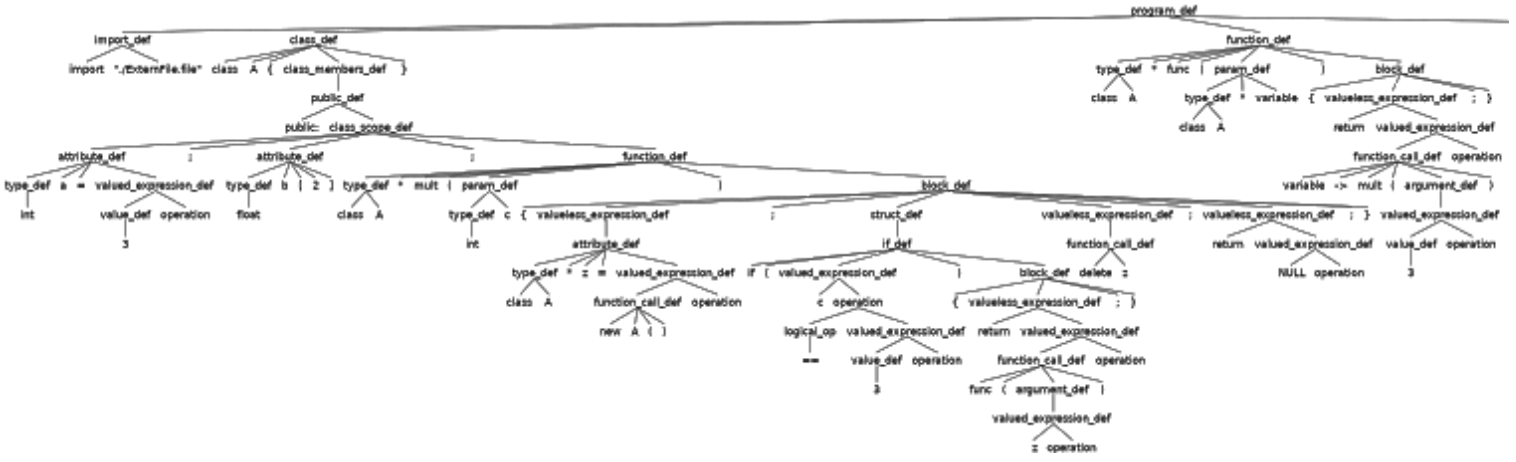
```
double MyFunction () {
        int * x = malloc(1 * sizeof(int));
        class MyClass * a = new A();
        a->func(&(x));
        delete a;
        double ret = *(x);
        free(x);
        return ret;
}
void main (int argc, char **argv) {}
```

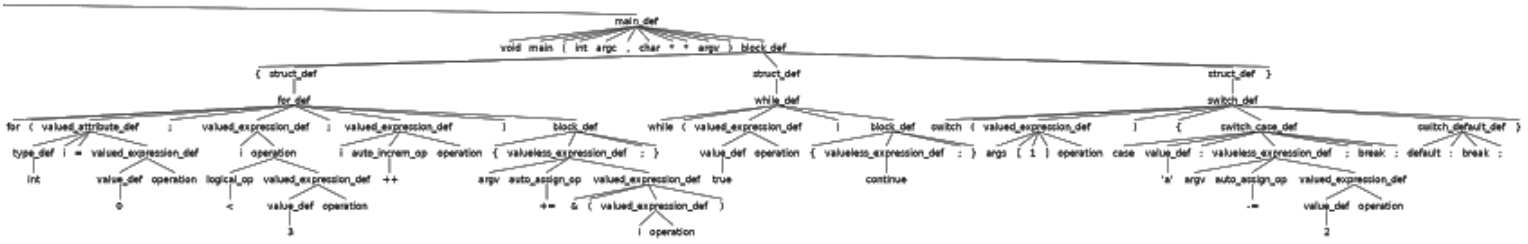


k. Programa completo

```
import "./ExternFile.file"
class A {
   public:
          int a = 3;
          float b[2];
          class A * mult (int c) {
                  class A * z = new A();
                  if (c == 3) {
                          return func(z);
                  }
                  delete z;
                  return NULL;
          }
}
class A * func (class A * variable) { return variable->mult(3); }
```

```
void main(int argc, char **argv) {
    for (int i = 0; i < 3; i++) {
            argv += &(i);
    }
    while (true) {
            continue;
    }
    switch (args[1]) {
            case 'a': argv -= 2;
                    break;
            default:
                    break;
    }
}
```



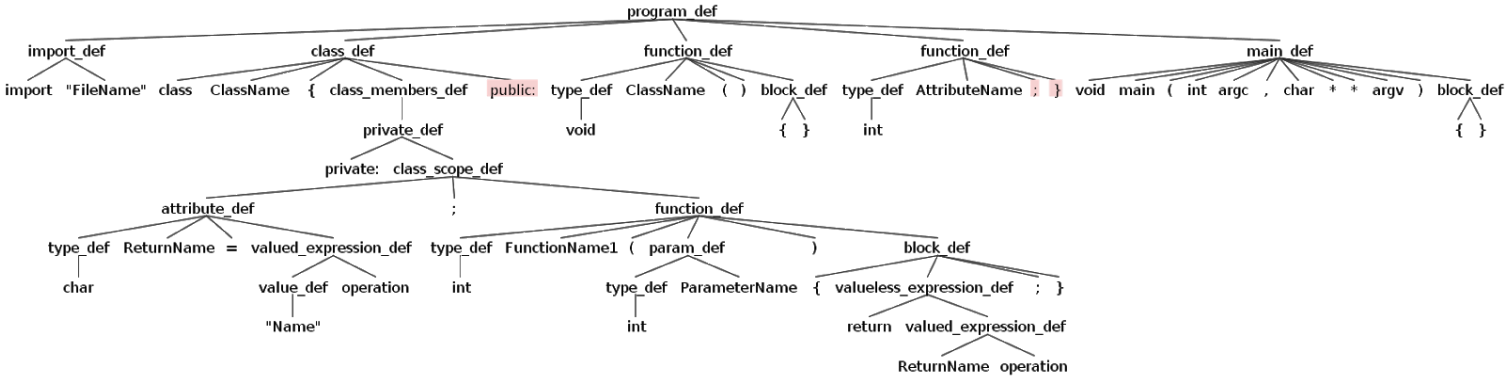


6. Exemplos de Programas com Erros

a. Incorreta organização do código

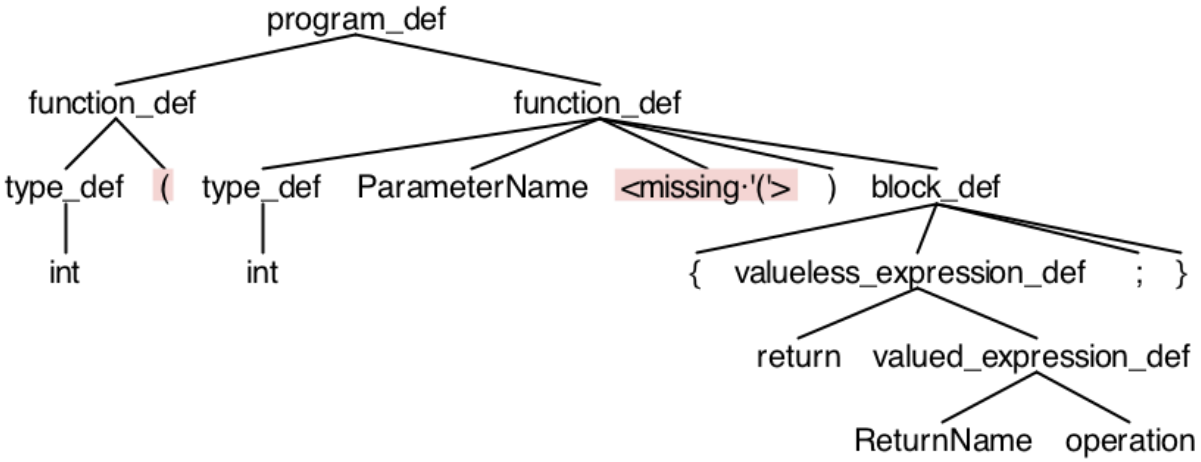
```
import "FileName"
class ClassName {
```

```
    private:
        char ReturnName = "Name";
        int FunctionName1 (int ParameterName) { return ReturnName; }
    public:
        void ClassName () {}
        int AttributeName;
}
void main (int argc, char **argv) {}
int FunctionName2 () {}
```



b. Função sem identificador

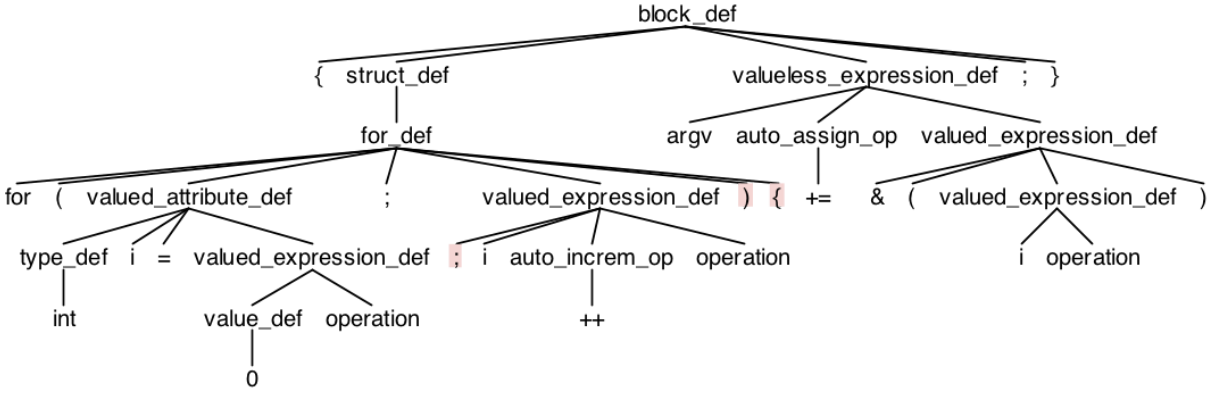
```
int (int ParameterName) { return ReturnName; }
```



c. For sem condição de parada

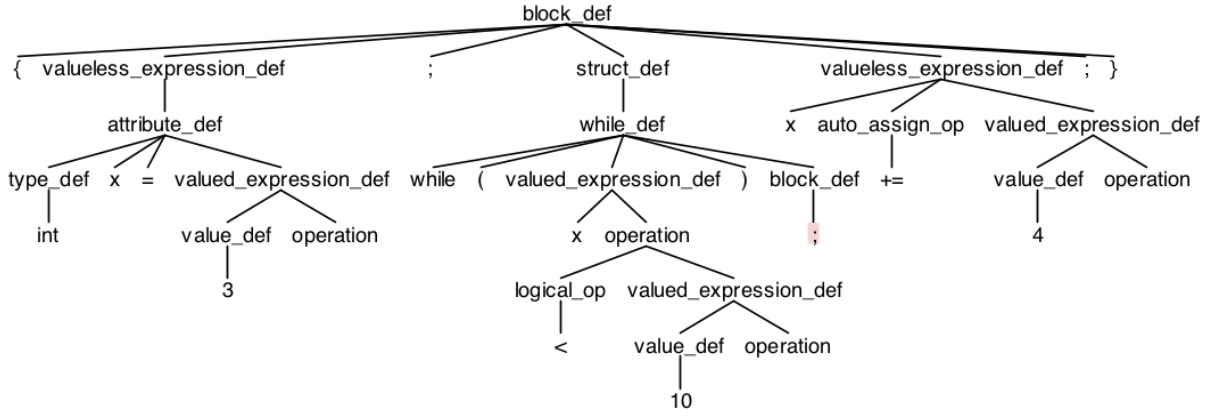
```
}
        for (int i = 0; ; i++) {
                argv += &(i);
        }
```

```
}
```



d. While sem escopo

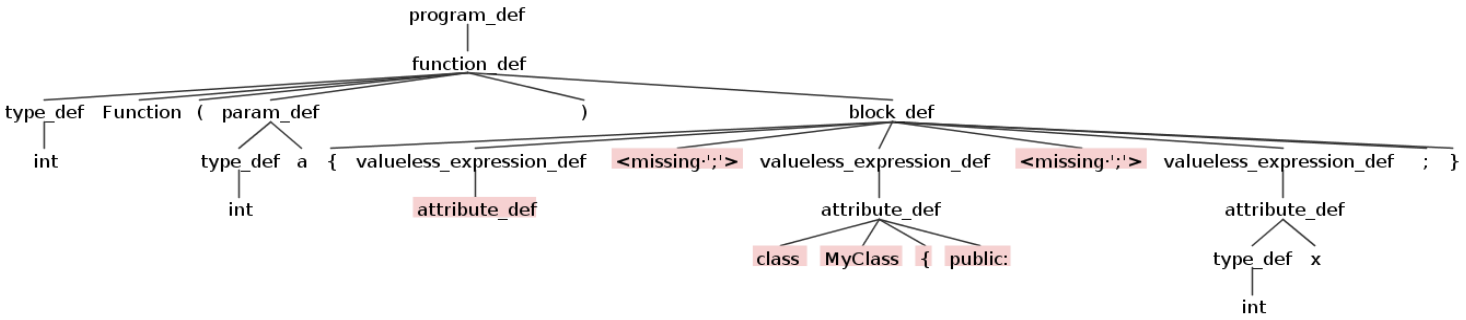
```
{
        int x = 3;
        while (x < 10);
        x += 4;
}
```



e. Definição de classe dentro de uma função

```
int Function(int a) {
        class MyClass {
                public:
                        int x;
        }
        return a * 2;
}
double Function2() {
```

```
        return Pi;
    }
```



7. Referências

1. http://www.antlr.org/
2. https://github.com/antlr/antlr4/tree/master/runtime/Cpp