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

Freedom-- :
Análises Léxica, Sintática e Semântica

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: (attribute_def SEMICOLON)* function_def* class_def* main_def ;
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 operation |
        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 argument_def)* ;
function_def:
        VOID_T ID OPEN_PAR param_def? CLOSE_PAR block_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 (MULT | OPEN_BRAK INT CLOSE_BRAK)? ID OPEN_PAR param_def? CLOSE_PAR block_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 (MULT ID | ID OPEN_BRAK INT CLOSE_BRAK) ASSIGN valued_expression_def ;
while_def: WHILE OPEN_PAR valued_expression_def CLOSE_PAR block_def ;
```

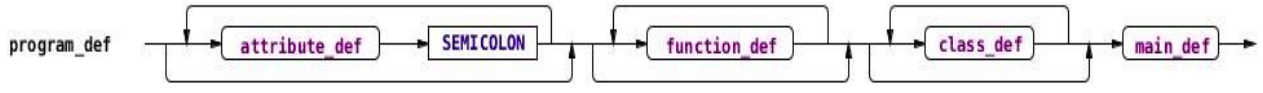
```
switch_def:
        SWITCH OPEN_PAR valued_expression_def CLOSE_PAR OPEN_KEY switch_case_def*
        switch_default_def CLOSE_KEY ;
switch_case_def:
        CASE value_def TWOPOINTS (valueless_expression_def SEMICOLON | struct_def)+
        BREAK SEMICOLON ;
switch_default_def:
        DEFAULT TWOPOINTS (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 | DOUBLE_T | CHAR_T | BOOL_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
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        : '(';
```

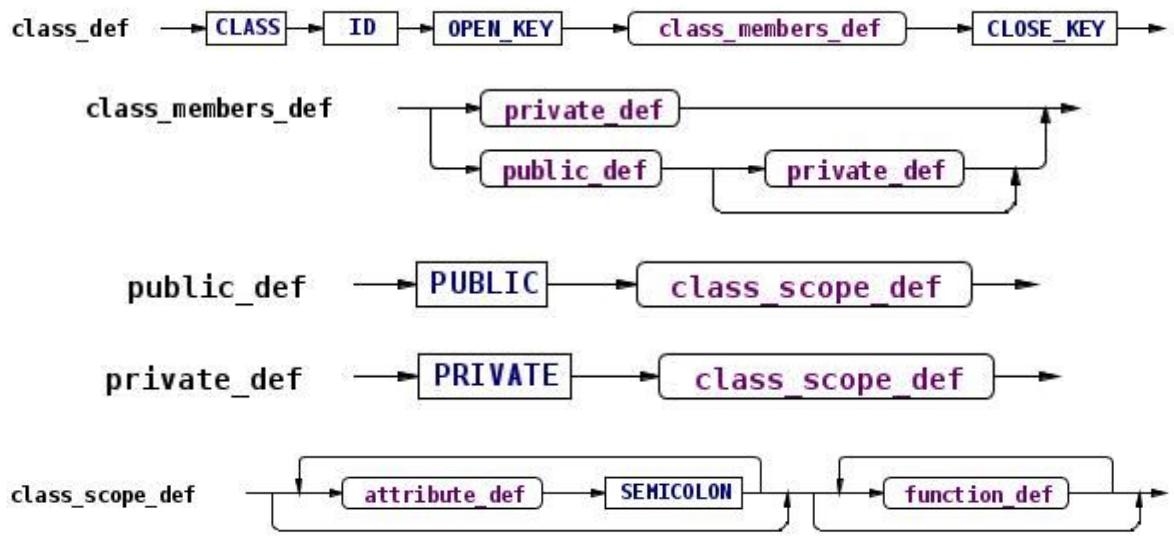
```
CLOSE_PAR    : ')';
OPEN_KEY     : '{';
CLOSE_KEY    : '}';
OPEN_BRAK    : '[';
CLOSE_BRAK   : ']';
COMMA        : ',';
SEMICOLON    : ';';
TWOPOINTS    : ':';

//! 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

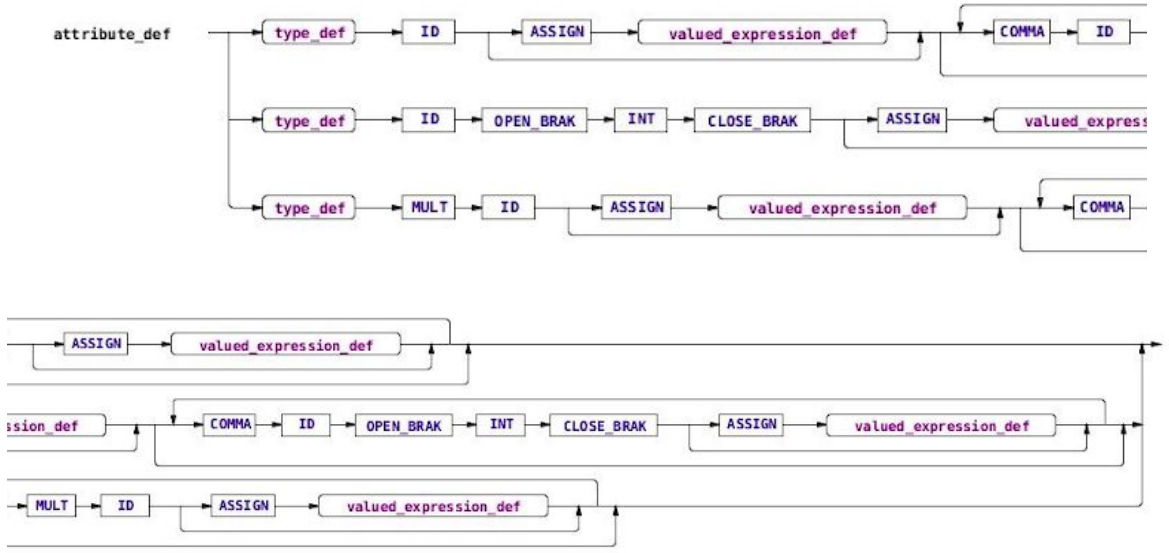
i. Estrutura:



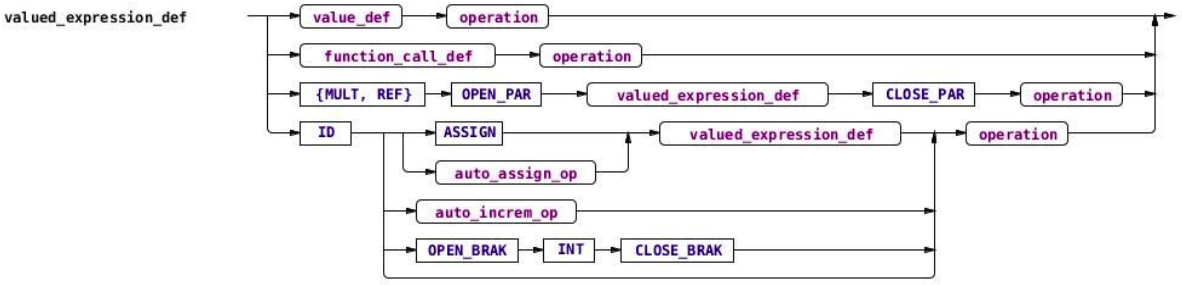
ii. Classes:

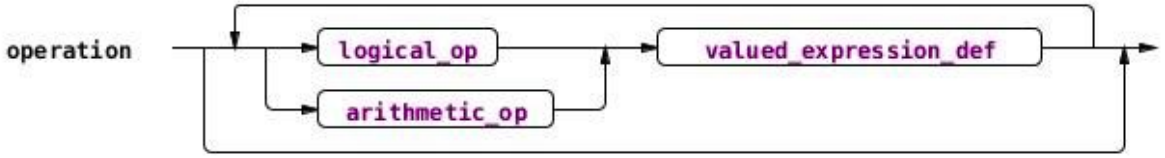


iii. Atributos:

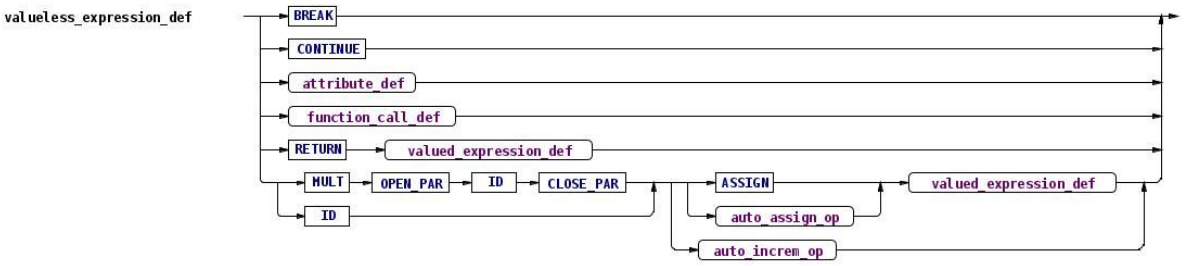


iv. Expressão valorada:

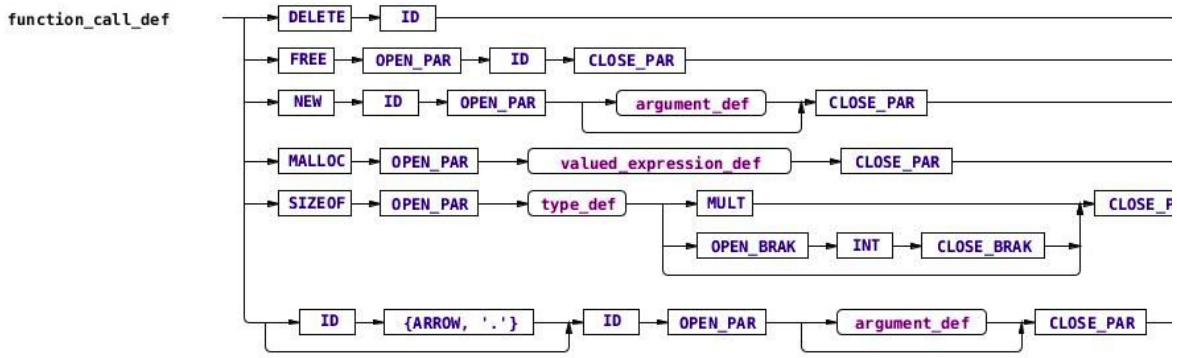


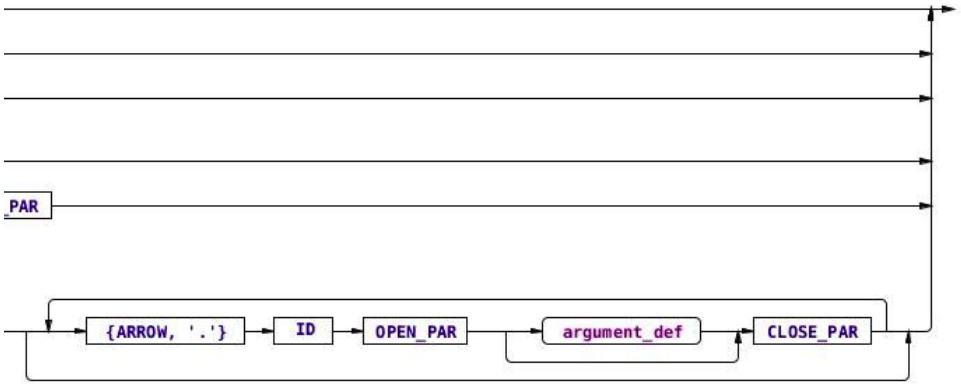


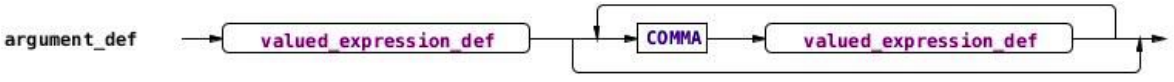
v. Expressão não valorada:



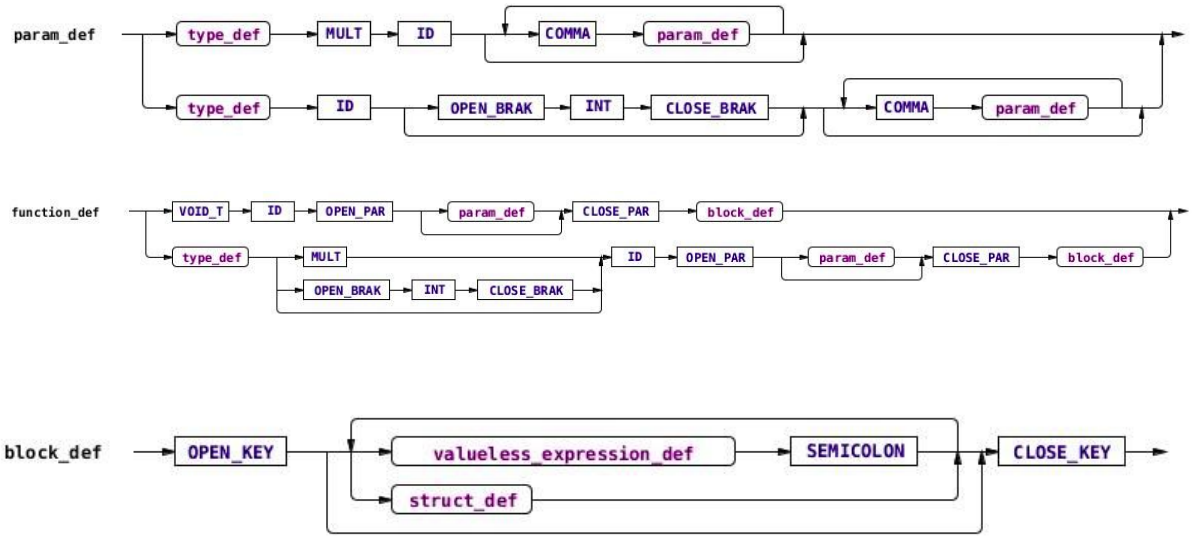
vi. Chamada de função:



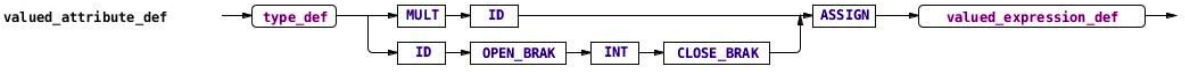




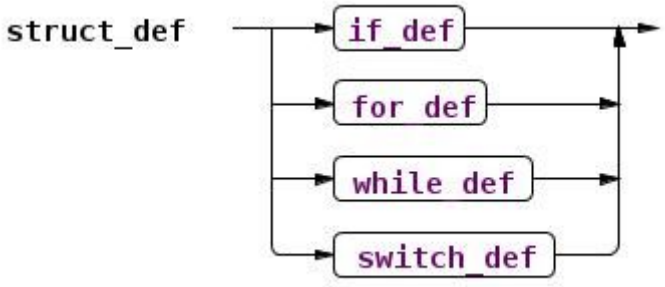
vii. Funções:

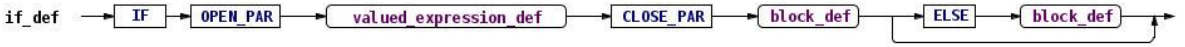


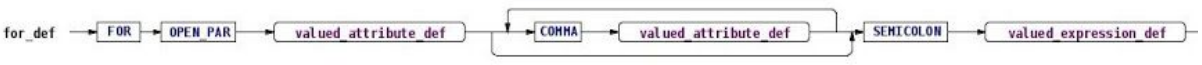
viii. Atributo valorado:

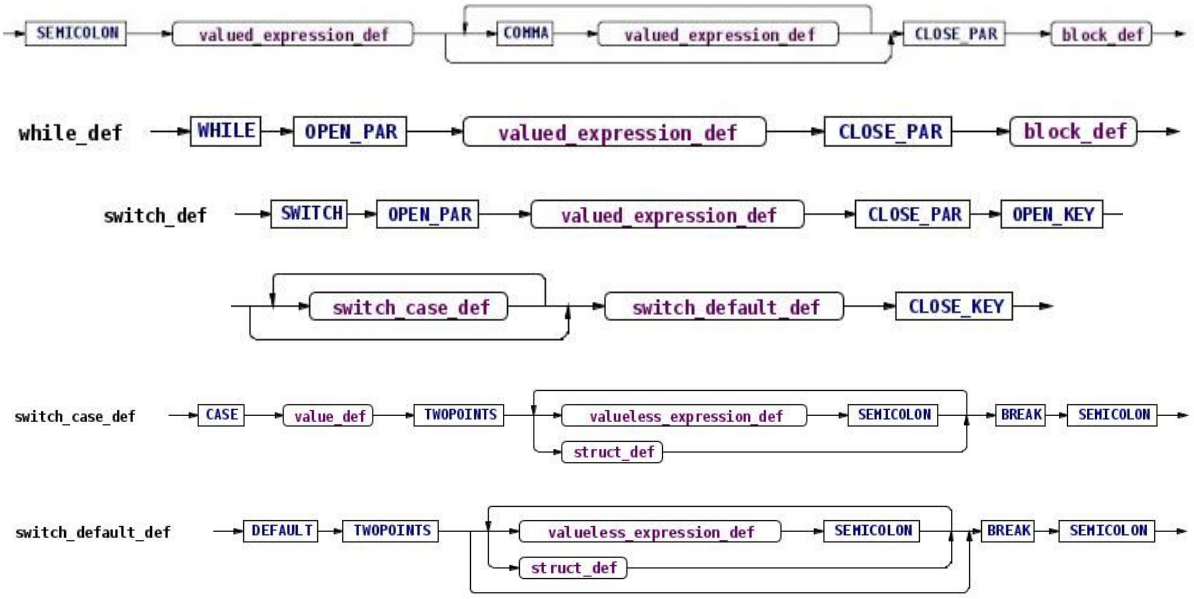


ix. Estruturas de seleção e repetição:

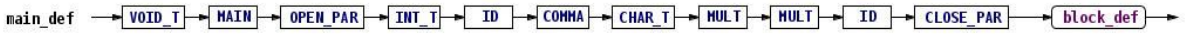








x. Função principal:



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 attributes)
                        attribute_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. Analisador Semântico (Conceitos e Definições)

O processo de análise semântica não é realizada automaticamente pela ferramenta Antlr. No entanto, ela facilita bastante sua implementação, uma vez que permite inserir alguns comandos diretamente na gramática. Tais comandos geram funções no analisador sintático, as quais ficam a cargo do usuário implementar. Ainda, essas funções podem ser chamadas passando-se como parâmetros os tokens que compõem a produção. Assim, é possível ter acesso à certos objetos durante a análise sintática para que seja possível realizar a análise semântica e, como a função é executada no contexto da produção em que foi inserida, é possível ter acesso à esse contexto.

a. Regras Semânticas

Neste trabalho, para cada uma das produções da gramática, foram definidas as regras semânticas necessárias para que o programa sendo avaliado possa de fato realizar o que propõe a linguagem. Abaixo é mostrada uma tabela com as produções do lado esquerdo e as correspondentes regras semânticas. Vale lembrar que todas elas, em geral, implicitamente possuem a regra onde os símbolos não terminais do lado direito da produção recebem o escopo do não terminal do lado esquerdo (exemplo: para a produção A -> B tem-se a regra B.scope = A.scope).

i. Descrição das funções criadas:

- scope_contains(...): Verifica se um determinado ID já foi declarado.

- is_dynamic_alloc(...): Verifica se a variável aponta para uma memória alocada dinamicamente.
- verify_return(...): Verifica em um conjunto de instruções se existe uma instrução de retorno.

ii. Tabela de regras e predicados das produções:

Produção	Regras e Predicados
program_def -> (attribute_def SEMICOLON)* function_def* class_def* main_def	
class_def -> CLASS ID OPEN_KEY class_members_def CLOSE_KEY	**Predicado:** !scope_contains(class_def.scope, ID.name)?
class_members_def -> private_def	
class_members_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)?)*	**Regra:** attribute_def.type = type_def.type **Predicado:** !scope_contains(attribute_def.scope, ID.name)? attribute_def.type =? valued_expression_def.type
attribute_def -> type_def ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)? (COMMA ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)?)*	**Regra:** attribute_def.type = type_def.type + OPEN_BRAK INT CLOSE_BRAK

	Predicado: !scope_contains(attribute_def.scope, ID.name)?
attribute_def -> type_def MULT ID (ASSIGN valued_expression_def)? (COMMA MULT ID (ASSIGN valued_expression_def)?)*	**Regra:** attribute_def.type = type_def.type + MULT **Predicados:** !scope_contains(attribute_def.scope, ID.name)? attribute_def.type =? valued_expression_def.type
valued_expression_def -> value_def operation	**Regras:** valued_expression_def.value = value_def.value operation.op operation.value valued_expression_def.type = valued_expression_def.value.type
valued_expression_def -> function_call_def operation	**Regras:** valued_expression_def.value = function_call_def.return_value operation.op operation.value valued_expression_def.type = valued_expression_def.value.type **Predicados:** function_call_def.type !=? ‘void’ function_call_def.type !=? ‘class *’ function_call_def.type !=? ‘address’ function_call_def.type !=? ‘valueless’
valued_expression_def -> (MULT | REF) OPEN_PAR valued_expression_def CLOSE_PAR operation	**Regras:** valued_expression_def.value = ((MULT | REF) valued_expression_def).value operation.op operation.value valued_expression_def.type =

	valued_expression_def.value.type
valued_expression_def -> ID (((ASSIGN | auto_assign_op) valued_expression_def) | auto_increm_op | OPEN_BRAK INT CLOSE_BRAK)? operation	**Regras:** valued_expression_def.value = (ID (((ASSIGN | auto_assign_op) valued_expression_def) | auto_increm_op | OPEN_BRAK INT CLOSE_BRAK)?).value operation.op operation.value valued_expression_def.type = valued_expression_def.value.type **Predicado:** scope_contains(valued_expression_def.scope, ID.name)?
operation -> ((logical_op | arithmetic_op) valued_expression_def)*	**Regras:** operation.op = (logical_op.op | arithmetic_op.op) operation.type = valued_expression_def.type operation.value = valued_expression_def.value
function_call_def -> DELETE ID	**Regra:** function_call_def.return_type = ‘valueless’ **Predicados:** scope_contains(function_call_def.scope, ID.name)? ID.type =? ‘class’
function_call_def -> FREE OPEN_PAR ID CLOSE_PAR	**Regra:** function_call_def.return_type = ‘valueless’ **Predicados:** scope_contains(function_call_def.scope, ID.name)? is_dynamic_alloc(ID)?

function_call_def -> NEW ID OPEN_PAR argument_def? CLOSE_PAR	**Regra:** function_call_def.return_type = ‘class *’ **Predicados:** scope_contains(function_call_def.scope, ID.name)? ID.type =? ‘class’ ID.param_type =? argument_def.type
function_call_def -> MALLOC OPEN_PAR valued_expression_def CLOSE_PAR	**Regra:** function_call_def.return_type = ‘address’ **Predicado:** valued_expression_def.type =? 'int'
function_call_def -> SIZEOF OPEN_PAR type_def (MULT+ | (OPEN_BRAK INT CLOSE_BRAK)+)? CLOSE_PAR	**Regra:** function_call_def.return_type = ‘int’
function_call_def -> (ID ('.' | ARROW))? ID OPEN_PAR argument_def? CLOSE_PAR (('.' | ARROW) ID OPEN_PAR argument_def? CLOSE_PAR)*	**Regras:** function_call_def.return_type = ID[1].return_type function_call_def.return_value = ID[1].return_value (function_call_def.return_type = ID[2].return_type)* **Predicados:** (scope_contains(function_call_def.scope, ID[0].name)? && ID[0].type =? ‘class’)? scope_contains(function_call_def.scope, ID[1].name)? && ID[1].type =? ‘function’ && ID[1].param_type =? argument_def[0].type (scope_contains(function_call_def.scope, ID[2].name)? && ID[2].type =? ‘function’ && ID[2].param_type =? argument_def[1].type
argument_def -> valued_expression_def (COMMA argument_def)*	**Regra:** argument_def[0].type = valued_expression_def.type (+

	argument_def[1])*
function_def -> type_def MULT ID (COMMA param_def)*	**Regras:** function_def.param_type = param_def.type function_def.return_type = type_def + MULT ID.type = ‘function’ **Predicados:** !scope_contains(function_def.scope, ID.name)?
function_def -> type_def ID (OPEN_BRAK INT CLOSE_BRAK)? (COMMA param_def)*	**Regras:** function_def.param_type = param_def.type function_def.return_type = type_def + (OPEN_BRAK INT CLOSE_BRAK)? ID.type = ‘function’ **Predicados:** !scope_contains(function_def.scope, ID.name)?
param_def -> type_def MULT ID (COMMA param_def)*	**Regras:** ID.type = type_def.type + MULT param_def[0].type = ID.type + (param_def[1].type)* **Predicado:** !scope_contains(param_def[0].scope, ID.name)?
param_def -> type_def ID (OPEN_BRAK INT CLOSE_BRAK)? (COMMA param_def)*	**Regras:** ID.type = type_def.type + (OPEN_BRAK INT CLOSE_BRAK)? param_def[0].type = ID.type + (param_def[1].type)* **Predicado:** !scope_contains(param_def.scope,

	ID.name)?
block_def -> OPEN_KEY (valueless_expression_def SEMICOLON | struct_def)* CLOSE_KEY	**Regra:** block_def.type = verify_return(valueless_expression_def, struct_def)
valueless_expression_def -> BREAK	**Regra:** valueless_expression_def.type = ‘break’
valueless_expression_def -> CONTINUE	**Regra:** valueless_expression_def.type = ‘continue’
valueless_expression_def -> attribute_def	**Regras:** valueless_expression_def.type = attribute_def.type valueless_expression_def.value = attribute_def.value
valueless_expression_def -> function_call_def	**Regras:** valueless_expression_def.type = function_call_def.return_type valueless_expression_def.value = function_call_def.return_value
valueless_expression_def -> RETURN valued_expression_def	**Regras:** valueless_expression_def.type = ‘return’ + valued_expression_def.type valueless_expression_def.value = valued_expression_def.value
valueless_expression_def -> (MULT OPEN_PAR ID CLOSE_PAR | ID) ((ASSIGN | auto_assign_op) valued_expression_def | auto_increm_op)	**Regra:** valueless_expression_def.type = ‘valueless’ **Predicados:** scope_contains(valueless_expression_def. scope, ID.name) ID.type =? valued_expression_def.type

struct_def -> if_def	**Regra:** struct_def.type = if_def.type
struct_def -> for_def	**Regra:** struct_def.type = for_def.type
struct_def -> while_def	**Regra:** struct_def.type = while_def.type
struct_def -> switch_def	**Regra:** struct_def.type = switch_def.type
if_def -> IF OPEN_PAR valued_expression_def CLOSE_PAR block_def (ELSE block_def)?	**Regra:** if_def.type = verify_return(block_def[0], block_def[1])
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	**Regra:** for_def.type = verify_return(block_def)
valued_attribute_def -> type_def (MULT ID | ID OPEN_BRAK INT CLOSE_BRAK) ASSIGN valued_expression_def	**Regra:** valued_attribute_def.type = type_def.type (MULT | OPEN_BRAK INT CLOSE_BRAK) ID.type = type_def.type (MULT | OPEN_BRAK INT CLOSE_BRAK) ID.value = valued_expression_def.value **Predicados:** !scope_contains(valued_attribute_def.scope, ID.name)? valued_expression_def.type =? type_def.type + (MULT | OPEN_BRAK INT CLOSE_BRAK)
while_def -> WHILE OPEN_PAR valued_expression_def CLOSE_PAR block_def	**Regra:**

	while_def.type = verify_return(block_def)
switch_def -> SWITCH OPEN_PAR valued_expression_def CLOSE_PAR OPEN_KEY switch_case_def* switch_default_def CLOSE_KEY	**Regras:** switch_def.conditional_type = valued_expression_def.type switch_default_def.type = verify_return(switch_case_def, switch_default_def) **Predicado:** switch_case_def.conditional_type =? switch_def.conditional_type
switch_case_def -> CASE value_def TWOPOINTS (valueless_expression_def SEMICOLON | struct_def)+ BREAK SEMICOLON	**Regras:** switch_case_def.conditional_type = value_def.type switch_case_def.type = verify_return(valueless_expression_def, struct_def)
switch_default_def -> DEFAULT TWOPOINTS (valueless_expression_def SEMICOLON | struct_def)* BREAK SEMICOLON	**Regra:** switch_default_def.type = verify_return(valueless_expression_def, struct_def)
main_def -> VOID_T MAIN OPEN_PAR INT_T ID COMMA CHAR_T MULT MULT ID CLOSE_PAR block_def	
type_def -> INT_T	**Regra:** type_def.type = ‘int’
type_def -> DOUBLE_T	**Regra:** type_def.type = ‘double’
type_def -> CHAR_T	**Regra:** type_def.type = ‘char’
type_def -> BOOL_T	**Regra:**

	type_def.type = ‘bool’
type_def -> CLASS ID	**Regras:** type_def.type = ‘class’ type_val = ID[0].value
value_def -> INT	**Regras:** value_def.value = INT.value value_def.type = ‘int’
value_def -> CHAR	**Regras:** value_def.value = CHAR.value value_def.type = ‘char’
value_def -> STRING	**Regras:** value_def.value = STRING.value value_def.type = ‘string’
value_def -> INTEGER	**Regras:** value_def.value = INTEGER.value value_def.type = ‘integer’
value_def -> FLOATING	**Regras:** value_def.value = FLOATING.value value_def.type = ‘floating’
value_def -> BOOLEAN	**Regras:** value_def.value = BOOLEAN.value value_def.type = ‘bool’
value_def -> NULL	**Regras:** value_def.type = ‘null’
logical_op -> BIGGER	**Regra:**

	logical_op.op = ‘>’
logical_op -> LESS_EQ	**Regra:** logical_op.op = ‘<=’
logical_op -> BIGGER_EQ	**Regra:** logical_op.op = ‘>=’
logical_op -> EQUALS	**Regra:** logical_op.op = ‘==’
logical_op -> NOT_EQUALS	**Regra:** logical_op.op = ‘!=’
logical_op -> AND	**Regra:** logical_op.op = ‘&&’
logical_op -> OR	**Regra:** logical_op.op = ‘||’
logical_op -> LESS	**Regra:** logical_op.op = ‘<’
arithmetic_op -> MINUS	**Regra:** arithmetic_op.op = ‘-’
arithmetic_op -> MULT	**Regra:** arithmetic_op.op = ‘*’
arithmetic_op -> DIV	**Regra:** arithmetic_op.op = ‘/’
arithmetic_op -> PLUS	**Regra:** arithmetic_op.op = ‘+’
auto_assign_op -> AUTOMINUS	**Regra:**

	auto_assign_op.op = '-='
auto_assign_op -> AUTOMULT	**Regra:** auto_assign_op.op = '*='
auto_assign_op -> AUTODIV	**Regra:** auto_assign_op.op = '/='
auto_assign_op -> AUTOPLUS	**Regra:** auto_assign_op.op = '+='
auto_increm_op -> INCREM	**Regra:** auto_increm_op.op = '++'
auto_increm_op -> DECREM	**Regra:** auto_increm_op.op = '--'

b. Exemplos de aplicação das regras semânticas

i. Comando de atribuição correto:

Exemplo: int x[10];

- **Produção 1:**
  - attribute_def -> type_def ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)? (COMMA ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)?)*
  - **Regra:** attribute_def.type = type_def.type + OPEN_BRAK INT CLOSE_BRAK
    - attribute_def.type = int + [10] **Correto**
  - **Predicado:** !scope_contains(attribute_def.scope, ID.name)?
    - !scope_contains(attribute_def.scope, 'x')? **Correto**
- **Produção 2:**
  - type_def -> INT
  - **Regra:** type_def.type = 'int'

        - type_def.type = int **Correto**

ii. Alocação de memória com número negativo:

Ao alocar dinamicamente um espaço de memória passando como parâmetro um valor menor ou igual a 0, ocorre um erro semântico devido a falha do predicado *"valued_expression_def.type =? 'int'".*

Exemplo: int * x = malloc(-1);

- **Produção 1:** function_call_def -> MALLOC OPEN_PAR valued_expression_def CLOSE_PAR
    - **Regra:** function_call_def.return_type = 'address'
        - function_call_def.return_type = address
    - **Predicado:** valued_expression_def.type =? 'int'
        - valued_expression_def.type =? integer **Errado**
- **Produção 2:** valued_expression_def -> value_def operation
    - **Regra 1:** valued_expression_def.value = value_def.value operation.op operation.value
        - valued_expression_def.value = -1 **Correto**
    - **Regra 2:** valued_expression_def.type = valued_expression_def.value.type
        - valued_expression_def.type = integer **Correto**
- **Produção 3:** value_def -> INTEGER
    - **Regra 1:** value_def.value = INTEGER.value
        - value_def.value = -1 **Correto**
    - **Regra 2:** value_def.type = 'integer'
        - value_def.type = integer **Correto**

6. Analisador Semântico (Implementação)

Dado que em ambas as disciplinas de linguagens formais e compiladores o aprofundamento sobre o uso da tabela de símbolos para análise semântica é bem superficial, neste trabalho foi decidido realizar a implementação da mesma manualmente. Sendo assim, não houve nenhum estudo sobre se existe e, se existe, como funciona a tabela de símbolos do Antlr.

a. Tabela de símbolos

Para a construção da tabela de símbolos foi criado uma classe chamada, denominada *SymbolEntry*, a qual representa uma instância da tabela. Essa classe possui o seguinte conjunto de atributos:

- Tipo: define se a entrada é pertence à um atributo, classe função etc.
- Id: identifica o nome da entrada, ou seja, o nome da função, classe etc.
- Escopo da classe: identifica à qual classe essa entrada pertence;
- Escopo de função: identifica à qual função essa entrada pertence;
- Permissão: informa se a entrada é pública ou privada;
- Características: serve com um campo adicional para características a mais de uma entrada de acordo com o tipo. Se a entrada é de um atributo, por exemplo, então ela tem como característica o tipo do atributo. Já uma função teria como característica o tipo do retorno e os tipos de parâmetros;
- Validade: verdadeiro quando a entrada é de uma declaração de um atributo, classe ou função e falso quando é apenas o uso delas.

i. Código

```
...
class SymbolEntry {
        public String type;
        public String id;
        public String c_scope;
        public String f_scope;
        public String permission;
        public ArrayList<String> features;
        public boolean valid;

        public SymbolEntry() {
                this.type = "null";
```

```
                this.id = "null";
                this.c_scope = "null";
                this.f_scope = "null";
                this.permission = "null";
                this.features = new ArrayList<String>();
                this.valid = false;
        }
};
```

A tabela de símbolos em si é bem simples, consistindo apenas um vetor de objetos do tipo *SymbolEntry*, assim como mostra o código abaixo.

```
...
public static ArrayList<SymbolEntry> _symbolTable = new ArrayList<SymbolEntry>();
...
```

b. Definição de Escopo

Neste trabalho a construção do escopo é de extrema importância para derivação de diversas outras análises semânticas. Isso se deve ao fato de que o escopo é construído concomitantemente com a construção da árvore sintática do programa. Abaixo é ilustrado qual a prioridade de escopo o analizador semântico leva em consideração para suas análises.

Legenda: O escopo global (alta prioridade) engloba o escopo de classe (média prioridade) e ambos englobam o escopo de função (baixa prioridade).

i. Definição do escopo de um nó da árvore sintática

Para definir o escopo de um nó da árvore, foi criado uma interface no parser do compilador. Esta interface serve unicamente para definir métodos que todos os nós da árvore de sintaxe deve implementar para definir corretamente seu escopo, e é mostrada no código abaixo:

```
interface ScopeInformation {
        public String type();
        public String c_scope();
        public String f_scope();
        public String permission();
        public String name();
}
```

Todo não terminal da gramática possui uma classe que define seu contexto na árvore de sintaxe e é responsável por de fato realizar a análise sintática da produção quando este símbolo está do lado esquerdo. Todas as classes de contexto foram estendidas à interface de escopo para que a mesma possa definir quais as características iniciais do escopo da produção correspondente. Abaixo, é mostrado um exemplo da classe de contexto que encapsula o comportamento do não terminal *Class_def*:

```
…
public static class Class_defContext extends ParserRuleContext implements
ScopeInformation {
                …
                @Override
                public String type()            { return "class"; }
                @Override
                public String c_scope()         { return _c_scope; }
                @Override
                public String f_scope()         { return _f_scope; }
                @Override
                public String permission()      { return _permission; }
                @Override
                public String name()            { return _name; }
```

```
        public String _permission;
        public String _c_scope;
        public String _f_scope;
        public String _name = "null";
}

public final Class_defContext class_def() throws RecognitionException,
Exception {
        Class_defContext _localctx = new Class_defContext(_ctx, getState());
        enterRule(_localctx, 2, RULE_class_def);
        try { enterOuterAlt(_localctx, 1); {
                setState(82);
                match(CLASS);
                setState(83);
                match(ID);

                _localctx._name = _localctx.ID().getSymbol().getText();

                SymbolEntry entry = new SymbolEntry();

                entry.c_scope = _localctx.c_scope();
                entry.f_scope = _localctx.f_scope();
                entry.features.add("null");
                entry.permission = _localctx.permission();
                entry.id = _localctx.ID().getSymbol().getText();
                entry.type = _localctx.type();
                entry.valid = true;

                lookUpTable(entry);

                setState(84);
                match(OPEN_KEY);
                setState(85);
                class_members_def();
```

```
                setState(86);
                match(CLOSE_KEY);
            }} catch (RecognitionException re) {
                _localctx.exception = re;
                _errHandler.reportError(this, re);
                _errHandler.recover(this, re);
            } finally { exitRule(); }
            return _localctx;
        }
        ....
```

c. Regras Semânticas

As análise semânticas consiste, basicamente, em definir e verificar detalhadamente as características dos identificadores do programa fonte através da tabela de símbolos, ou seja, avaliar se determinada instância ou uso de um identificador segue as regras semânticas especificadas a seguir:

i. Todo atributo global deve ser primeiro declarado e depois utilizado;
ii. Todo atributo, função e classe devem estar declaradas no programa;
iii. Um mesmo identificador não pode ser utilizado como tipos diferentes;
iv. Toda classe deve ter identificador único dentre as classes, funções e atributos;
v. Nenhum atributo pode ter o mesmo nome de um atributo, função ou classe de escopo igual ou superior;
vi. A declaração e uso de uma função deve ter o mesmo número de argumentos.

d. Look Up Table e Final Check

Todas as verificações semânticas de escopo foram implementadas dentro da função lookUpTable(), a qual recebe as entradas das tabelas para os identificadores encontrados durante o processo de análise sintática. Abaixo segue parte do código,

visto que o mesmo possui inúmeros tratamentos e, portanto, é bastante extenso:

```
…
public void lookUpTable(SymbolEntry entry) throws Exception {
        SymbolEntry temp;
        for (int i = 0; i < _symbolTable.size(); i++) {
                temp = _symbolTable.get(i);
                if (!temp.id.equals(entry.id))
                        continue;

                if (temp.type.equals("class")) {
                        if (entry.type.equals("class"))
                                throw new Exception(...);

                        if (entry.type.equals("variable"))
                                throw new Exception(...);

                        if (entry.type.equals("function")) {
                                if (entry.c_scope.equals(temp.id)) {
                                        for (SymbolEntry e : _symbolTable)
                                                if (e.type.equals("function") && e.id.equals(temp.id))
                                                        throw new Exception(...);
                                        continue;
                                }
                                throw new Exception(...);
                        }
                }
                …
        }
        {
                if (!entry.valid && entry.type.equals("variable") &&
                entry.c_scope.equals("null") && entry.f_scope.equals("null"))
                        throw new Exception(...);

                if (!entry.valid && entry.type.equals("variable") &&
```

```
                entry.c_scope.equals("null") && !entry.f_scope.equals("null"))
                    throw new Exception(...);
            }
            _symbolTable.add(entry);
    }

    private void finalCheck() throws Exception {
            String msg = "";
            boolean error = false;
            for (SymbolEntry entry : _symbolTable)
                if (!entry.valid) {
                    msg += " - " + entry.id + " tipo: " + entry.type + "\n";
                    error = true;
                }
            if (error)
                throw new Exception(...);
    }
```

e. Exemplos de Análise Semântica

- **Variável global não definida**

```
int FunctionName () {
    return undefinedGlobalVariable;
}

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

Erro: java.lang.Exception: Variável undefinedGlobalVariable está sendo usada antes de ser declarada!

- **Uso global de uma variável**

```
int undefinedGlobalVariable = firstVariable + 1;

int FunctionName () {
    return undefinedGlobalVariable;
}

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

Erro: java.lang.Exception: Variável firstVariable está sendo usada antes de ser declarada!

- **Função global não definida**

```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

int FunctionName () {
        return globalVariable + undefinedFunction(firstVariable);
}

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

Erro: java.lang.Exception: As seguintes variáveis não foram definidas (ou definidas no lugar errado): undefinedFunction tipo: function!

- **Conflitos entre nomes de variáveis/funções/classes**

```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

double definedFunction(double conflictBetweenNames) {
        return conflictBetweenNames * 2;
}

int FunctionName () {
        return globalVariable + definedFunction(firstVariable);
}

void conflictBetweenNames() { int nothing; }

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

Erro: java.lang.Exception: conflictBetweenNames já foi declarado localmente como variável!

- **Uso de uma variável não definida dentro de uma classe**

```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

double definedFunction(double x) {
        return x * 2;
```

```
}

int FunctionName () {
	return globalVariable + definedFunction(firstVariable);
}

void conflictBetweenNames() { int nothing; }

class A {
	public:
		void A() { x = 2; }
}

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

Erro: java.lang.Exception: As seguintes variáveis não foram definidas (ou definidas no lugar errado): x tipo: variable!

- **Uso conflitante de nomes (variável / função)**

```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

double definedFunction(double x) {
	conflictUseMode();
	return x * 2;
}

int FunctionName () {
	conflictUseMode += 2;
	return globalVariable + definedFunction(firstVariable);
}

void conflictBetweenNames() { int nothing; }

class A {
	public:
		void A() { x = 2; }
	private:
		int x;
}

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

Erro: java.lang.Exception: As seguintes variáveis não foram definidas (ou definidas no lugar errado): conflictUseMode tipo: function!

- **Quantidade errada de argumentos**

```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

double definedFunction(double x) {
        return x * 2;
}

int FunctionName () {
        return globalVariable + definedFunction(firstVariable);
}

bool wrongArgumentsAmount(int x, int y) {
        return firstVariable + definedFunction(x, y);
}

void notConflictBetweenNames() { int nothing; }

class A {
        public:
                void A() { x = 2; }
        private:
                int x;
}

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

Erro: java.lang.Exception: definedFunction é uma função com a seguinte assinatura: double(type)!

- **Programa Correto**

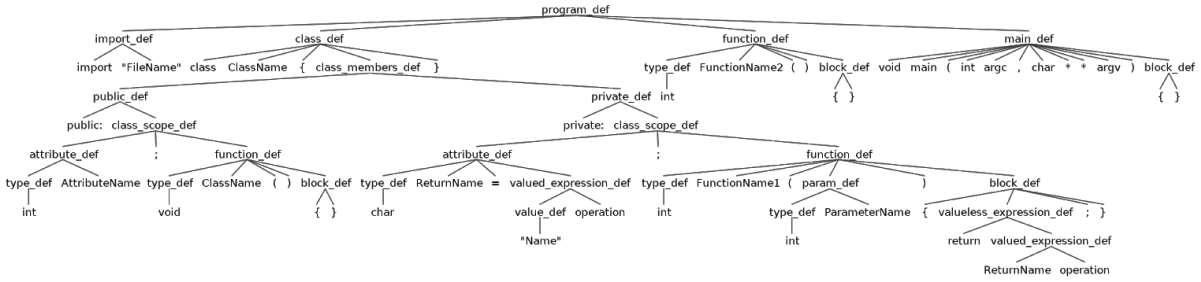
```
int firstVariable = 3;
int globalVariable = firstVariable + 1;

double definedFunction(double x) {
        return x * 2;
}

int FunctionName () {
        return globalVariable + definedFunction(firstVariable);
}
```

```
bool wrongArgumentsAmount(int x, int y) {
        return y + definedFunction(x);
}

void notConflictBetweenNames() { int nothing; }

class A {
        public:
                void A() { x = 2; }
        private:
                int x;
}

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

2. Exemplos de Programas Corretos

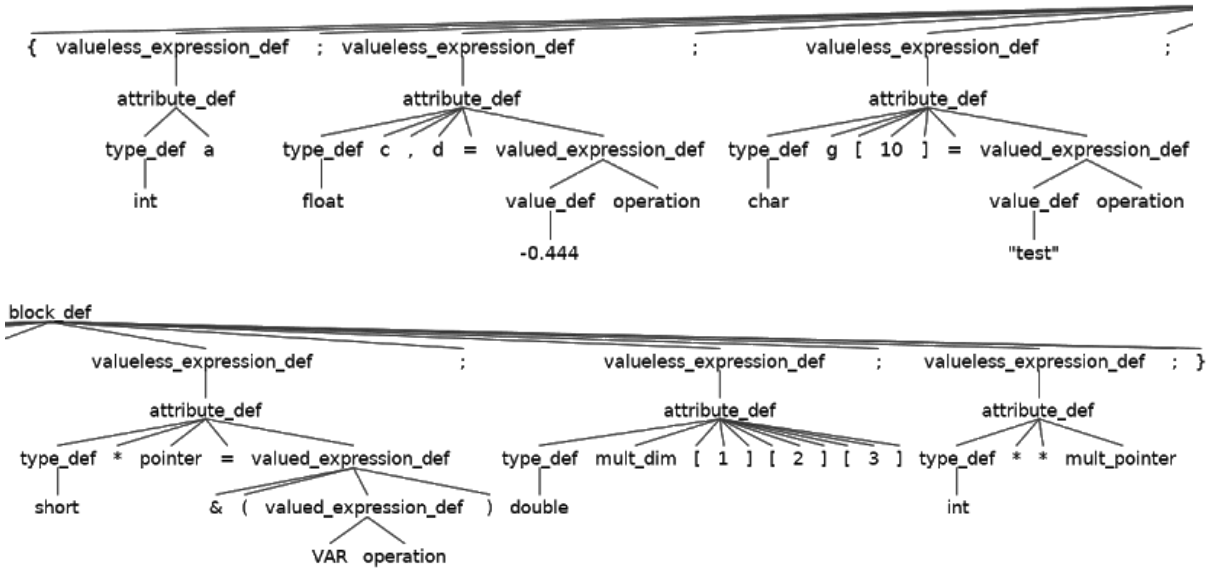
- **Estrutura**

```
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) {}
```



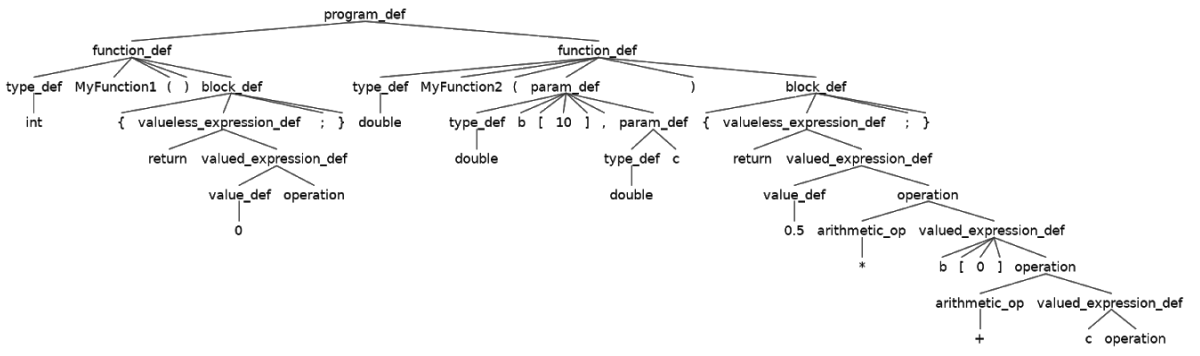
b. Declaração de variáveis e atributos

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



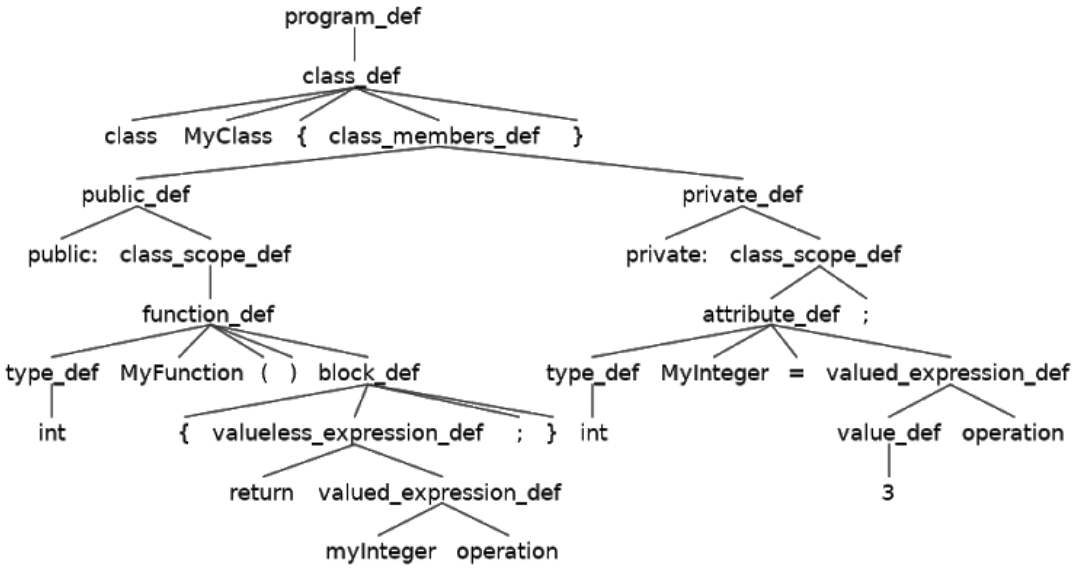
c. Declaração de funções

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



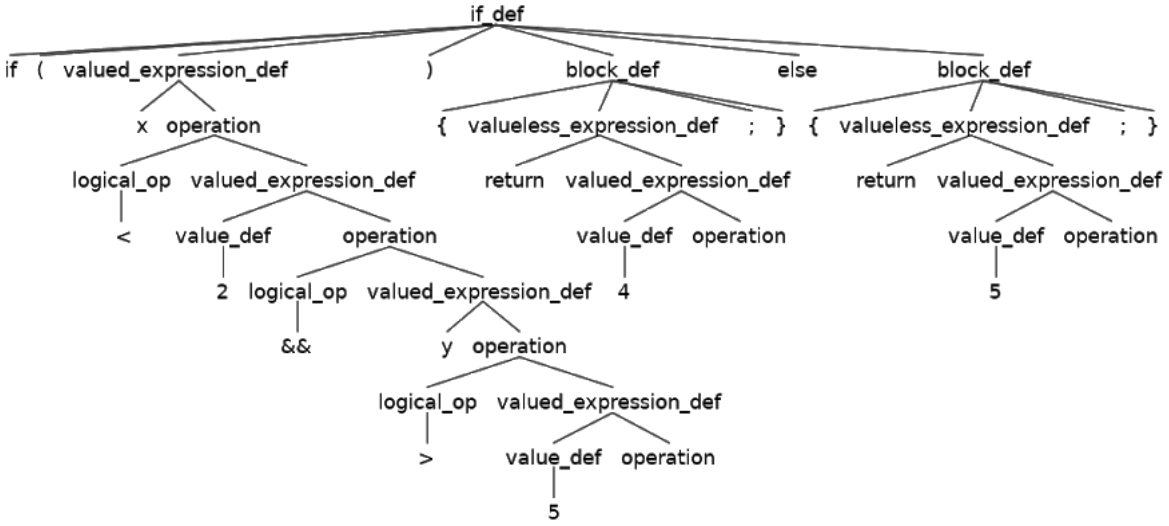
d. Definição de classes

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



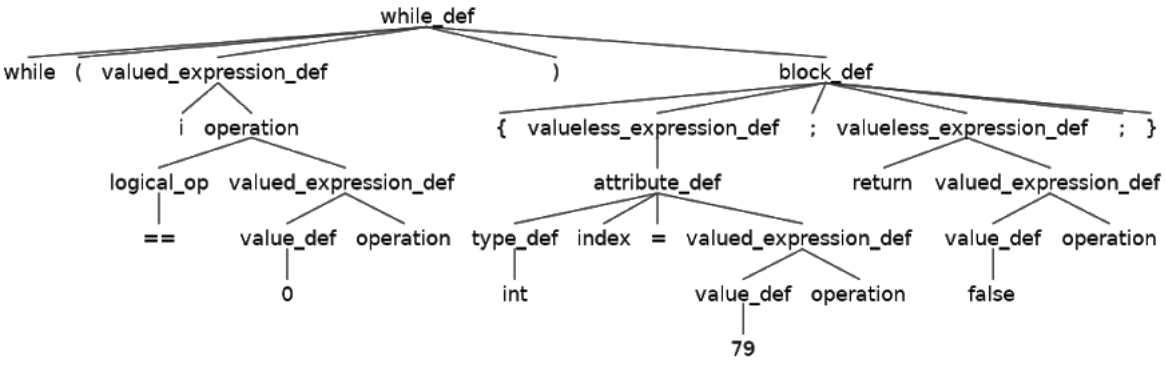
e. Estrutura de seleção if

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



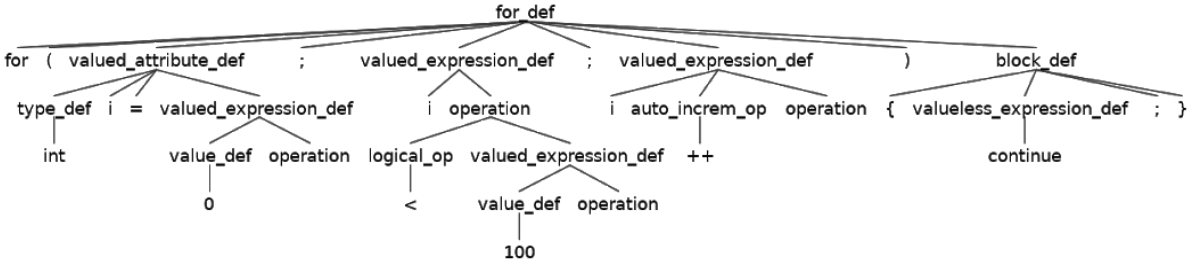
f. Estrutura de seleção while

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



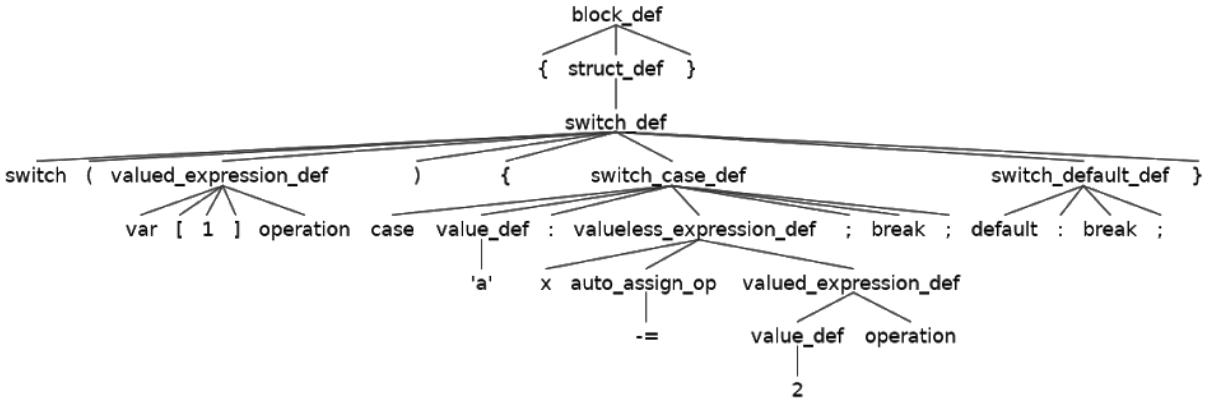
g. Estrutura de seleção for

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



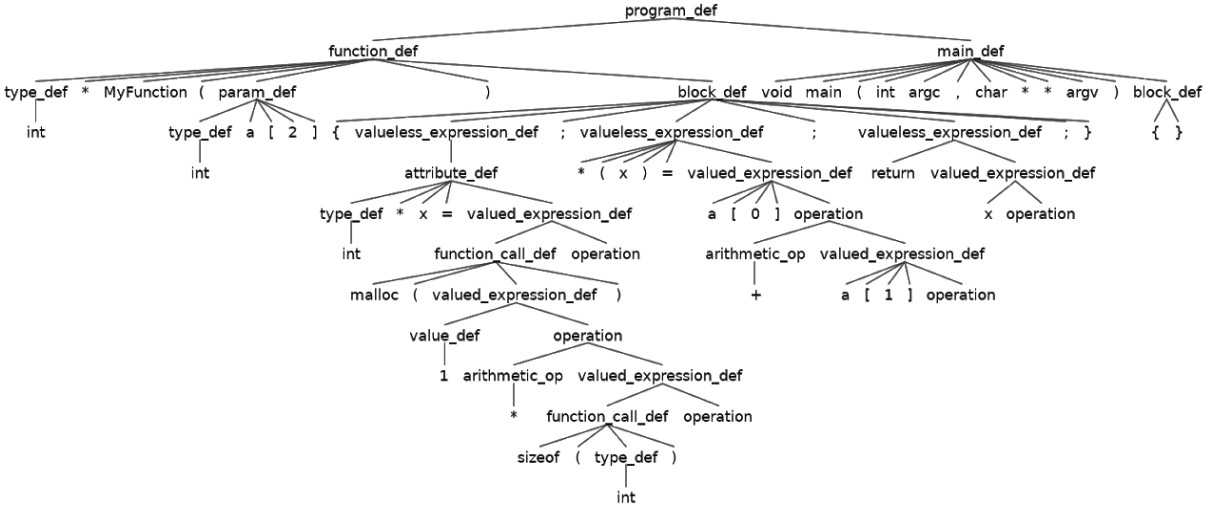
h. Estrutura de seleção switch case

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



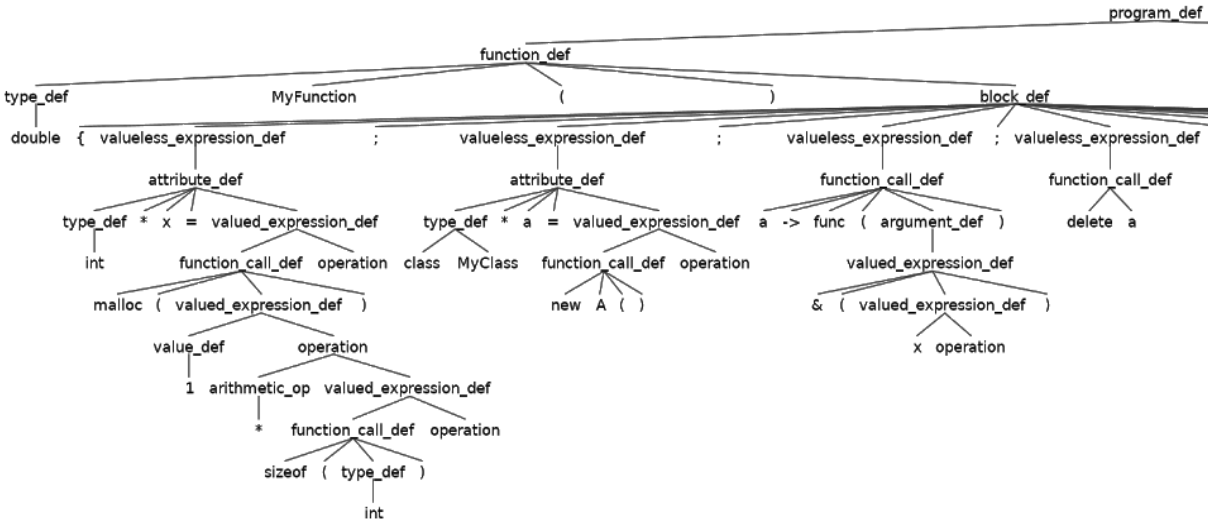
i. Ponteiros

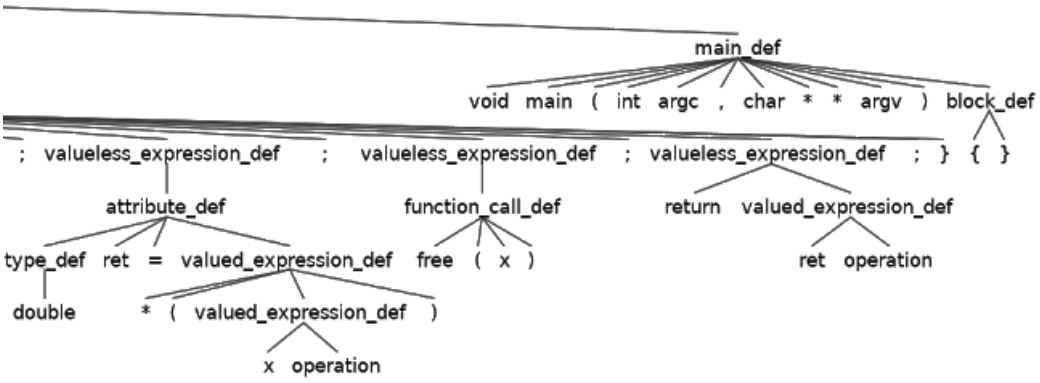
```
int * MyFunction (int a[2]) {
         int * x = malloc(1 * sizeof(int));
         *(x) = a[0] + a[1];
         return x;
}
void main (int argc, char **argv) {}
```



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) {}
```

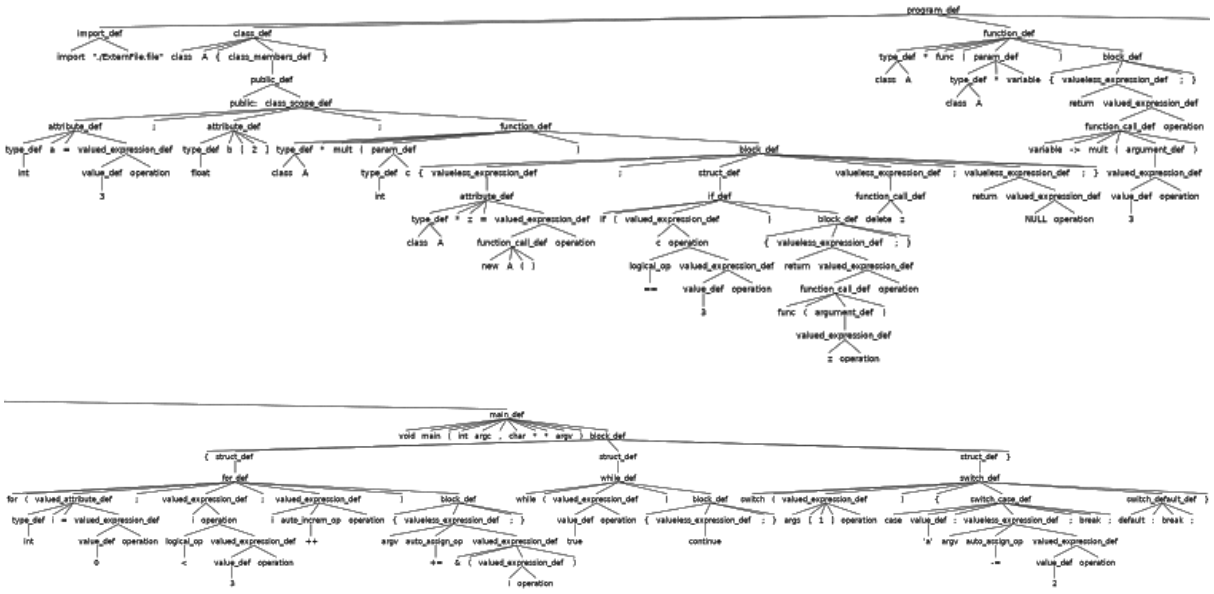




k. Programa completo

```
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;
```

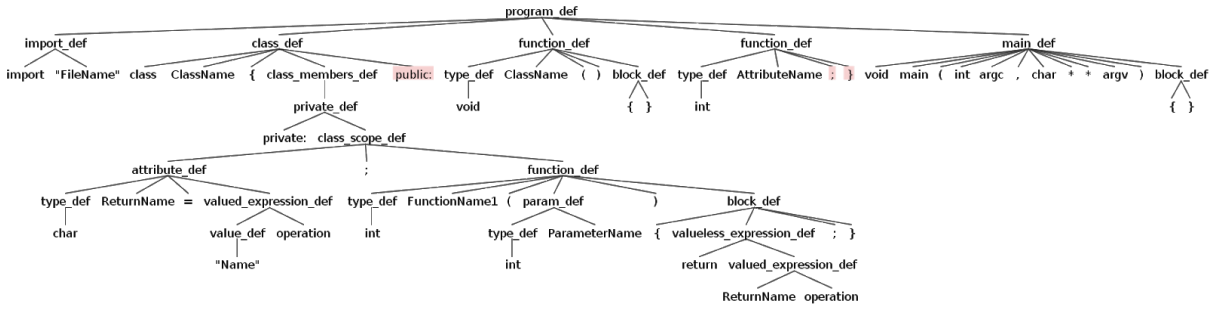
```
    }
}
```



3. Exemplos de Programas com Erros

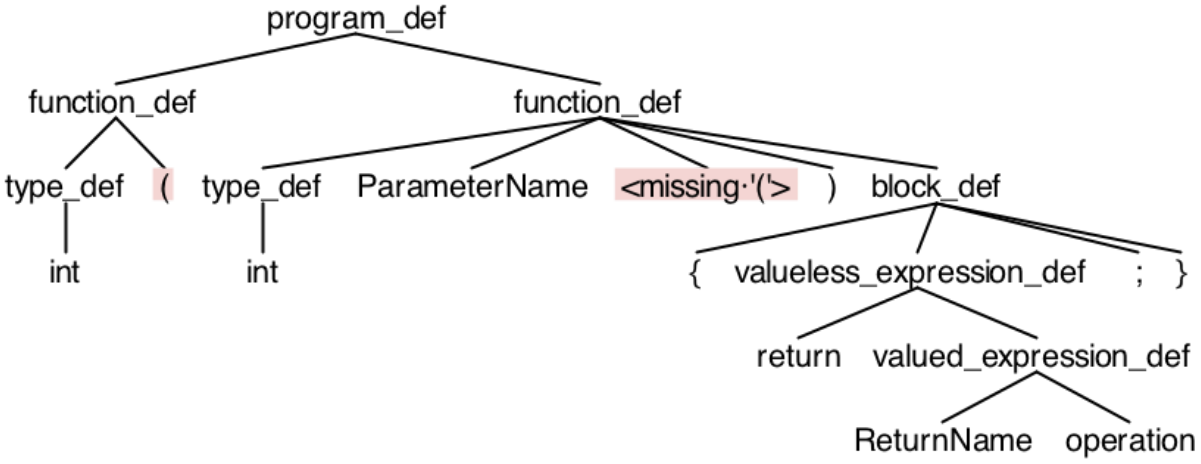
- **Incorreta organização do código**

```
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 () {}
```



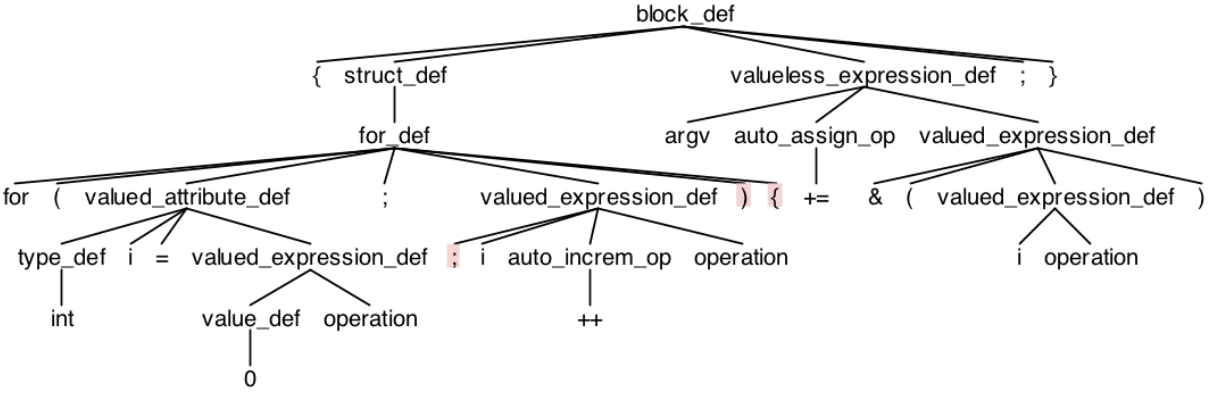
b. Função sem identificador

int (int ParameterName) { return ReturnName; }



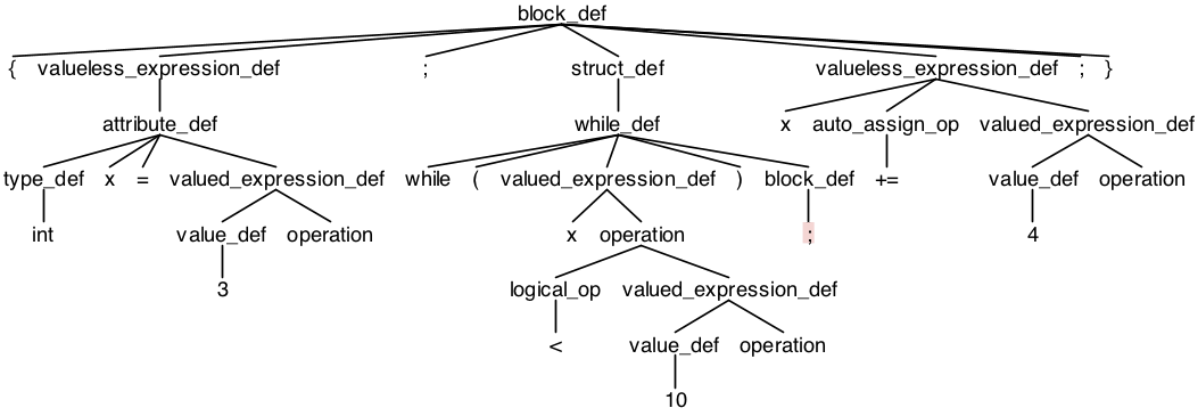
c. For sem condição de parada

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



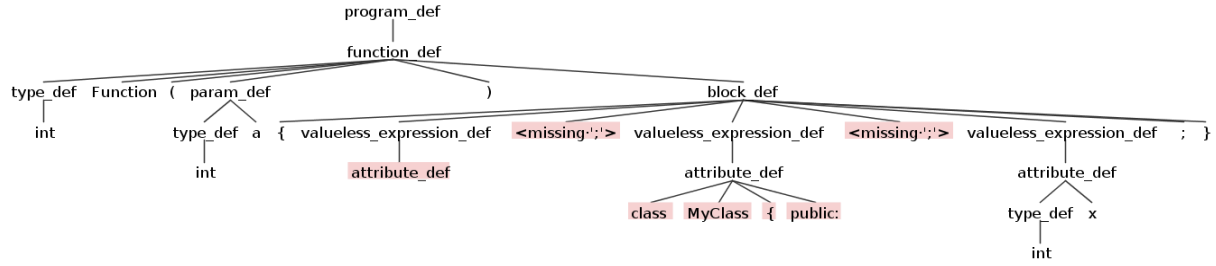
d. While sem escopo

```
{
        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;
}
```



4. Referências

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