

UNIVERSIDADE DE COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

COMPILADORES 2013/2014

---

# Compilador para linguagem iJava

---

*Autor:*

David CARDOSO

Número: 2011164039

*Autor:*

Bruno CACEIRO

Número: 2008107991

2 de Junho de 2014

# Índice

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise Lexical</b>	<b>3</b>
2.1	Tokens . . . . .	3
2.1.1	Comentários . . . . .	5
2.1.2	Tratamento de Erros . . . . .	5
<b>3</b>	<b>Análise Sintática e Semântica</b>	<b>6</b>
3.1	Gramática . . . . .	7
3.2	Gramática Inicial . . . . .	7
3.2.1	Gramática Final . . . . .	8
3.3	Árvore de Sintaxe Abstrata . . . . .	12
3.3.1	Estruturas . . . . .	12
3.3.2	Criação da Árvore . . . . .	12
3.3.3	Exemplo . . . . .	13
3.4	Análise Semântica . . . . .	14
3.5	Tabela de Símbolos . . . . .	15
3.5.1	Estruturas . . . . .	15
3.6	Tratamento de Erros Semânticos . . . . .	16
<b>4</b>	<b>Geração de Código</b>	<b>17</b>
<b>5</b>	<b>Conclusão</b>	<b>17</b>

# 1 Introdução

Este projecto consiste no desenvolvimento de um compilador para a linguagem *iJava* (imperative Java), que consiste num pequeno subconjunto da linguagem Java (versão 5.0). Os programas da linguagem *iJava* são constituídos por uma única classe (a principal), contendo necessariamente um método *main*, e podendo conter outros métodos e atributos, todos eles estáticos e (possivelmente) públicos.

O projecto foi estruturado em 3 fases, primeiramente foi feita a Análise Lexical, implementada na linguagem *C* e utilizando a ferramenta *lex*. A segunda fase consistiu na análise sintática, recorrendo ao *yacc/bison*, com a construção da árvore de sintaxe abstrata e análise semântica (tabelas de símbolos, deteção de erros semânticos). No final foi feita a geração de código, em *LLVM*.

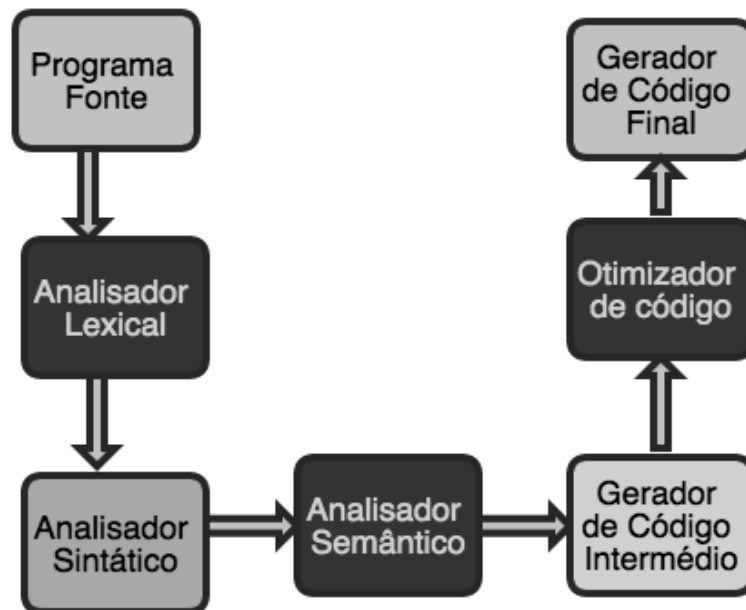


Figura 1: Fases de Compilação

## 2 Análise Lexical

A Análise Lexical consiste em analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos (*tokens*) que podem ser manipulados mais facilmente por um *parser*. Assim, é uma forma de verificar um determinado alfabeto, neste caso o alfabeto da linguagem *iJava*. Esta análise pode ser dividida em três fases:

- Extração e classificação de *tokens*;
- Eliminação de delimitadores e comentários;
- Tratamento de erros;

### 2.1 Tokens

- **ID:** Sequências alfanuméricas começadas por uma letra, onde os símbolos "\_" e "\$" contam como letras. Maiúsculas e minúsculas são consideradas ID's diferentes (*case sensitive*).
  - **Expressão Regular:**  $[a - zA - Z\_\$]([a - zA - Z\_\$0 - 9])^*$
- **INTLIT:** Sequências de dígitos decimais e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x"
  - **Expressão Regular:**  $(([0 - 9]) + |("0x"[0 - 9a-fA - F])^+)$
- **BOOLLIT:** "true" | "false"
- **INT:** "int"
- **BOOL:** "boolean"
- **NEW:** "new"
- **IF:** "if"
- **ELSE:** "else"
- **WHILE:** "while"
- **PRINT:** "System.out.println"
- **PARSEINT:** "Integer.parseInt"
- **CLASS:** "class"
- **PUBLIC:** "public"
- **STATIC:** "static"
- **VOID:** "void"

- **STRING:** "String"
- **DOTLENGTH:** ".length"
- **RETURN:** "return"
- **OCURV:** "("
- **CCURV:** ")"
- **OBRACE:** "{"
- **CBRACE:** "}"
- **OSQUARE:** "["
- **CSQUARE:** "]"
- **ASSIGN:** "="
- **SEMIC:** ";"
- **COMMA:** ","

Para a Análise Sintática foi necessário separar alguns *tokens* devido às diferentes prioridades que cada operador tem. Assim, os *tokens* foram agrupados pela ordem de precedência de operações.

- **OP1:** "&&"
- **OP1OR:** "|"
- **OP2:** "<" | ">" | "<=" | ">="
- **OP2EQS:** "==" | "!="
- **OP3:** "+" | "-"
- **OP4:** "\*" | "/" | "%" | "
- **NOT:** "!"

O *iJava* é um subconjunto da linguagem *Java*, como tal, existe um conjunto de funcionalidades que embora não sejam suportadas, têm de ser consideradas. Assim, foi necessário tratar todo um conjunto de palavras reservadas de forma a permitir que sejam lexicalmente válidas mas não sintaticamente.

- **RESERVED:**

- abstract | assert | break | byte | case | catch | char | const | continue | default | do | double | enum | extends | final | finally | float | for | goto | implements | import | instanceof | interface | long | native | package | private | protected | short | strictfp | super | switch | synchronized | this | throw | throws | transient | try | volatile | null | ++ | --

### 2.1.1 Comentários

Existem duas maneiras de fazer comentários:

- Comentar apenas uma linha `//<código>`.  
Caso seja detectado (`//`) todo o código que se segue é ignorado até encontrar uma mudança de linha.

**Expressão Regular:** `//".*`

- Comentar um bloco de código `/*<código>*/`

```
1  < COMMENT > <<EOF>> {BEGIN 0}
2  < COMMENT > "*/"      {BEGIN 0}
3  < COMMENT > "\n"      {}
4  < COMMENT > .          {}
5  "/*"                  {BEGIN COMMENT}
6
```

Listing 1: Detecção de Comentários

Caso seja detectado (`/*`) todo o código é ignorado até que seja encontrado o seu correspondente (`*/`). Caso seja detectado o EOF então é apresentada uma mensagem de erro: `"Line %d, col %d: unterminated comment"`. Foi criado um estado adicional no *lex* para poder tratar esta situação. Foi a criação deste estado que permitiu ignorar todos os *tokens* presentes dentro do comentário.

### 2.1.2 Tratamento de Erros

Se forem detectados erros lexicais no ficheiro de entrada então é impressa uma mensagem de erro no *stdout*:

- `"Line<num linha>,col<num coluna>:illegal character('<c >'\n)"`
- `"Line<num linha>,col<num coluna>:unterminated comment\n"`

Para podermos imprimir as mensagens de erro com o número da linha e coluna criámos uma variável *column* para poder contar as colunas e utilizamos a variável *yylineno*, disponibilizada pelo *yacc*, para sabermos o número das linhas. Para a contagem de colunas aumentamos a variável referente às colunas consoante o tamanho de cada *token*. Quando há uma mudança de linha, inicializa-se o contador das colunas a 1.

Para tratar o caso dos comentários criámos duas variáveis adicionais (*Commentline*, *Commentcolumn*) para poder guardar a coluna e linha onde o comentário é inicializado.

### 3 Análise Sintática e Semântica

De forma a realizar a análise sintática foi utilizada a ferramenta *lex*, para reconhecer e isolar os *tokens*, sendo que de seguida serão enviados para o *yacc* que irá ser responsável por verificar se estes pertencem à gramática da linguagem.

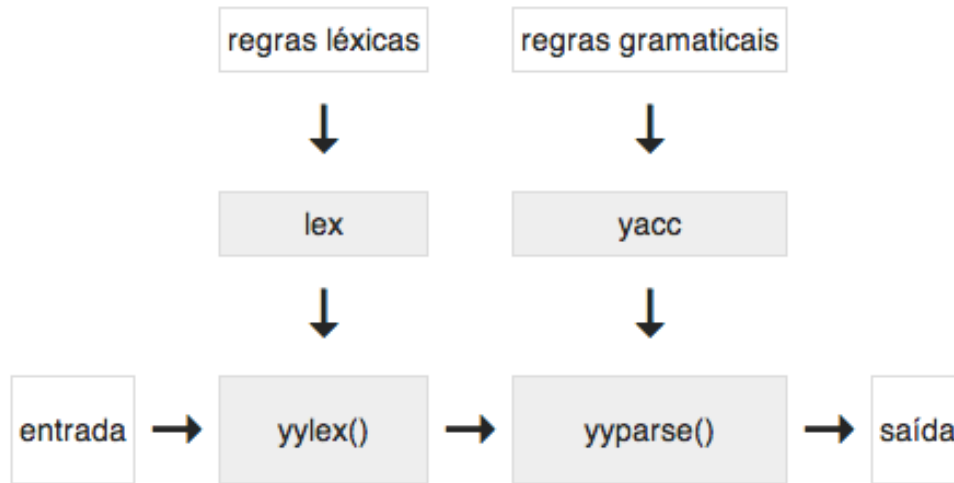


Figura 2: Relacionamento entre *lex* e *yacc*, retirado de <http://pt.wikipedia.org/wiki/Yacc>

Para realizar a ligação entre o *yacc* e o *lex* foram definidos *tokens* no *yacc*, posteriormente importados pelo *lex*. Sempre que o *lex* detecta uma sequência de caracteres correspondente a um *token* válido retorna um código acordado entre o *yacc* e *lex* de forma a identificar de forma única esse *token* (*enums*), sendo este valor enviado para o *yacc*. Caso o *token* corresponda a um tipo passível de ser processado (*INTLIT*, *BOOLIT*, *ID* e *operadores*), é guardado na variável *yyvar* o valor introduzido pelo utilizador.

Sendo a variável *yyval* também responsável por definir o tipo de retorno de cada regra da gramática, foi necessário acrescentar outros tipos de dados, de forma a permitir a criação da árvore de sintaxe abstrata. Assim, foi criada uma *union*, definida no *yacc*, que permite partilhar, no mesmo espaço de memória, vários tipos diferentes.

```
1 %union{
2     struct _Node* node;
3     char* token;
4     struct _idList* listId;
5     int type;
6 }
```

Listing 2: Union

Para permitir a percepção, pelo *yacc*, da linha e coluna que estão atualmente a serem utilizadas, foram também declaradas várias variáveis externas, partilhadas pelo *lex*.

```
1 extern int column;
2 extern int yylineno;
3 extern char* yytext;
4 extern int yyleng;
```

### 3.1 Gramática

A gramática é a maneira formal de especificar a sintaxe de uma linguagem. Desenvolver uma gramática não ambígua é um dos passos mais importantes para o sucesso de um compilador. Para a gramática da linguagem *iJava* usamos a notação **BNF** (*Backus Naur Form*), visto que a mesma é utilizada pelo *yaac* e permite remover as ambiguidades.

### 3.2 Gramática Inicial

```
Start → Program
Program → CLASS ID OBRACE FieldDecl | MethodDecl CBRACE
FieldDecl → STATIC VarDecl
MethodDecl → PUBLIC STATIC ( Type | VOID ) ID OCURV
           [ FormalParams ] CCURV OBRACE VarDecl Statement CBRACE
FormalParams → Type ID COMMA Type ID
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID COMMA ID SEMIC
Type → ( INT | BOOL ) [ OSQUARE CSQUARE ]
Statement → OBRACE Statement CBRACE
Statement → IF OCURV Expr CCURV Statement [ ELSE Statement ]
Statement → WHILE OCURV Expr CCURV Statement
Statement → PRINT OCURV Expr CCURV SEMIC
Statement → ID [ OSQUARE Expr CSQUARE ] ASSIGN Expr SEMIC
Statement → RETURN [ Expr ] SEMIC
Expr → Expr ( OP1 | OP2 | OP3 | OP4 ) Expr
Expr → Expr OSQUARE Expr CSQUARE
Expr → ID | INTLIT | BOOLLIT
Expr → NEW ( INT | BOOL ) OSQUARE Expr CSQUARE
Expr → OCURV Expr CCURV
Expr → Expr DOTLENGTH | ( OP3 | NOT ) Expr
Expr → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → ID OCURV [ Args ] CCURV
Args → Expr COMMA Expr
```

Lembramos que, em notação **ENBF**, os símbolos [...] englobam *tokens* opcionais e {...} implicam a repetição dos *tokens* 0 ou mais vezes.



### 3.2.1 Gramática Final

A gramática que nos foi dada era ambígua e por isso tivemos de efetuar diversas alterações para permitir a análise sintática ascendente com o *yacc*.

Algumas das alterações que efetuámos foram:

- Criação de estados adicionais para as regras que implicam a repetição de *tokens*.
- Criação de estados adicionais para as regras que continham *tokens* opcionais
- Estabelecimento de regras de prioridade de forma a gerir regras de precedência entre operadores
- Definição das regras de associação dos operadores ( à esquerda/direita)

```
1 %nonassoc IFX
2 %nonassoc ELSE
3
4 %left OP1OR
5 %left OP1
6 %left OP2EQS
7 %left OP2
8 %left OP3
9 %left OP4
10 %right NOT
11 %left OSQUARE DOTLENGTH
```

Listing 4: Associação de Operadores

Deveríamos ter usado recursividade à esquerda pois assim teríamos em memória apenas os elementos que estaríamos a analisar visto que estamos a efetuar reduções à medida que estamos a ler o *input*.

Foram ainda necessárias realizar alterações na gramática, de forma a impedir a indexação de *arrays* ( $a[1][2]$ ). Para tal as expressões foram divididas em dois tipos, expressões indexáveis e não indexáveis.

A gramática final, utilizada no *yacc* é a seguinte apresentada.

```
Start :
        CLASS ID OBRACE field_or_method_declaration CBRACE;

field_or_method_declaration :
        FieldDecl field_or_method_declaration
        | MethodDecl field_or_method_declaration
        ;

FieldDecl :
        STATIC VarDecl VarDecl_REPETITION;

MethodDecl :
```

```

        PUBLIC STATIC method_type_declaration ID OCURV FormalParams OCURV
        OBRACE VarDecl_REPETITION statement_declaration_REPETITION CBRACE;

method_type_declaration :
    Type
    | VOID
    ;

FormalParams :
    Type ID several_FormalParams
    | STRING OSQUARE CSQUARE ID
    ;

several_FormalParams :
    COMMA Type ID several_FormalParams
    ;

VarDecl_REPETITION :
    VarDecl VarDecl_REPETITION
    ;

VarDecl :
    Type ID several_var_decl_in_same_instructionOPTIONAL SEMIC;

several_var_decl_in_same_instructionOPTIONAL :
    COMMA ID several_var_decl_in_same_instructionOPTIONAL
    ;

Type :
    INT OSQUARE CSQUARE
    | BOOL OSQUARE CSQUARE
    | INT
    | BOOL
    ;

statement_declaration_REPETITION :
    Statement statement_declaration_REPETITION
    ;

Statement :
    OBRACE several_statement CBRACE
    | IF OCURV Expr CCURV Statement %prec IFX
    | IF OCURV Expr CCURV Statement ELSE Statement
    | WHILE OCURV Expr CCURV Statement
    | PRINT OCURV Expr CCURV SEMIC
    | ID array_indexOPTIONAL ASSIGN Expr SEMIC
    | RETURN return_expression SEMIC
    ;

```

```

several_statement :
    Statement several_statement
    |
    ;

array_indexOPTIONAL :
    OSQUARE Expr CSQUARE
    |
    ;

return_expression :
    Expr
    |
    ;

IndexableExpr :
    ID
    |
    INTLIT
    |
    BOOLLIT
    |
    ID OCURV Args_OPTIONAL CCURV
    |
    OCURV Expr CCURV
    |
    Expr DOTLENGTH
    |
    IndexableExpr OSQUARE Expr CSQUARE
    |
    PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
    ;

Expr :
    Expr OP1 Expr %prec OP1
    |
    Expr OP1OR Expr %prec OP1OR
    |
    Expr OP4 Expr %prec OP4
    |
    Expr OP3 Expr %prec OP3
    |
    Expr OP2 Expr %prec OP2
    |
    Expr OP2EQS Expr %prec OP2EQS
    |
    OP3 Expr %prec NOT
    |
    NOT Expr %prec NOT
    |
    NEW INT OSQUARE Expr CSQUARE
    |
    NEW BOOL OSQUARE Expr CSQUARE
    |
    IndexableExpr
    ;

Args_OPTIONAL :
    Args
    |
    ;

Args :
    Expr comma_expr ;

comma_expr :
    COMMA Expr comma_expr
    |

```

;

Listing 5: Gramática Final

## 3.3 Árvore de Sintaxe Abstrata

### 3.3.1 Estruturas

Para a árvore de sintaxe abstrata optámos por utilizar um nó genérico (*Node*) com a seguinte estrutura:

```
1 /* General Node */
2 typedef struct _Node
3 {
4     //Type of the Node (to identify the type of the node)
5     NodeType n_type;
6
7     //Type of the Struct (Int, Void, String,...)
8     Type type;
9
10    //Id or list of id's
11    listID* id;
12
13    //The tree next nodes (the case of if)
14    struct _Node* n1;
15    struct _Node* n2;
16    struct _Node* n3;
17
18    //Next node
19    struct _Node* next;
20
21    //Literals (to store the values)
22    char* value;
23
24    char isStatic;
25 }Node;
26
27
28 /* Linked list of ID's (for multiple declaration of variables) */
29 typedef struct _idList
30 {
31     char* id;
32     struct _idList* next;
33 }listID;
```

Listing 6: Estruturas para representação de Nós da Árvore

### 3.3.2 Criação da Árvore

```
1 listID* insertID(Node* currentNode, char* id);
2 listID* newVarID(char* id, listID* next);
3 NodeType getOperatorType(char* op);
4 Node* createNull();
5 Node* insertClass(char* id, Node* statements);
6 Node* newVarDecl(int type, char* id, listID* moreIds, Node* next);
7 Node* setNext(Node* current, Node* next);
8 Node* setStatic(Node* currentNode);
```

```

9 Node* newMethod(int type, char* id, Node* params, Node* varDecl, Node*
    statements);
10 Node* insertCompound(Node* expression);
11 Node* insertIf(Node* expression, Node* statement1, Node* statement2);
12 Node* insertPrint(Node* expression);
13 Node* insertWhile(Node* expression, Node* statements);
14 Node* insertReturn(Node* expression);
15 Node* insertStore(char* id, Node* arrayIndex, Node* expression);
16 Node* createTerminalNode(int n_type, char* token);
17 Node* newParamDecl(int type, char* id, listID* moreIds, Node* next);
18 Node* insertDotLength(Node* expression);
19 Node* insertLoadArray(Node* expression, Node* indexExpression);
20 Node* insertParseInt(char* id, Node* indexExpression);
21 Node* insertNewArray(int type, Node* expression);
22 Node* createCall(char* id, Node* *args);
23 Node* insertExpression(char* op, Node* exp);
24 Node* insertDoubleExpression(Node* exp1, char* op, Node* exp2);
25

```

Listing 7: Funções para a criação da árvore

### 3.3.3 Exemplo

Considerando o seguinte programa:

```

1 class gcd {
2     public static void main(String[] args) {
3         int x;
4         return;
5     }
6 }

```

Listing 8: Programa Exemplo

A árvore gerada é a seguinte:

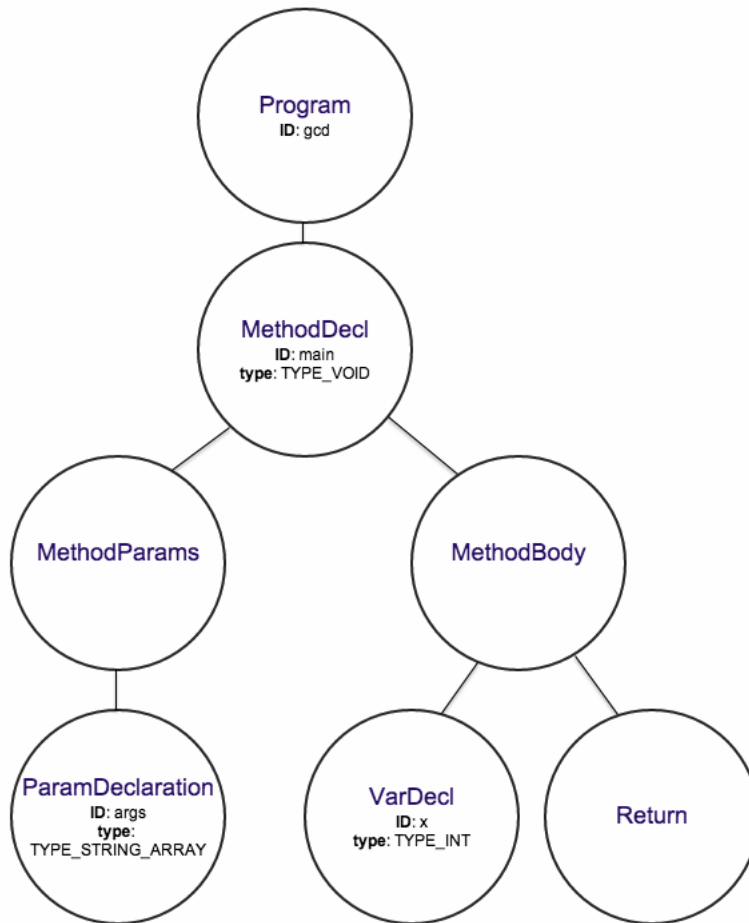


Figura 3: Árvore de Sintaxe Abstrata para o Programa Exemplo

### 3.4 Análise Semântica

A Análise Semântica tem como principais objectivos a ligação das definições de variáveis com a sua utilização, a verificação da correcção de tipos, declarações e chamadas de funções. Esta análise é dividida em duas fases:

1. Para cada *scope* no programa:
  - Processar as declarações:
    - Adicionar novas entradas na tabela de símbolos
    - Apresentar mensagem de erro caso haja variáveis repetidas
  - Processar os *statements*
    - Procurar variáveis que não foram declaradas neste *scope* ou no *scope* global, e caso não tenham sido, apresentar mensagem de erro
2. Processar todos os *statements* do programa outra vez
  - Usar a tabela de símbolos para determinar o tipo de cada expressão e procurar erros de tipo (atribuições, cálculos, contagem e tipos de argumentos correctos,...)

## 3.5 Tabela de Símbolos

A Tabela de símbolos é criada a partir da Árvore de Sintaxe Abstrata, onde se percorrem todos os nós relativos às declarações de métodos e atributos, sendo criada uma nova entrada na tabela para cada um deles.

### 3.5.1 Estruturas

Para representação da tabela de símbolos, foram utilizadas as seguintes estruturas:

```
1  /* Table Node */
2  typedef struct _TableNode
3  {
4      //Type of the Node (to identify the type of the node)
5      TableType n_type;
6
7      //Type of the Struct (Int, Void, String,...)
8      Type type;
9
10     //ID or list of id's
11     listID* id;
12
13     //Next node
14     struct _TableNode* next;
15
16     //If is a param
17     char isParam;
18 }TableNode;
19
20 /* Table */
21 typedef struct _Table{
22     TableNode* table;
23     struct _Table* next;
24 }Table;
```

Listing 9: Estruturas para representação da Tabela de Símbolos



## 3.6 Tratamento de Erros Semânticos

Para o tratamento de erros semânticos utilizamos as seguintes funções:

```
1  /* Check if global variables have the same name as the Methods:
2     - If they have print error message: "Symbol %s already defined" and exit
3  */
4  void    checkIfExists(char* id, Table* local);
5  void    checkSemanticErrors(Node* ast, Table* local, Table* main);
6  void    checkErrors(Node* ast, Table* symbols, Table* main);
7
8  /* Check if ID exists:
9     - If exists return the Type
10    - If doesn't exist, print error message: "Cannot find symbol %s\n" and exit
11  */
12 int      checkifIDExists(char* id, TableType type, Table* table, Table* main);
13 Table*   getMethodTable(Table* main, char* methodID);
14
15 /* Check if the Literal is valid (decimal / hexadecimal / octal)
16    - If it is invalid print error message: "Invalid literal %s\n" and exit
17  */
18 void     validIntLit(char* lit);
19 void     checkTypes(Node* ast, Table* main);
20 void     operatorError2Types(int op, int n1, int n2);
21 void     operatorError1Types(int op, int n1);
22 void     assignmentError(char* var, int n1, int n2);
23 void     assignmentErrorArray(char* var, int n1, int n2);
24 void     setTable(Table* oi);
25 int      getFunctionType();
26 void     statementError(int op, int n1, int n2);
27 void     statementError1oranother(int op, int n1, int n2, int n3);
28 char*    getFunctionName();
29 void     getErrorCall(int i, char* name, int n1, int n2);
```

Listing 10: Funções para tratamento de erros semânticos

**4    Geração de Código**

**5    Conclusão**