

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

**Freedom-- : Análises Léxica, Sintática, Semântica  
e Geração de Código**

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\_BRACK INT CLOSE\_BRACK )? 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\_BRACK INT CLOSE\_BRACK)? 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\_BRACK INT CLOSE\_BRACK)? ID OPEN\_PAR param\_def? CLOSE\_PAR

block\_def ;

param\_def:

type\_def MULT ID (COMMA param\_def)\* |  
type\_def (MULT | OPEN\_BRACK INT CLOSE\_BRACK)? 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\_BRACK INT CLOSE\_BRACK) 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

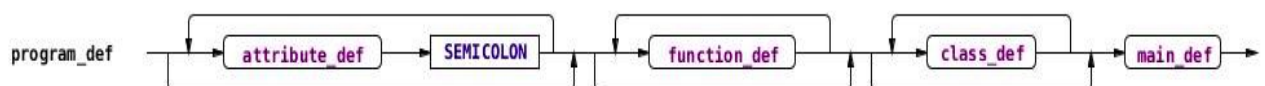
```

fragment NUMBER : '0'..'9' ;
fragment ESC    : '\ ('b' | 't' | 'n' | 'f' | 'r' ) ;

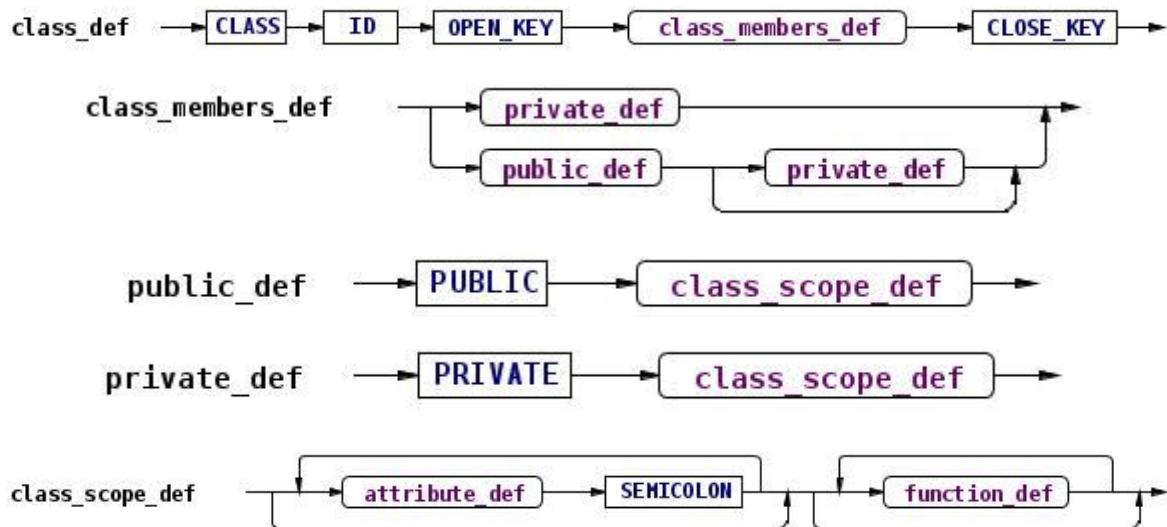
```

## b. Grafos EBNF

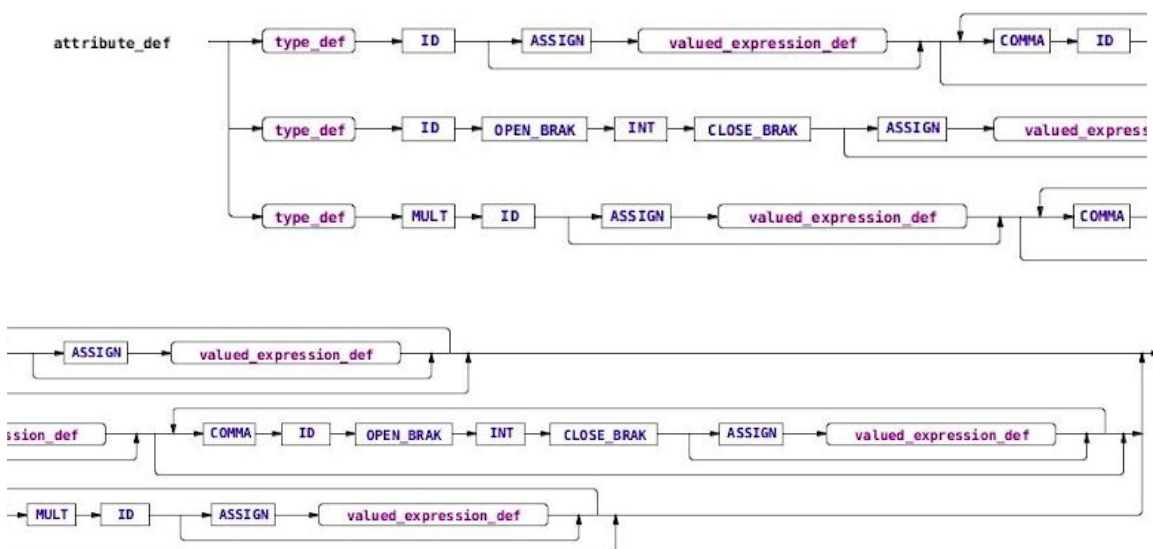
### i. Estrutura:



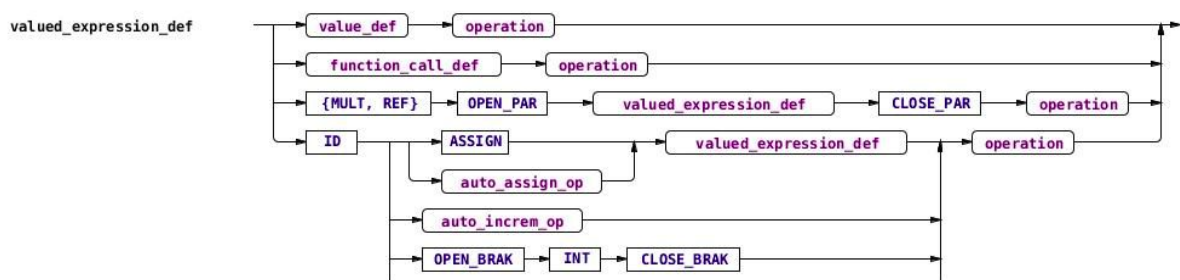
### ii. Classes:

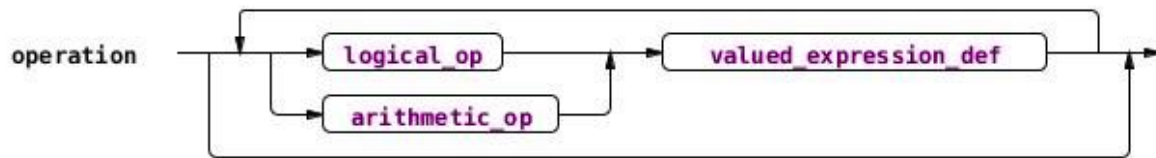


### iii. Atributos:

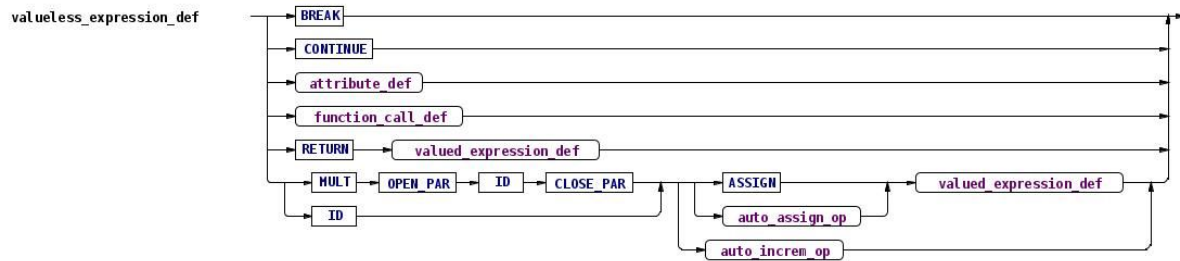


### iv. Expressão valorada:

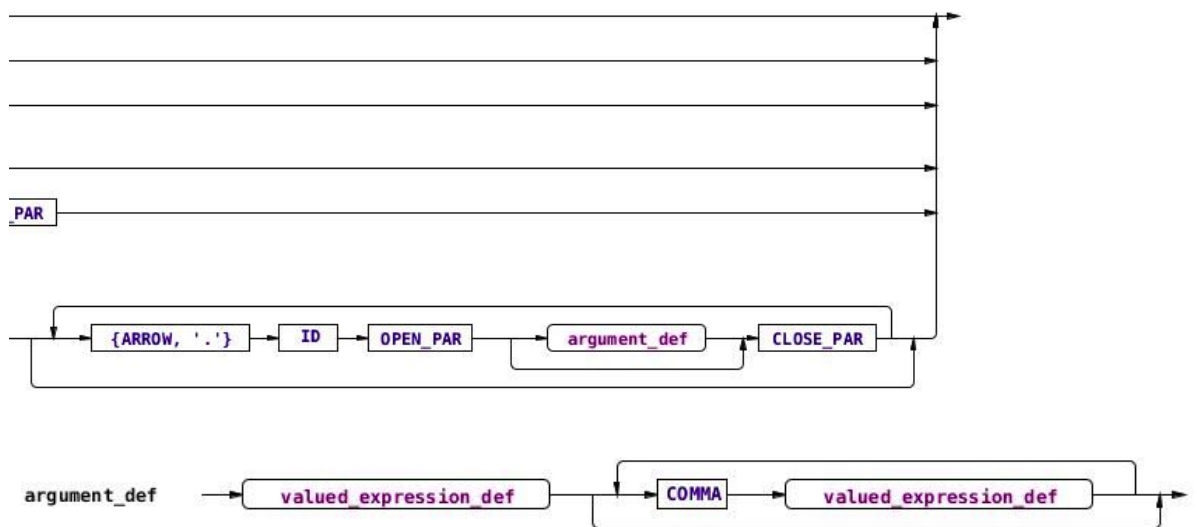
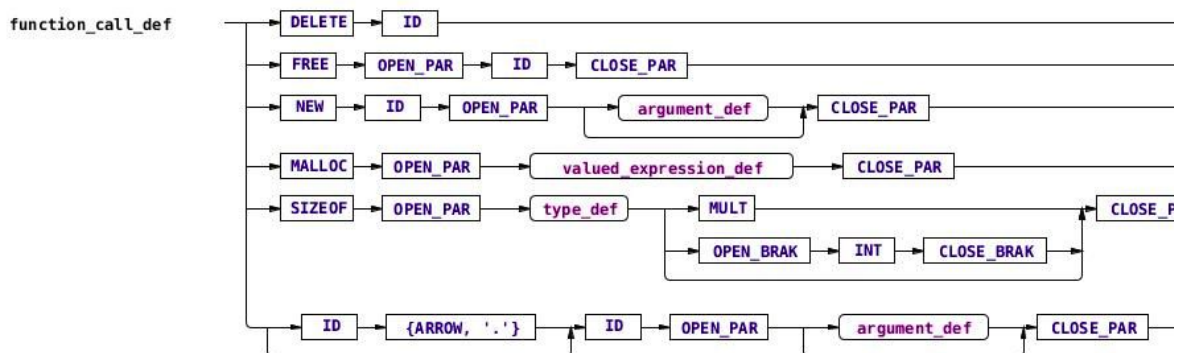




v. Expressão não valorada:

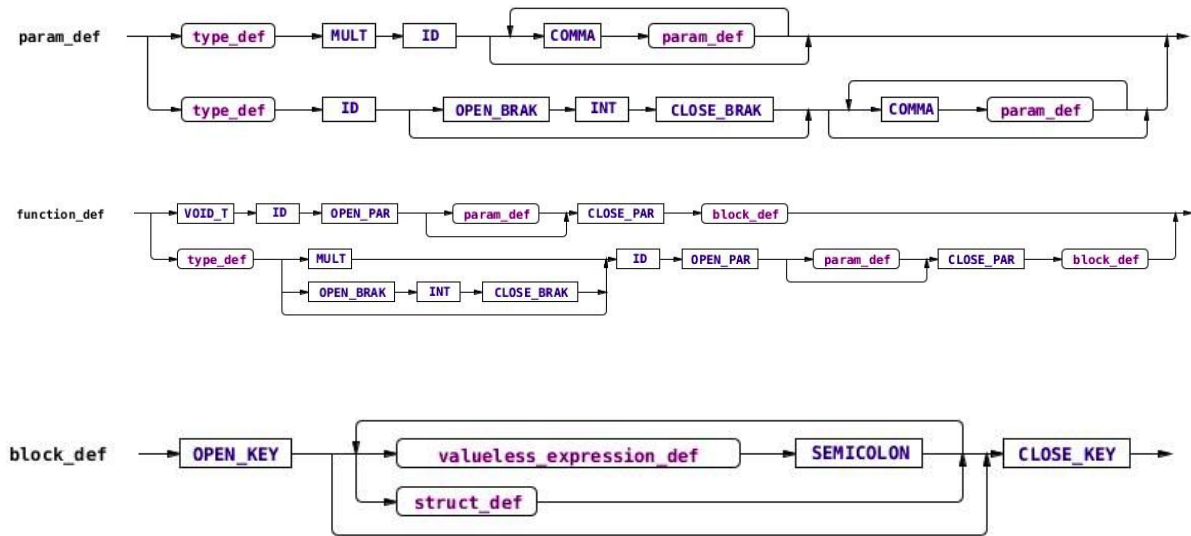


vi. Chamada de função:

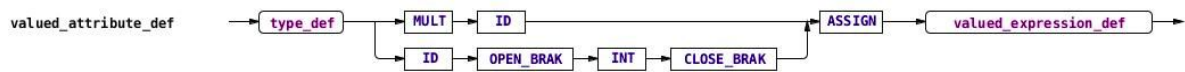




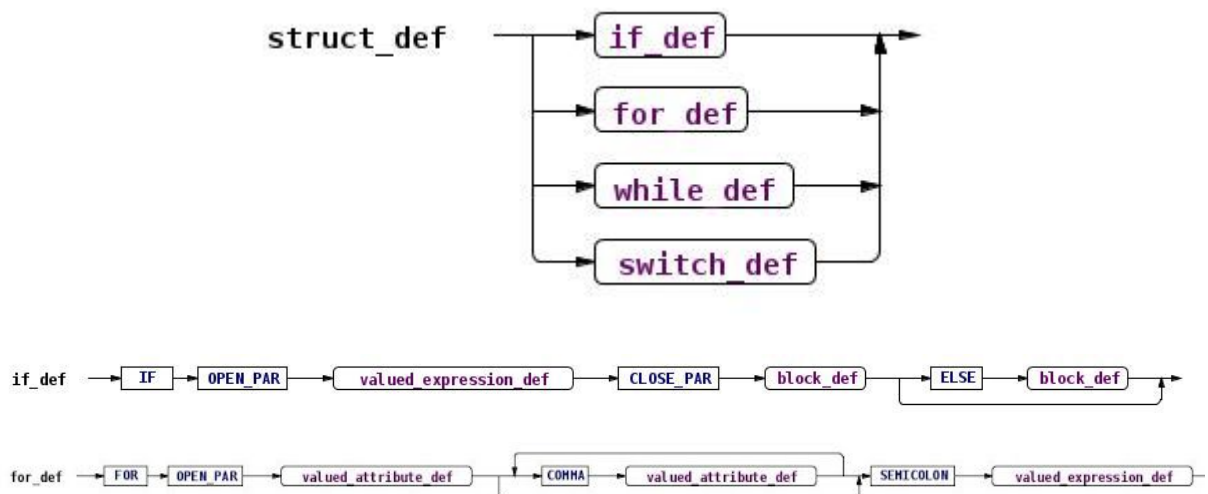
## vii. Funções:

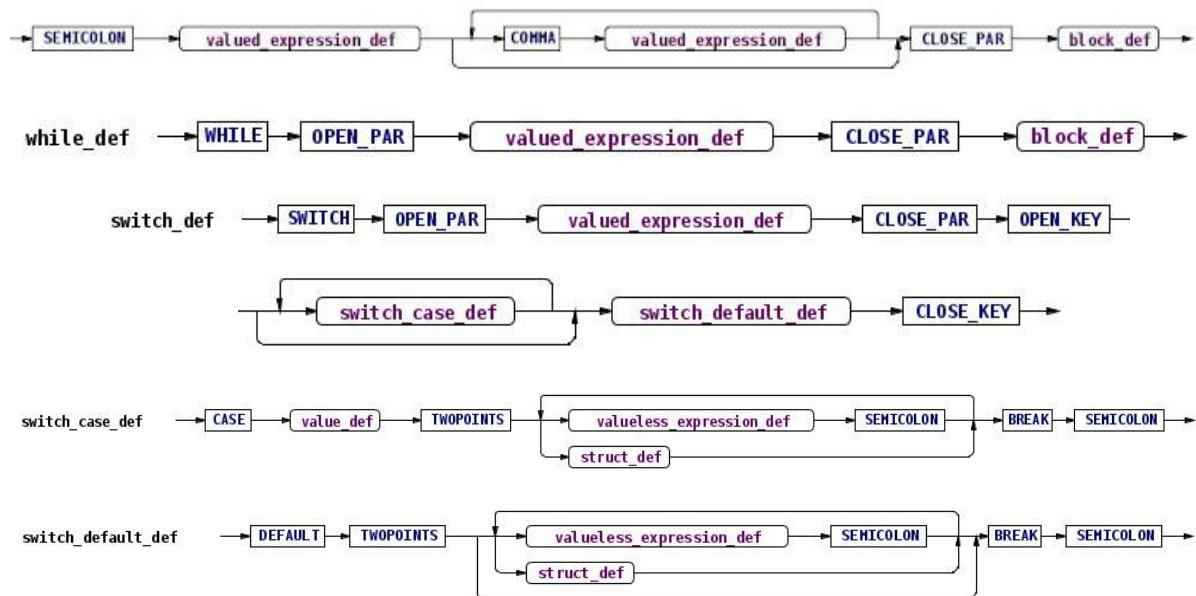


## viii. Atributo valorado:

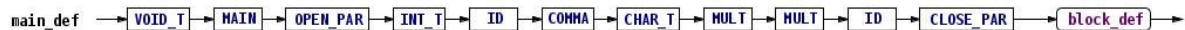


## ix. Estruturas de seleção e repetição:





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 analisador 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  $A \rightarrow B$  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
<code>program_def -&gt; (attribute_def SEMICOLON)* function_def* class_def* main_def</code>	
<code>class_def -&gt; CLASS ID OPEN_KEY class_members_def CLOSE_KEY</code>	<b>Predicado:</b> <code>!scope_contains(class_def.scope, ID.name)?</code>
<code>class_members_def -&gt; private_def</code>	
<code>class_members_def -&gt; public_def private_def?</code>	
<code>public_def -&gt; PUBLIC class_scope_def</code>	
<code>private_def -&gt; PRIVATE class_scope_def</code>	
<code>class_scope_def -&gt; (attribute_def SEMICOLON)* function_def*</code>	
<code>attribute_def -&gt; type_def ID (ASSIGN valued_expression_def)? (COMMA ID (ASSIGN valued_expression_def)?)*</code>	<b>Regra:</b> <code>attribute_def.type = type_def.type</code>  <b>Predicado:</b> <code>!scope_contains(attribute_def.scope, ID.name)?</code>  <code>attribute_def.type == valued_expression_def.type</code>
<code>attribute_def -&gt; type_def ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)? (COMMA ID OPEN_BRAK INT CLOSE_BRAK (ASSIGN valued_expression_def)?)*</code>	<b>Regra:</b> <code>attribute_def.type = type_def.type + OPEN_BRAK INT CLOSE_BRAK</code>

	<b>Predicado:</b>  !scope_contains(attribute_def.scope, ID.name)?
attribute_def -> type_def MULT ID (ASSIGN valued_expression_def)? (COMMA MULT ID (ASSIGN valued_expression_def)?)*	<b>Regra:</b>  attribute_def.type = type_def.type + MULT  <b>Predicados:</b>  !scope_contains(attribute_def.scope, ID.name)?  attribute_def.type =? valued_expression_def.type
valued_expression_def -> value_def operation	<b>Regras:</b>  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	<b>Regras:</b>  valued_expression_def.value = function_call_def.return_value operation.op operation.value  valued_expression_def.type = valued_expression_def.value.type  <b>Predicados:</b>  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	<b>Regras:</b>  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	<b>Regras:</b>  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  <b>Predicado:</b>  scope_contains(valued_expression_def.sco pe, ID.name)?
operation -> ((logical_op   arithmetic_op) valued_expression_def)*	<b>Regras:</b>  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	<b>Regra:</b>  function_call_def.return_type = 'valueless'  <b>Predicados:</b>  scope_contains(function_call_def.scope, ID.name)?  ID.type =? 'class'
function_call_def -> FREE OPEN_PAR ID CLOSE_PAR	<b>Regra:</b>  function_call_def.return_type = 'valueless'  <b>Predicados:</b>  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)*	<p><b>Regras:</b></p> <p>function_def.param_type = param_def.type</p> <p>function_def.return_type = type_def + MULT</p> <p>ID.type = 'function'</p> <p><b>Predicados:</b></p> <p>!scope_contains(function_def.scope, ID.name)?</p>
function_def -> type_def ID (OPEN_BRAK INT CLOSE_BRAK)? (COMMA param_def)*	<p><b>Regras:</b></p> <p>function_def.param_type = param_def.type</p> <p>function_def.return_type = type_def + (OPEN_BRAK INT CLOSE_BRAK)?</p> <p>ID.type = 'function'</p> <p><b>Predicados:</b></p> <p>!scope_contains(function_def.scope, ID.name)?</p>
param_def -> type_def MULT ID (COMMA param_def)*	<p><b>Regras:</b></p> <p>ID.type = type_def.type + MULT</p> <p>param_def[0].type = ID.type + (param_def[1].type)*</p> <p><b>Predicado:</b></p> <p>!scope_contains(param_def[0].scope, ID.name)?</p>
param_def -> type_def ID (OPEN_BRAK INT CLOSE_BRAK)? (COMMA param_def)*	<p><b>Regras:</b></p> <p>ID.type = type_def.type + (OPEN_BRAK INT CLOSE_BRAK)?</p> <p>param_def[0].type = ID.type + (param_def[1].type)*</p> <p><b>Predicado:</b></p> <p>!scope_contains(param_def.scope,</p>

	ID.name)?
block_def -> OPEN_KEY (valueless_expression_def SEMICOLON   struct_def)* CLOSE_KEY	<b>Regra:</b>  block_def.type = verify_return(valueless_expression_def, struct_def)
valueless_expression_def -> BREAK	<b>Regra:</b>  valueless_expression_def.type = 'break'
valueless_expression_def -> CONTINUE	<b>Regra:</b>  valueless_expression_def.type = 'continue'
valueless_expression_def -> attribute_def	<b>Regras:</b>  valueless_expression_def.type = attribute_def.type  valueless_expression_def.value = attribute_def.value
valueless_expression_def -> function_call_def	<b>Regras:</b>  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	<b>Regras:</b>  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)	<b>Regra:</b>  valueless_expression_def.type = 'valueless'  <b>Predicados:</b>  scope_contains(valueless_expression_def. scope, ID.name)  ID.type =? valued_expression_def.type

struct_def -> if_def	<b>Regra:</b> struct_def.type = if_def.type
struct_def -> for_def	<b>Regra:</b> struct_def.type = for_def.type
struct_def -> while_def	<b>Regra:</b> struct_def.type = while_def.type
struct_def -> switch_def	<b>Regra:</b> struct_def.type = switch_def.type
if_def -> IF OPEN_PAR valued_expression_def CLOSE_PAR block_def (ELSE block_def)?	<b>Regra:</b> 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	<b>Regra:</b> for_def.type = verify_return(block_def)
valued_attribute_def -> type_def (MULT ID   ID OPEN_BRAK INT CLOSE_BRAK) ASSIGN valued_expression_def	<b>Regra:</b> 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  <b>Predicados:</b> !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	<b>Regra:</b>

	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	<b>Regras:</b>  switch_def.conditional_type = valued_expression_def.type  switch_default_def.type = verify_return(switch_case_def, switch_default_def)  <b>Predicado:</b>  switch_case_def.conditional_type =? switch_def.conditional_type
switch_case_def -> CASE value_def TWOPOINTS (valueless_expression_def SEMICOLON   struct_def)+ BREAK SEMICOLON	<b>Regras:</b>  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	<b>Regra:</b>  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	<b>Regra:</b>  type_def.type = 'int'
type_def -> DOUBLE_T	<b>Regra:</b>  type_def.type = 'double'
type_def -> CHAR_T	<b>Regra:</b>  type_def.type = 'char'
type_def -> BOOL_T	<b>Regra:</b>

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

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

	auto_assign_op.op = '-='
auto_assign_op -> AUTOMULT	<b>Regra:</b> auto_assign_op.op = '*='
auto_assign_op -> AUTODIV	<b>Regra:</b> auto_assign_op.op = '/='
auto_assign_op -> AUTOPLUS	<b>Regra:</b> auto_assign_op.op = '+='
auto_increm_op -> INCREM	<b>Regra:</b> auto_increm_op.op = '++'
auto_increm_op -> DECREM	<b>Regra:</b> 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 criada 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:** 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:** 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:** 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:** `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:** 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:** 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:** `definedFunction` é uma função com a seguinte assinatura: `double(type)!`

- **Diversos Erros**

```
int global = notDeclared + 10;

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

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

class NotDeclared {
    private:
```

```

        double global;
    }

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

```

#### Erros:

- Variável **notDeclared** está sendo usada antes de ser declarada!
- Variável **globalVariable** está sendo usada antes de ser declarada!
- Variável **firstVariable** está sendo usada antes de ser declarada!
- **global** já foi declarado globalmente como variável!

#### ● 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) {}

```

## 7. Geração de Código

A geração de código intermediário não é um processo fornecido automaticamente pelo ANTLR. Em razão disso, houve a necessidade de utilizar

uma ferramenta alternativa que facilitasse esta etapa de implementação do compilador. Dentre as alternativas analisadas, LLVM foi a escolhida devido a sua curva de aprendizado mais suave e sintaxe próxima ao assembly do MIPS, da qual os membros do grupo já tinham conhecimento, diferentemente das demais opções disponíveis.

#### **a. Código intermediário do LLVM**

A representação do código intermediário utilizada pela ferramenta é realizada através de uma linguagem de programação de baixo nível, muito parecida com *assembly*. Assim como a linguagem deste trabalho, a linguagem do LLVM é fortemente tipada. Outra característica interessante da ferramenta LLVM são os formatos das instruções, usando a abordagem RISC, com um conjunto de instruções simples utilizando três endereços.

#### **b. Implementação através do padrão *Visitor***

Para implementar a geração de código foi utilizada a interface *Visitor* construída automaticamente pelo ANTLR, sendo esta implementada manualmente para o processo de geração do código intermediário. A principal utilidade de usar o *Visitor* é a facilidade de obter informações dos nós da árvore de sintaxe. Isso é possível uma vez que a análise léxica, sintática e semântica são realizadas antes da execução do *Visitor* que, por sua vez, realiza uma busca em profundidade partindo do nó raiz da árvore previamente construída.

Devido a escolha de gerar o código manualmente, a complexidade do trabalho forçou a equipe a implementar apenas operações com inteiros de 32 bits (*i32*). Por outro lado, o foco do desenvolvimento foi criar todas as estruturas básicas existentes em praticamente todas as linguagens de programação para que funcionassem com variáveis inteiras, dentre elas as estruturas *if*, *for*, *while* e até mesmo funções. Deste modo, foi possível exercitar quase todos os tópicos de geração de código através do LLVM, ficando de fora apenas as estruturas e operações mais complexas da linguagem FreedomLessLess, como classes e alocação dinâmica.

A seguir são explicadas alguns dos métodos implementados para a geração de código intermediário.

### c. *Visitor visitFunction\_def(...)*

A representação intermediária de uma função começa pela identificação do seu tipo e do nome, este último deve ser precedido pelo símbolo @. Em seguida, há a verificação da existência de parâmetros onde, caso existam, será executada a função *accept(...)* chamando o *Visitor* para gerar o código que representará os parâmetros da função. Após, a representação do bloco da função (na qual possui a mesma lógica da geração do código dos parâmetros), é retornada a String resultante da concatenação dos resultados de cada etapa.

```
String type = ctx.type_def().accept(this);
String name = "@" + ctx.ID().getText();

String param = "";
if (ctx.param_def() != null)
    param = ctx.param_def().accept(this);

String block = ctx.block_def().accept(this);

_types.put(name, type);

String code = "\n define " + type + " " + name + "(" + param + ") {\n";
code += "entry:\n";
code += block;
code += "}\n\n";

return code;
```

### d. *Visitor visitParam\_def(...)*

O processo de geração do código intermediário dos parâmetros da função utiliza um método simples. Inicialmente existe a necessidade de identificar o tipo e nome dos parâmetros através da função *accept(...)*. Após a identificação do primeiro parâmetro, é preciso verificar se essa função possui outros parâmetros e, caso

exista, a função *accept(...)* chamará este método do *Visitor* que continuará o reconhecimento até não haver mais parâmetros. O método irá concatenar a representação do código intermediário e retornará a ao método *visitFunction\_def(...)*.

```
String type = ctx.type_def().accept(this);
String name = "%" + ctx.ID().getText();

_types.put(name, type);
_is_register.put(name, true);

String code = type + " " + name;

if (ctx.param_def(0) != null)
    code += ", " + ctx.param_def(0).accept(this);

return code;
```

#### **e. *Visitor visitFor\_def(...)***

A lógica da geração do código intermediário da estrutura de repetição *for* inicia subindo na árvore até o nodo pai (*block\_def*). Posteriormente acontece a geração do código das expressões de inicialização, checagem do momento de parar a o loop e incremento da variável já inicializada. Em seguida, é realizada a lógica para alterar entre seções do código, pois o *for* exige isso. Após o fluxo de execução de um *for* iniciar com o carregamento da expressão de inicialização, se faz necessário utilizar um label para ir até a seção de código que tem a expressão de comparação. Essa sessão de código também utiliza um label para se direcionar até o segmento que incrementa a variável inicializada anteriormente e uma *string* com essa lógica é retornada.

```
String block = ctx.block_def().accept(this);

String def = ctx.valued_attribute_def(0).accept(this);

String inc = ctx.valued_expression_def(1).accept(this);

String condition = ctx.valued_expression_def(0).accept(this);
```

```

String tmp_cond = _current_var;
_types.put(tmp_cond, "i1");

String code = def;

String labelCond = "label" + _label_number++;
String labelLoop = "label" + _label_number++;
String labelEndLoop = "label" + _label_number++;

code += "br label %" + labelCond + "\n";
code += "ret i32 0\n";

code += labelCond + ":\n";

code += condition;
code += "br i1 " + tmp_cond + ", label %" + labelLoop + ", label %" + labelEndLoop + "\n";
code += "ret i32 0\n";

code += labelLoop + ":\n";
code += block;
code += inc;

code += "br label %" + labelCond + "\n";
code += "ret i32 0\n";

code += labelEndLoop + ":\n";

return code;

```

## f. Exemplo

Como exemplo de geração de código, temos abaixo a função main que tem o comportamento de simplesmente incrementar o valor de uma variável enquanto o valor dessa variável é inferior a 100.

```

void main(int argc, char ** argv) {
    int i = 0;
    while (i < 100) { i = i + 1; }
}

```



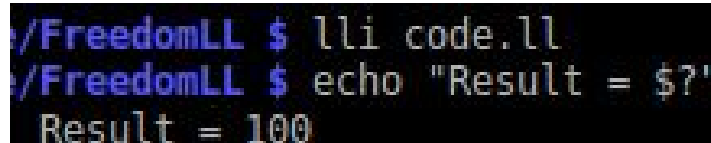
```
    return i;  
}
```

Após a execução do compilador sobre esse programa obtemos o código intermediário do LLVM, o qual pode ser visto abaixo:

```
define i32 @main() {  
    entry:  
  
    %i = alloca i32  
    store i32 1, i32* %i  
    br label %label0  
  
    ret i32 0  
label0:  
  
    %tmp2 = load i32, i32* %i  
    %tmp3 = icmp slt i32 %tmp2, 100  
  
    br i1 %tmp3, label %label1, label %label2  
  
    ret i32 0  
label1:  
  
    %tmp0 = load i32, i32* %i  
    %tmp1 = add i32 %tmp0, 1  
    store i32 %tmp1, i32* %i  
  
    br label %label0  
  
    ret i32 0  
label2:  
  
    %tmp4 = load i32, i32* %i  
    ret i32 %tmp4  
}
```

Através do comando *lli* do LLVM é possível executar um compilador *just-in-time* que realizará a interpretação e execução do *bytecode* do LLVM,

permitindo obter o resultado através do comando `echo "Result = $?"` que, por sua vez, imprime o valor de saída do último comando executado, neste caso, o resultado da execução do programa acima, assim como pode ser visto na imagem abaixo.



```
/FreedomLL $ lli code.ll
/FreedomLL $ echo "Result = $?"
Result = 100
```

#### g. Função *print(...)*

Embora não tenha sido possível implementar funções para imprimir *strings* durante a execução dos programas, foi realizado uma pesquisa para a viabilidade de imprimir números inteiros de 32 bits. Abaixo é mostrado um exemplo simples de como essa característica foi implementada no LLVM:

```
@.pstr = private unnamed_addr constant [4 x i8] c"%u\0A\00"
declare i32 @printf(i8*, ...)
define void @print(i32 %i) {
    call i32 (i8*, ...) @printf(i8* @getelementptr inbounds
                                ([4 x i8], [4 x i8]* @.pstr, i32 0, i32 0), i32 %i)
    ret void
}
```

onde “[4 x i8]” especifica o tamanho da *string* sendo impressa (4 caracteres de 8 *bits*) e “c” especifica o formato de impressão (*character*) e o padrão especial “%u” especifica o local onde o valor será inserido na *string* impressa. Deste modo, ao criar uma função vazia chamada *print*, o usuário poderá chama-lá para imprimir o valor de uma variável no terminal.

Para que essa impressão pudesse ser realizada “automaticamente” para qualquer tipo de variável no código feito pelo usuário, bastaria alterar a chamada da função *print*, função auxiliar para o caso específico implementado, para *printf* e verificar o nome da função sendo chamada durante a leitura realizada pelo parser.

Caso fosse um *printf(...)* seriam criados os códigos necessários de acordo com a função *printf* do biblioteca da linguagem c.

O código abaixo exemplifica como a impressão de valores inteiros é realizada:

```
int print(int i) { }

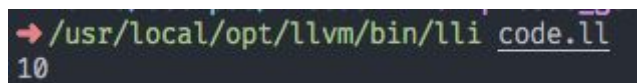
void main(int argc, char ** argv) {
    print(10);
    return 0;
}
```

Após a execução do compilador sobre esse programa obtemos o código intermediário do LLVM, o qual pode ser visto abaixo:

```
@.pstr = private unnamed_addr constant [4 x i8] c"%u\0A\00"
declare i32 @printf(i8*, ...)
define void @print(i32 %i) {
    call i32 (i8*, ...) @printf(i8* @getelementptr inbounds
                                ([4 x i8], [4 x i8]* @.pstr, i32 0, i32 0), i32 %i)
    ret void
}

define i32 @main() {
entry:
    call void @print(i32 10)
    ret i32 0
}
```

E a saída é mostrada na imagem abaixo:



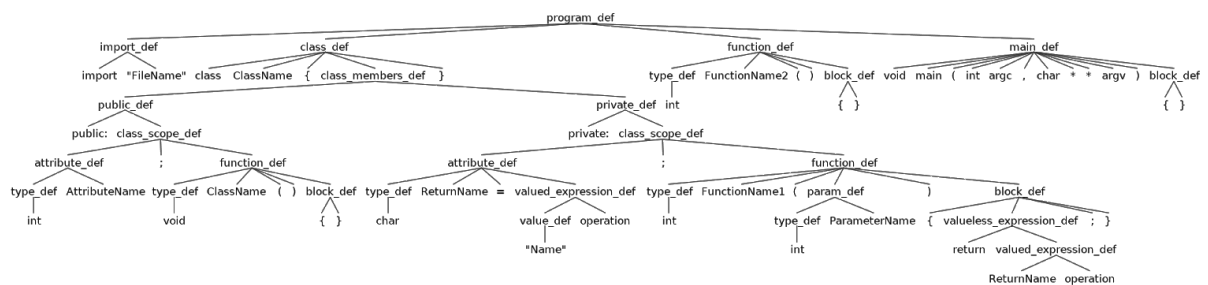
```
→ /usr/local/opt/llvm/bin/lli code.ll
10
```

## 8. Anexos

- **Programas Semanticamente Corretos**

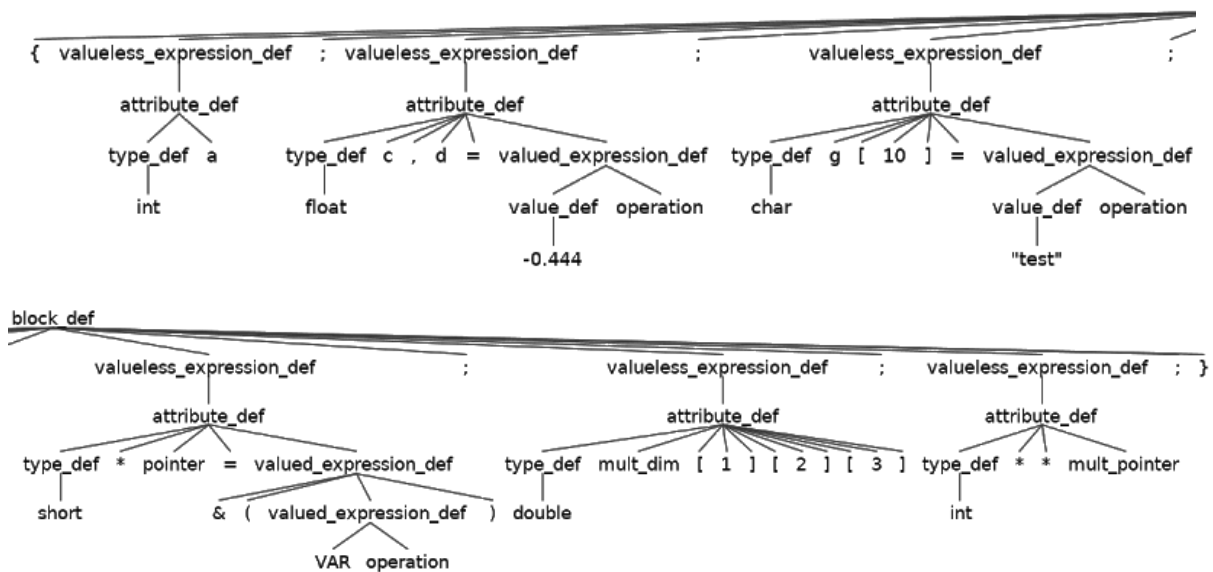
- **Estrutura**

```
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) {}
```



- **Declaração de variáveis e atributos**

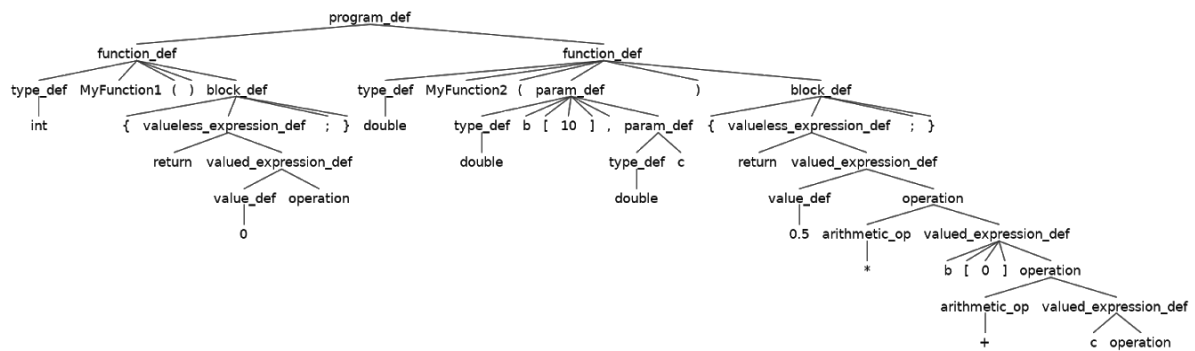
```
{
    int a;
    float c, d = -0.444;
    char g[10] = "test";
    short * pointer = &(VAR);
    double mult_dim[1][2][3];
    int ** mult_pointer;
}
```



### ○ Declaração de funções

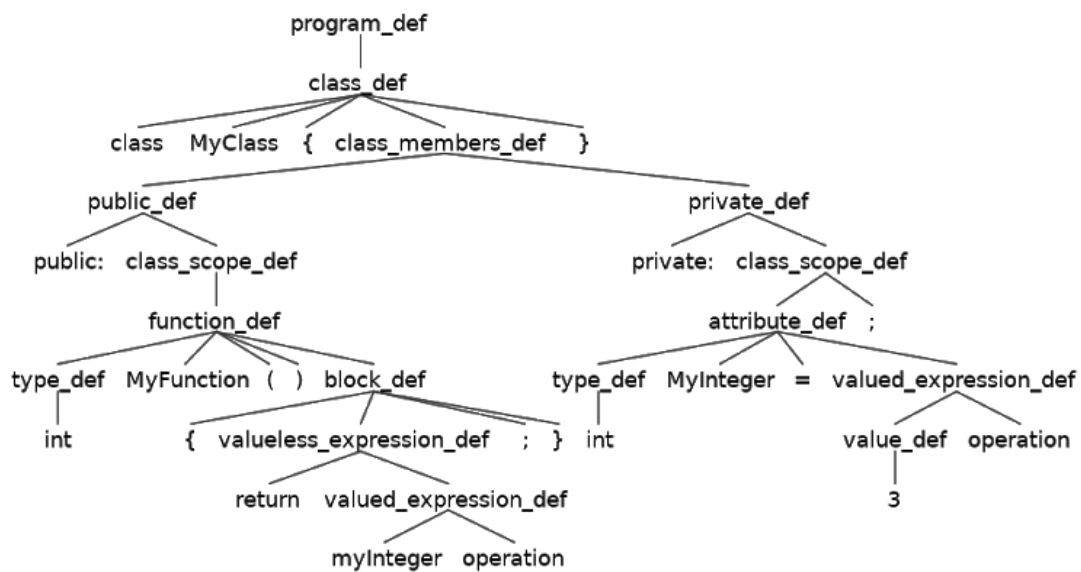
```
int MyFunction1 () { return 0; }
```

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



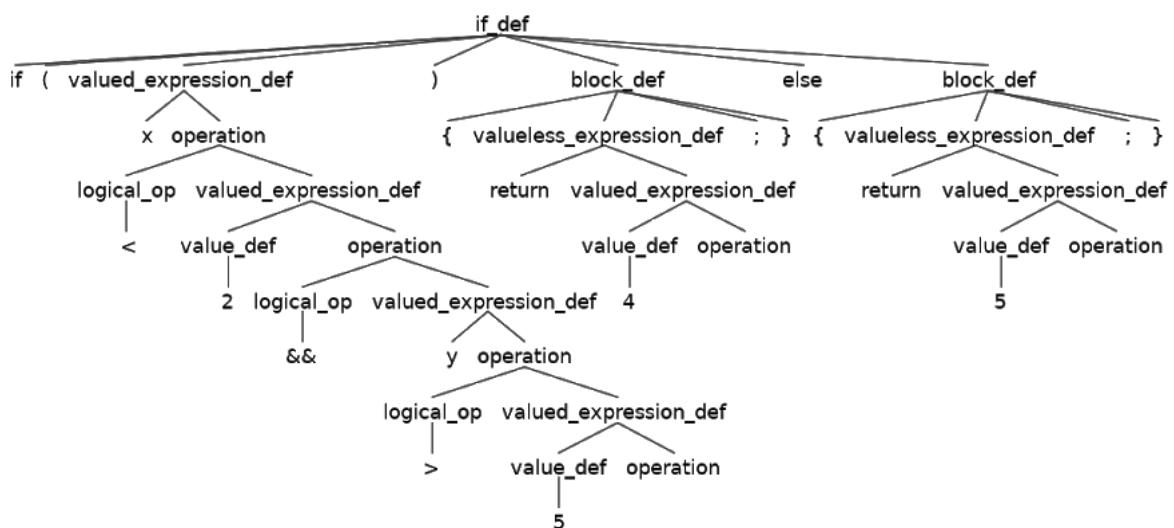
### ○ Definição de classes

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



### ○ Estrutura de seleção if

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



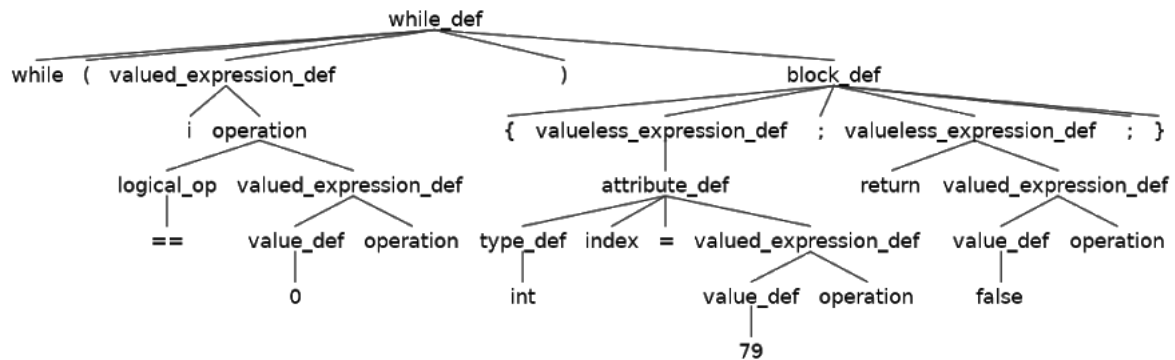
### ○ Estrutura de seleção while

```
while (i == 0) {
```

```

int index = 79;
return false;
}

```

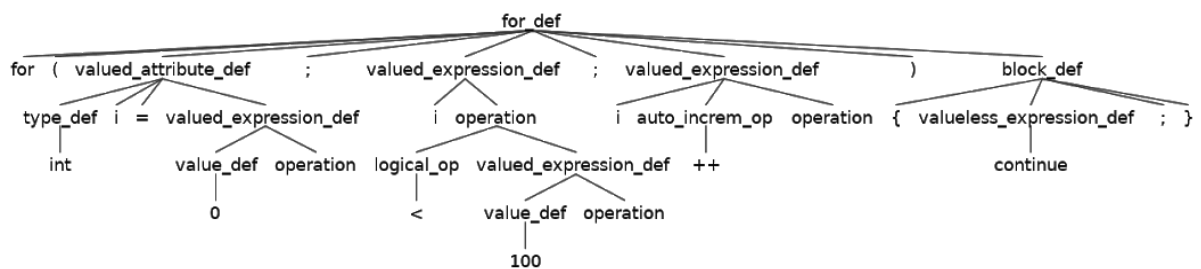


### ○ Estrutura de seleção for

```

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

```

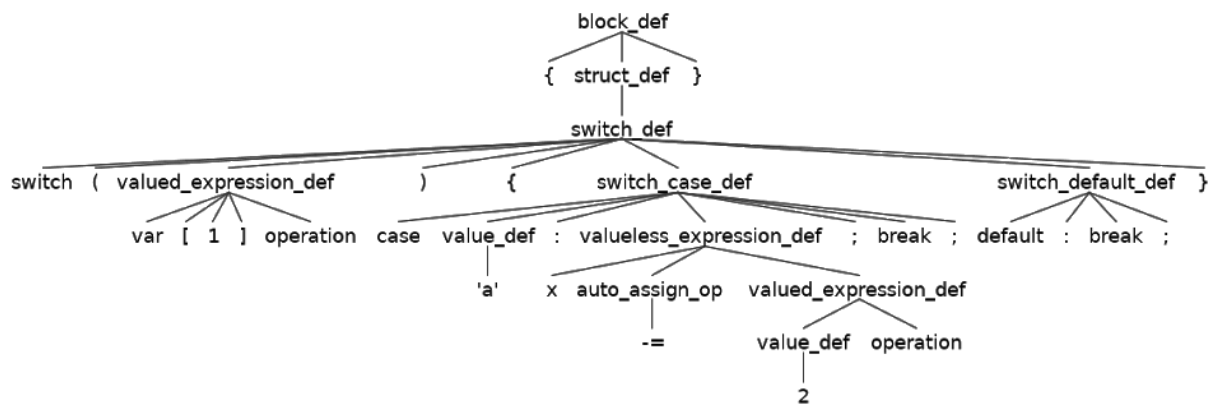


### ○ Estrutura de seleção switch case

```

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

```

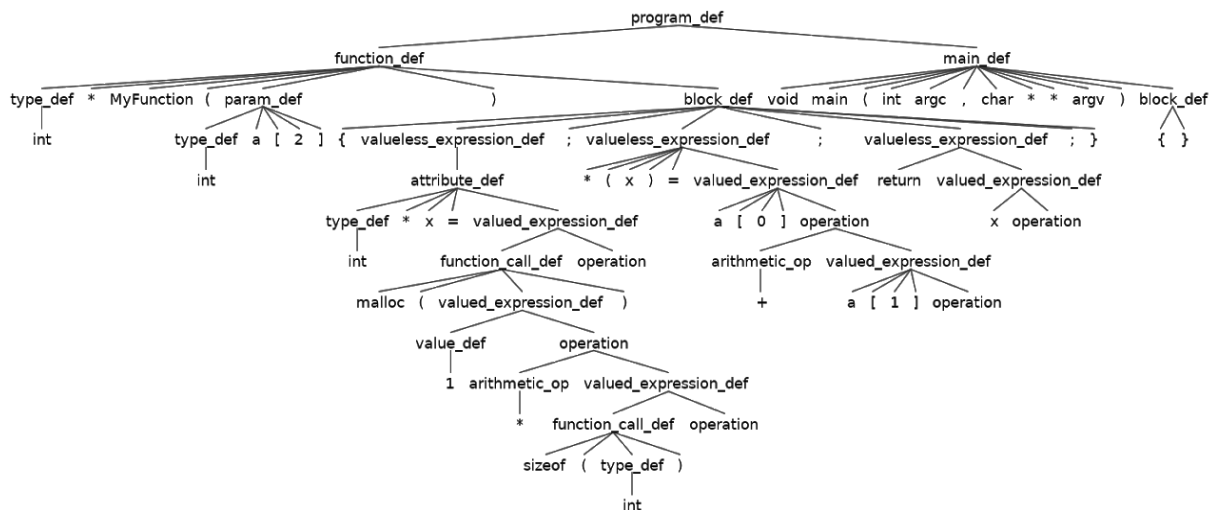


### ○ Ponteiros

```

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

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

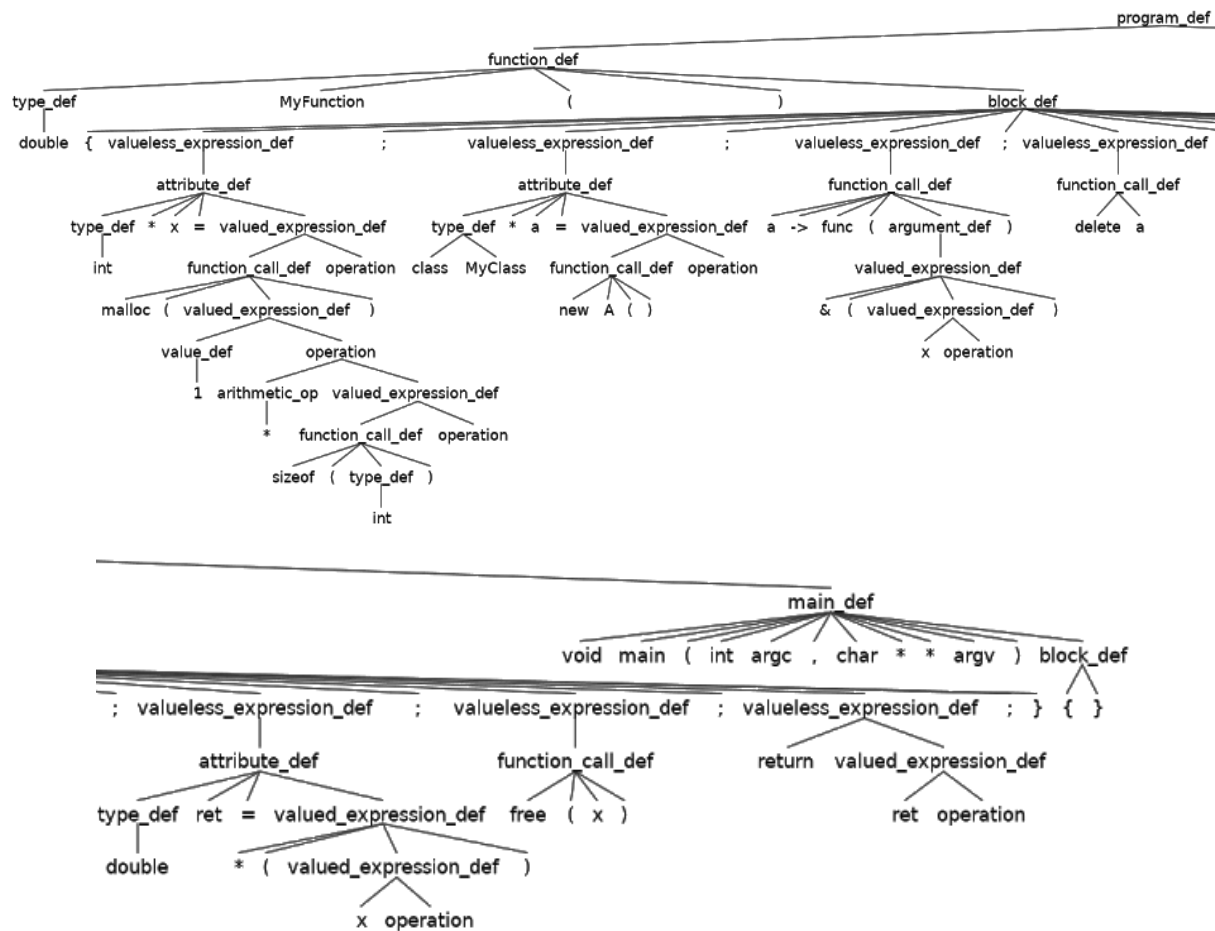


### ○ Alocação dinâmica de memória

```

double MyFunction () {
    int * x = malloc(1 * sizeof(int));
    class MyClass * a = new A();
    a->func(&(x));
    delete a;
}
  
```





- **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);
            }
        }
    }
}
```

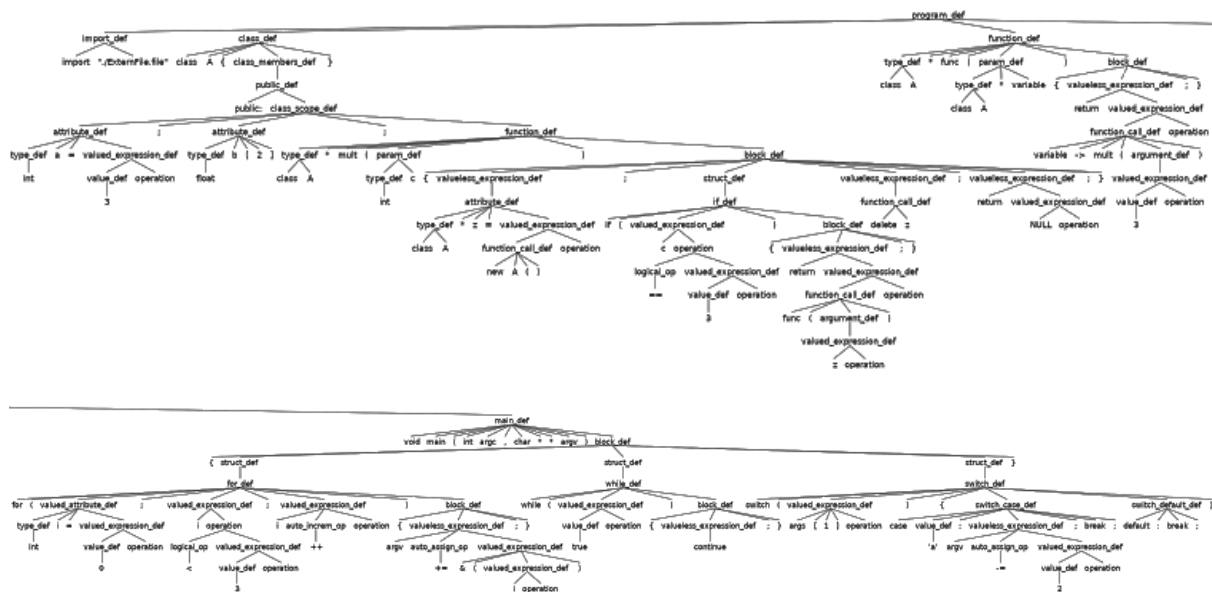
```

        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;
    }
}

```



## ● Exemplos de Programas Semanticamente Incorretos

### ○ Incorreta organização do código

```

import "FileName"
class ClassName {

```

```

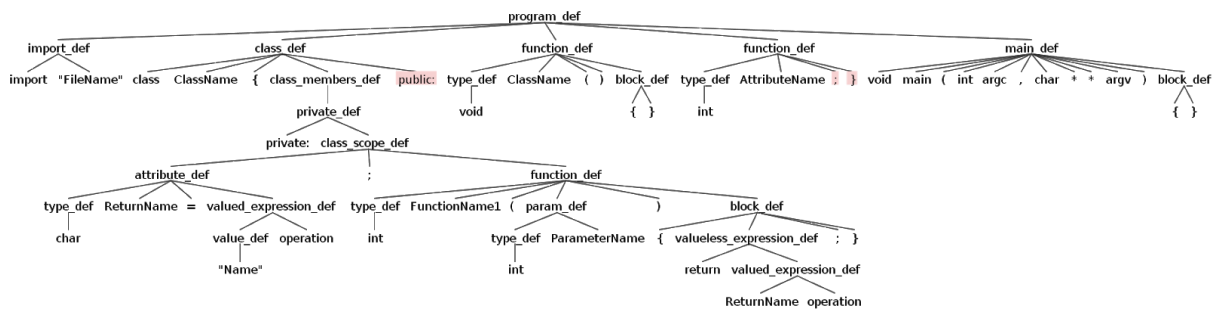
private:
    char ReturnName = "Name";
    int FunctionName1 (int ParameterName) { return ReturnName; }

public:
    void ClassName () {}
    int AttributeName;

}

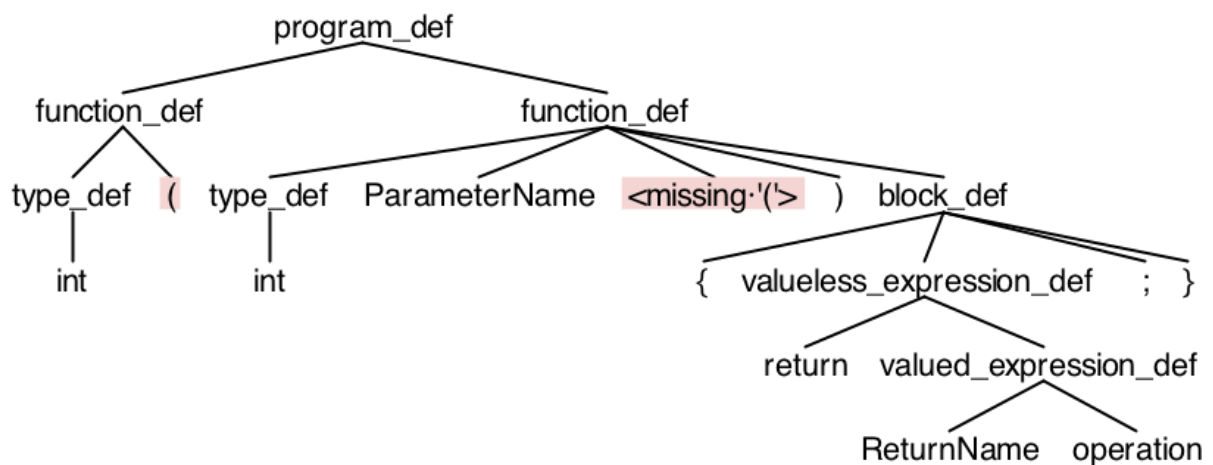
void main (int argc, char **argv) {}
int FunctionName2 () {}

```



### ○ Função sem identificador

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



### ○ For sem condição de parada

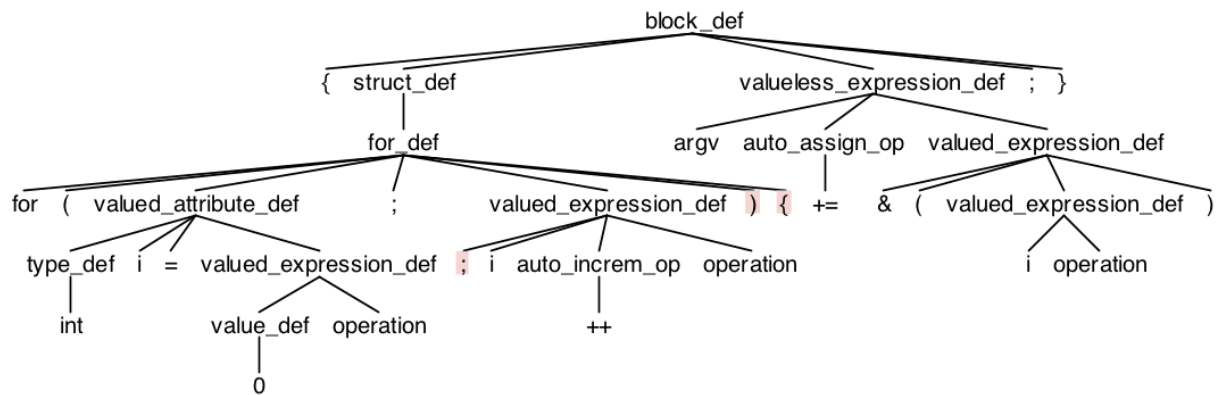
```

}

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

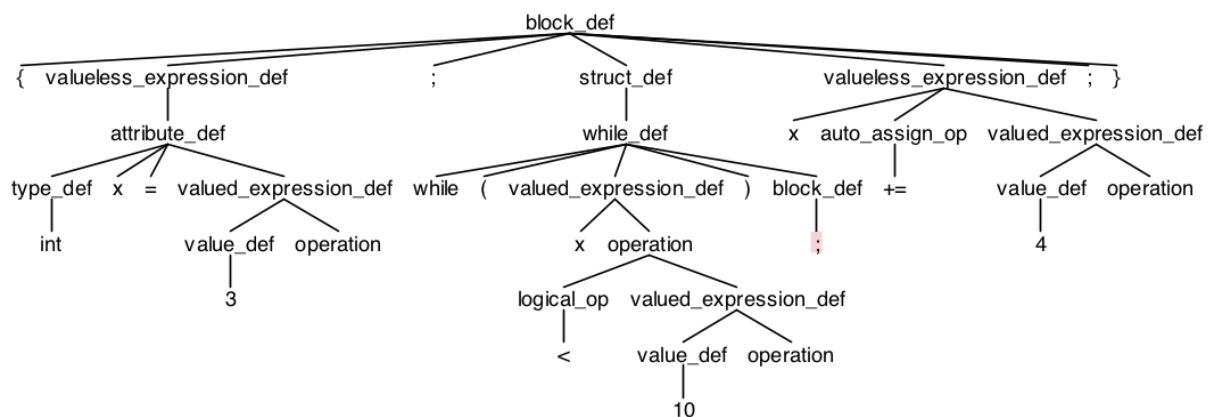
}

```



- **While sem escopo**

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

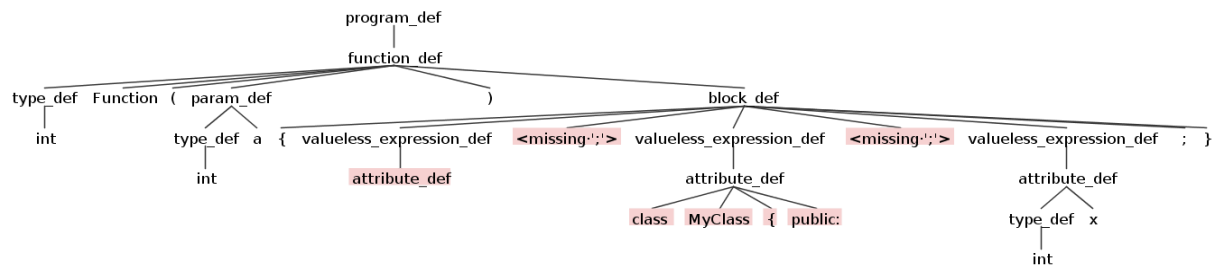


- **Definição de classe dentro de uma função**

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

double Function2() {
    return Pi;
}
```

}



## 9. Referências

1. <http://wwwantlr.org/>
2. <https://github.com/antlr/antlr4/tree/master/runtime/Cpp>
3. <https://llvm.org/docs/CommandGuide/lli.html>
4. <https://www.ibm.com/developerworks/library/os-createcompilerllvm1/index.html>
5. <https://github.com/annavicc/INE5426/blob/master/CodeGeneration/llvm/hello.ll>
6. <https://stackoverflow.com/questions/31092531/llvm-ir-printing-a-number>