Implementação de um compilador e ambiente de desenvolvimento para a linguagem MicroJava

Fernando Henrique da Silva

Gabriel Nobrega de Lima

Contents

0 Introdução

Este documento prevê a orientação e documentação do desenvolvimento de um compilador para a linguagem de programação MicroJava. Como esta tarefa está dirigida por meio de fases de compilação iremos abordar cada uma das etapas de desenvolvimento ao passo que estas forem sendo implementadas, tais fases são compreendidas pela Análise Léxica, Análise Sintática, Análise Semântica e Geração de código. Em cada uma das fases serão comentados todos os recursos que auxiliaram sua construção, a maneira pela qual a estrutura de dados foi construída, a execução de testes e seus resultados.

1.0 Análise Léxica

A análise léxica compreende a primeira fase no processo de compilação que recebe um fluxo de caracteres como entrada, denominado código fonte. Sua função é qualificar as cadeias de caracteres em *tokens* válidos de acordo com as especificações determinadas na linguagem MicroJava, orientando todo o processo de análise realizado em suas diferentes fases posteriores.

1.1 Definições da linguagem MicroJava para análise léxica

Definição dos símbolos que pertencem à gramática:

letter *= 'a'..'z' | 'A'..'Z'.*
digit *= '0'..'9'.*
whiteSpace *= ' ' | '\t' | '\r' |'\n'.*

Definição das cadeias pertencentes à linguagem:

ident *= letter {letter |digit}.*
number *= digit {digit}.*
charConst *= "'" char "'". // including '\r', '\t', '\n'*

Definição das palavras reservadas:

reserved = (*program, class,if, else, while, read, print, return,void, final, new)*

Definição dos operadores:

operators *= (+, -, *, /, %,==, !=, >, >=, <, <=, (,), [,], {, }, =, ;,* ***, , .)***

Especificação dos comentários:

// Esta cadeia deve ser ignorada no processo de compilação

1.2 Definição dos AFDs para o desenvolvimento do Analisador Léxico

A implementação do Analisador Léxico foi orientado através dos Autômatos Finitos Determinísticos(AFDs) definidos nas seções seguintes.

1.2.1 Identificadores e palavras chaves

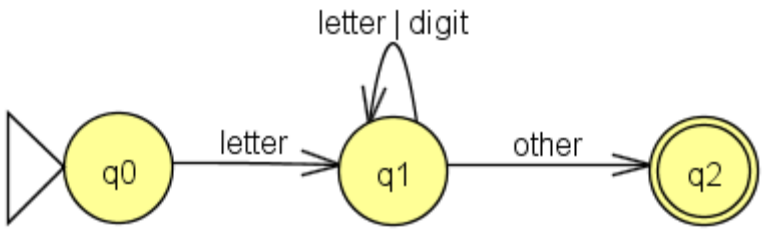

Figura 1 – Autômato para obtenção de identificadores e palavras chaves.

Nossa implementação utiliza o mesmo autômato para o reconhecimento dos identificadores e palavras chaves. A principio qualquer cadeia que satisfaça o autômato é um identificador, contudo, no estado final realizamos uma varredura verificando se a cadeia obtida está contida na tabela de palavras reservadas, se positivo identificamos a cadeia com o *token* reservado apropriado, caso contrário ele continuará sendo tratado como um identificador. Esta decisão de projeto permite que apenas um autômato defina todas as palavras reservadas e identificadores simultaneamente, o que simplifica a construção do Analisador Léxico.

1.2.2 Números

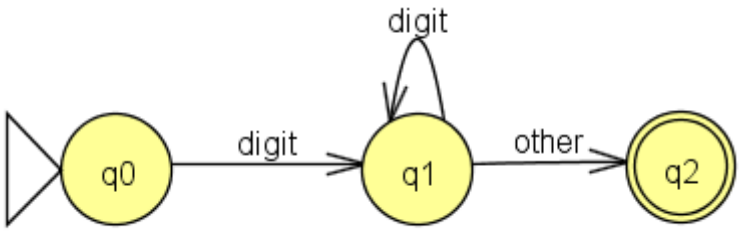

Figura 2 – Autômato para obtenção de números.

A especificação da linguagem define apenas números inteiros, desta maneira o autômato considera um número a cadeia que se inicia com um digito e termina com o surgimento de uma letra ou símbolo qualquer que não seja um digito.
Ressaltamos que a próxima cadeia deve ser iniciada com o ultimo caractere que promoveu a chegada ao estado final do autômato.

1.2.3 Operadores

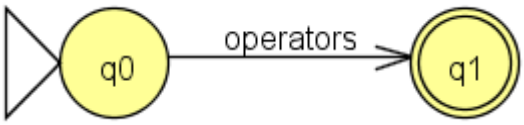

Figura 3 – AFD de generalização para os operadores.

O conjunto *operators* foi definido na seção de especificações da linguagem. Ressaltamos aqui que o autômato da Figura 3 expressa de maneira geral a lógica que deve ser implementada na Análise Léxica para os operadores, existem alguns casos na determinação de cadeias pertencentes ao conjunto *operators* que merecem maior detalhamento devido à presença de símbolos iniciais iguais. Um exemplo é a identificação das cadeias > e >= que exigem a obtenção de um segundo símbolo para a definição exata do *token*, estes casos que estão relacionados ao conjunto *operators* são abordados nos autômatos seguintes.

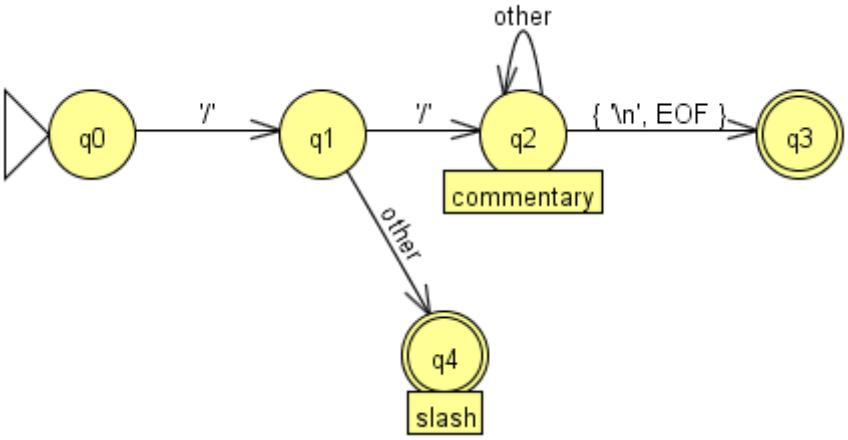

Figura 4 – AFD para comentários e operador divisão.

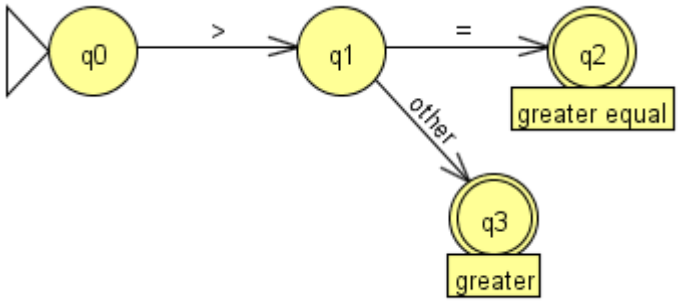

Figura 5 – AFD para maior-igual ou maior.

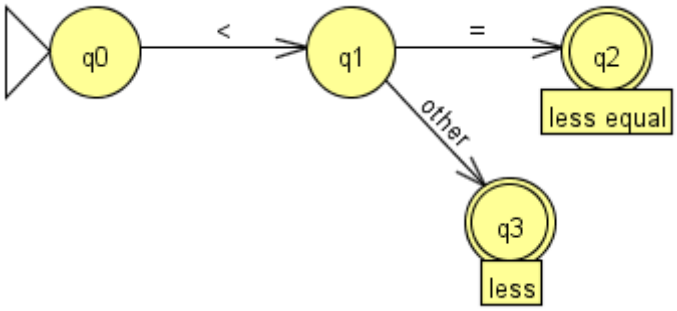

Figura 6 – AFD para menor-igual ou menor.

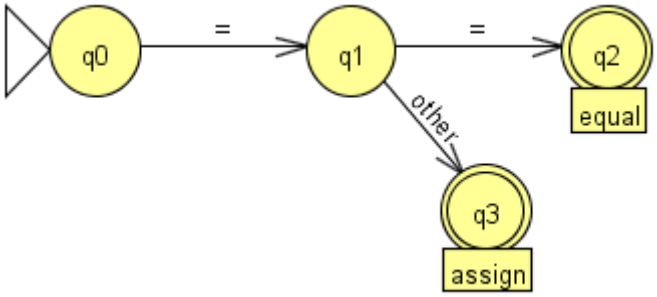

Figura 7 – AFD para atribuição e igualdade.

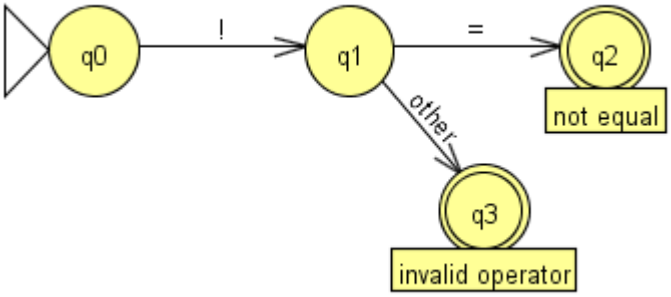

Figura 8 – AFD para negação e não igualdade.

Os demais casos para os operadores de linguagem seguem o autômato de generalização definido na Figura 3 por serem obtidos diretamente.

1.2.4 Charconst

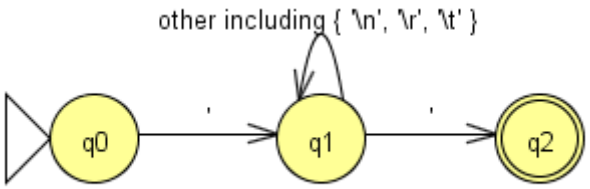

Figura 9 – AFD para as cadeias de caracteres.

O AFD da Figura 9 é apenas um esboço dos caracteres possíveis, sua maior relevância é mostrar em si que os símbolos especiais de pulo de linhas, retorno e tabulação devem estar disponíveis.

1.3 Detalhes de implementação

A implementação do analisador léxico está fortemente baseada em tabelas *Hash* o que facilita a identificação dos *tokens* e cadeias de forma direta e simples. Os *tokens* estão definidos sobre uma estrutura de enumeração, o que assegura uma codificação única e evitam problemas no decorrer do projeto:

```
public static enum Kind {
        UNKNOWN, EOF, IDENTIFIER, NUMBER, CHARACTER,
        // keywords
        CLASS, ELSE, FINAL, IF, NEW, PRINT, PROGRAM, READ,
        RETURN, VOID, WHILE,
        // operators
        PLUS, MINUS, TIMES, SLASH, REMAINDER, EQUAL, NOT_EQUAL, LESS,
        LESS_EQUAL, GREATER, GREATER_EQUAL, ASSIGN, SEMICOLON, COMMA,
        PERIOD, LEFT_PARENTHESIS, RIGHT_PARENTHESIS, LEFT_BRACKET,
        RIGHT_BRACKET, LEFT_BRACE, RIGHT_BRACE
    }
```

Com as definições relacionamos cada um dos *tokens* com suas respectivas cadeias representativas, criando a tabela de símbolos em uma estrutura de dados baseada em tabelas *Hash*. O fragmento de código a seguir exemplifica a estruturação descrita:

```
static final Map<Kind, String> Name;

Name.put(Kind.CLASS, "class");
Name.put(Kind.ELSE, "else");
Name.put(Kind.FINAL, "final");
Name.put(Kind.IF, "if");
Name.put(Kind.NEW, "new");
Name.put(Kind.PRINT, "print");
Name.put(Kind.PROGRAM, "program");
Name.put(Kind.READ, "read");
Name.put(Kind.RETURN, "return");
Name.put(Kind.VOID, "void");
Name.put(Kind.WHILE, "while");
            ...
```

A estrutura definida permite facilmente o intercabeamento de cadeias e tokens de forma direta e simples, o que facilita os casos em que o desenvolvedor possuí um *token* e gostaria de obter sua respectiva cadeia representativa, ou possua sua cadeia e queira obter seu *token* correspondente. O fato dos tipos de *token* estarem definidos sobre uma estrutura de enumeração permitem que problemas decorridos por falta de atenção no desenvolvimento sejam percebidos no momento de compilação, já que a linguagem Java define estruturas de enumeração como tipos bem definidos, fazendo com que passagens de inteiros quaisquer sobre argumentos do tipo *Kind* desencadeem erros na compilação.

1.4 Ambiente de testes e resultados

Todo o desenvolvimento da Análise Léxica foi auxiliado pela classe *TestScanner* que realiza baterias de testes automáticos sobre cada um dos métodos desenvolvidos, tornando o processo de construção mais rápido e preciso já que os erros são identificados rapidamente. A classe TestScanner possuí métodos que colocam sobre teste cada um dos procedimentos da Análise Léxica através de baterias de testes que são estrategicamente definidas visando testar todas as possibilidades de entradas possíveis, o que permite averiguar se o comportamento resultante é realmente o esperado. A classe de testes TestScanner pode ser visualizada no fragmento de código logo abaixo:

```
public class TestScanner {

    public static void main(String args[]) {
        if (args.length == 0) {
            executeTests();
        } else {
            for (int i = 0; i < args.length; ++i) {
                reportFile(args[i]);
            }
        }
    }

    private static void reportFile(String filename) {
        try {
            report("File: " + filename,
                    new InputStreamReader(new FileInputStream(filename)));
        } catch (IOException e) {
            System.out.println("-- cannot open file " + filename);
        }
    }

    private static void report(String header, Reader reader) {
        Scanner scanner = new Scanner(reader);
        Token token;
        System.out.println(header);
        while (true) {
            token = scanner.next();
```

```
            if (token.kind == Token.Kind.EOF) {
                break;
            }
            System.out.printf("%3d, %3d: %s ", token.line, token.column,
                    Token.getName(token.kind));
            switch (token.kind) {
                case IDENTIFIER:
                    System.out.println(token.string);
                    break;
                case NUMBER:
                    System.out.println(token.string + " = " +
token.intValue);
                    break;
                case CHARACTER:
                    System.out.println(token.string + " = " +
token.charValue);
                    break;
                case UNKNOWN:
                    System.out.println("- " + token.errorMessage
                            + ": " + token.string);
                    break;
                default:
                    System.out.println();
                    break;
            }
        }
        System.out.println();
    }

    private static void executeTests() {
        testNumbers();
        testCharacters();
        testIdentifiers();
        testKeywords();
        testOperators();
    }

    private static void testNumbers() {
        report("Test: numbers", new StringReader("1 255 001"));
    }

    private static void testCharacters() {
        report("Test: characters",
                new StringReader("'c' '1' '+' '@' ' ' '\\t' '\\r'
'\\n'"));
        report("Test: characters errors",
                new StringReader("''\n'\\\\'\n'12'"));
    }

    private static void testIdentifiers() {
        report("Test: identifiers", new StringReader("int char a xyz
world"));
    }

    private static void testKeywords() {
        report("Test: keywords",
                new StringReader("class else final if new print"
                + " program read return void while"));
    }

    private static void testOperators() {
```

```
        report("Test: operators",
                new StringReader("+ - * / % == != < <= > >= ="
                + " ; , .  ( ) [ ] { }"));
        report("Test: operators errors",
                new StringReader(" !# \n !!"));
    }
}
```

Cada um dos métodos testNumbers, testCharacters, testIdentifiers, testKeywords e testOperators submetem o analisador léxico a testes exaustivos sobre o reconhecimento de cada um dos tipos de cadeias possíveis, sendo que a inclusão e exclusão de testes se dá simplesmente pela adição ou remoção de cadeias passadas ao método *report*.

Para critério de demonstração todos os métodos definidos na Análise Léxica foram submetidos a uma dada bateria de testes obtendo os seguintes resultados:

```
Test: numbers
  1,   1: number 1 = 1
  1,   3: number 255 = 255
  1,   7: number 001 = 1

Test: characters
  1,   1: character 'c' = c
  1,   5: character '1' = 1
  1,   9: character '+' = +
  1,  13: character '@' = @
  1,  17: character ' ' =
  1,  21: character '\t' =
  1,  26: character '\r' =
  1,  31: character '\n' =


Test: characters errors
  1,   2: UNKOWN - empty character: ''
  2,   3: UNKOWN - invalid character: '\\'
  3,   3: UNKOWN - unclosed character: '12'

Test: identifiers
  1,   1: identifier int
  1,   5: identifier char
  1,  10: identifier a
  1,  12: identifier xyz
  1,  16: identifier world

Test: keywords
  1,   1: class
  1,   7: else
  1,  12: final
  1,  18: if
  1,  21: new
  1,  25: print
  1,  31: program
  1,  39: read
  1,  44: return
  1,  51: void
  1,  56: while
```

```
Test: operators
  1,   1: +
  1,   3: -
  1,   5: *
  1,   7: /
  1,   9: %
  1,  11: ==
  1,  14: !=
  1,  17: <
  1,  19: <=
  1,  22: >
  1,  24: >=
  1,  27: =
  1,  29: ;
  1,  31: ,
  1,  33: .
  1,  36: (
  1,  38: )
  1,  40: [
  1,  42: ]
  1,  44: {
  1,  46: }

Test: operators errors
  1,   2: UNKOWN - invalid operator: !#
  2,   2: UNKOWN - invalid operator: !!
```

Uma análise na bateria de testes permite visualizar que o analisador léxico se comportou bem sobre as entradas e identificou as cadeias em seus respectivos *tokens* de forma coerente. Foram realizados dezenas de outros testes e o sistema se comportou de forma estável, o que garante uma base sólida no desenvolvimento do compilador já que as demais fases são dependentes desta e uma implementação mal elaborada poderia acarretar em grande tempo despendido no processo de correção do software.

2.0 Análise sintática

O analisador sintático é uma fase do processo de compilação que verifica se as cadeias previamente identificadas por seus tokens correspondentes respeitam as definições de gramáticas da linguagem MicroJava. Esta etapa verifica somente se a ordem dos tokens nas cadeias pertence a linguagem, mas sua noção lógica não é analisada, sendo postergada para a próxima fase de compilação.

2.1 Definições da linguagem MicroJava

A linguagem MicroJava tem sua gramática definida na forma Normal Extendida de Backus-Naur(EBNF) como segue abaixo:

Program = *"program" ident {ConstDecl | VarDecl | ClassDecl} "{" {MethodDecl} "}".*

ConstDecl = *"final" Type ident "=" (number | charConst) ";".*

VarDecl = *Type ident {"," ident } ";".*

ClassDecl = "*class" ident "{" {VarDecl} "}".*

MethodDecl = *(Type | "void") ident "(" [FormPars] ")" {VarDecl} Block.*

FormPars = *Type ident {"," Type ident}.*

Type = *ident ["[" "]"].*

Block = *"{" {Statement} "}".*

Statement = *Designator ("=" Expr | ActPars) ";"*
| "if" "(" Condition ")" Statement
["else" Statement]
| "while" "(" Condition ")" Statement
| "return" [Expr] ";"
| "read" "(" Designator ")" ";"
| "print" "(" Expr ["," number] ")" ";"
| Block
| ";".

ActPars = *"(" [Expr {"," Expr}] ")".*

Condition = *Expr Relop Expr.*

Relop = *"==" | "!=" | ">" | ">=" | "<" | "<=".*

Expr = *["-"] Term {Addop Term}.*

Term = *Factor {Mulop Factor}.*

Factor = *Designator [ActPars]*
| number
| charConst
| "new" ident ["[" Expr "]"]
| "(" Expr ")".

Designator = *ident {"." ident | "[" Expr "]"}.*

Addop = *"+" | "-".*

Mulop = *"*" | "/" | "%".*

Esta gramática irá orientar todos o desenvolvimento do analisador sintático, já que ela nos permite visualizar todas as possibilidades estruturais que o MicroJava permite o programador utilizar.

2.2 Grafos sintáticos

A definição das linguagens normalmente é realizada utilizando a EBNF o que facilita a forma de expressá-la, no entanto, sua conversão para um grafo sintático simplifica a implementação de cada um dos procedimentos do analisador sintático. Visto que tal procedimento corrobora para a criação da fase de análise sintática produzimos para cada uma das regras um grafo sintático que nos auxiliou em seu desenvolvimento, estes podem ser vistos logo abaixo.

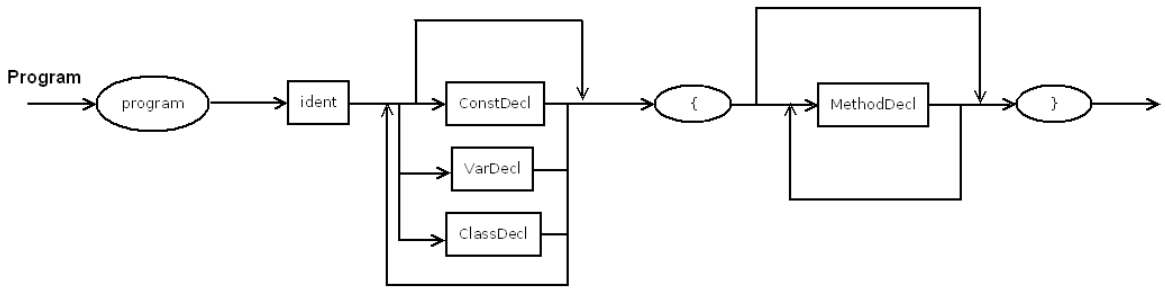

Figura 10 – Grafo sintático de Program.

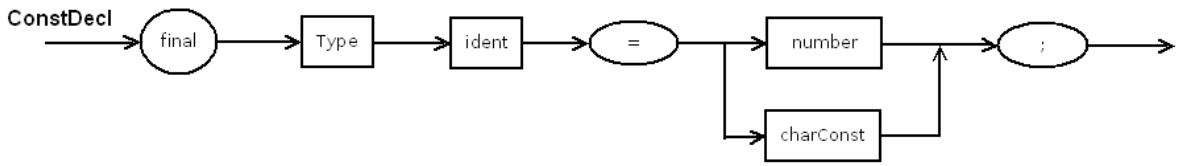

Figura 11 - Grafo sintático de ConstDecl.

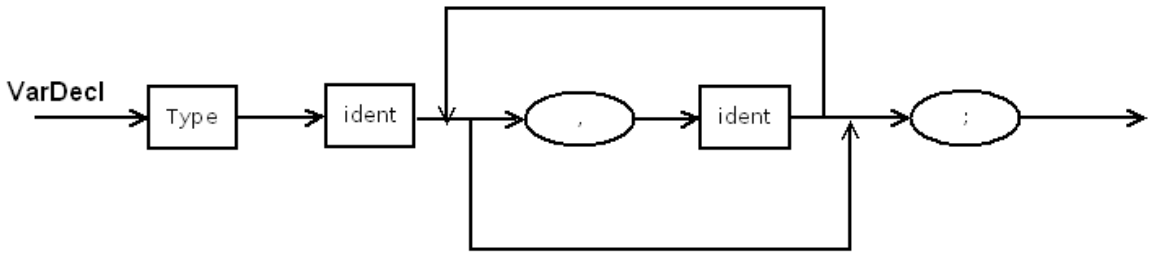

Figura 12 - Grafo sintático de VarDecl.

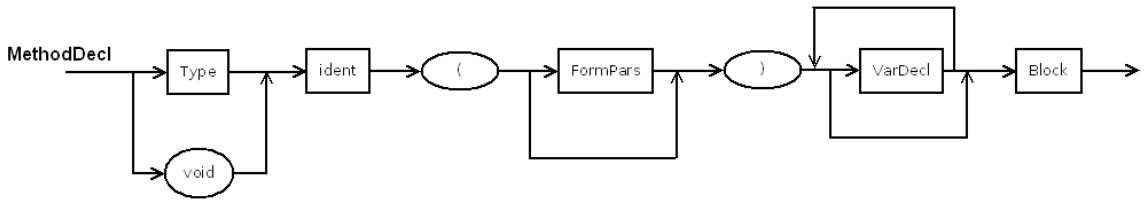

Figura 13 - Grafo sintático de MethodDecl.

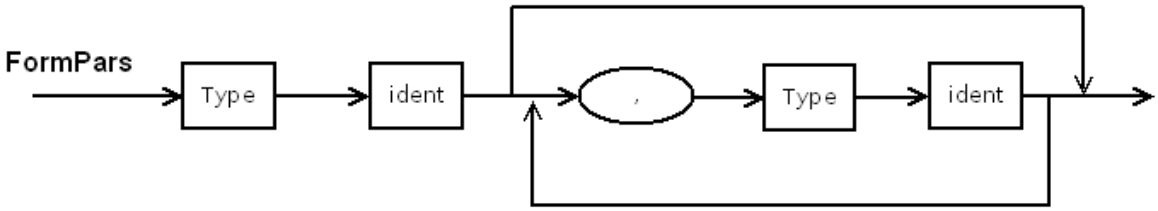

Figura 14 - Grafo sintático de FormPars.

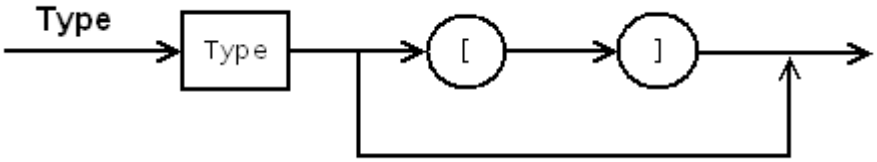

Figura 15 - Grafo sintático de Type.

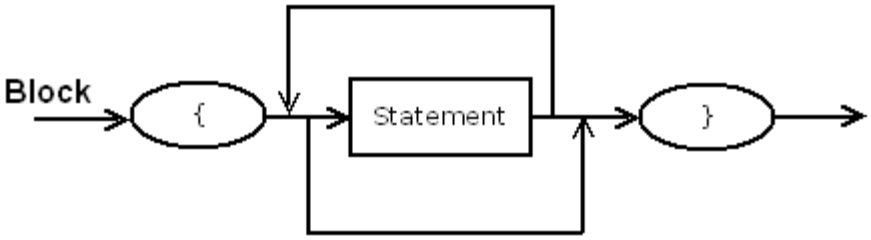

Figura 16 - Grafo sintático de Block.

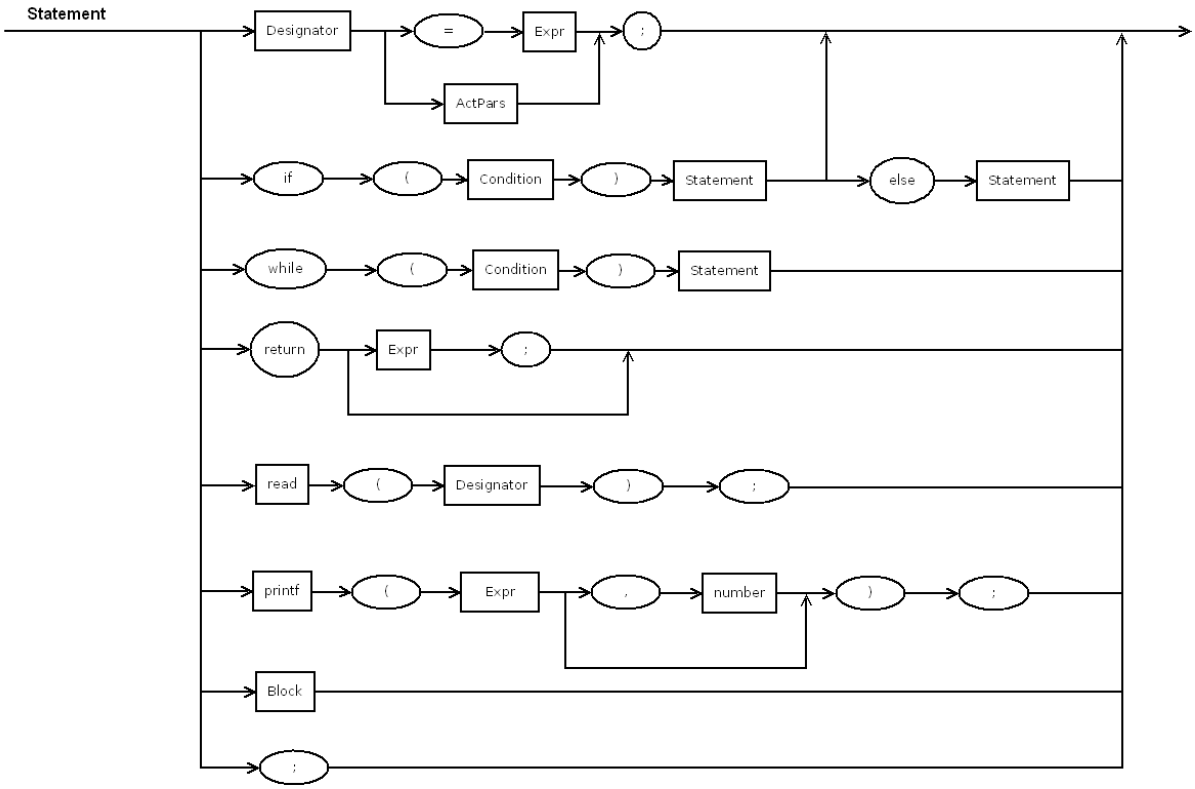

Figura 17 - Grafo sintático de Statement.

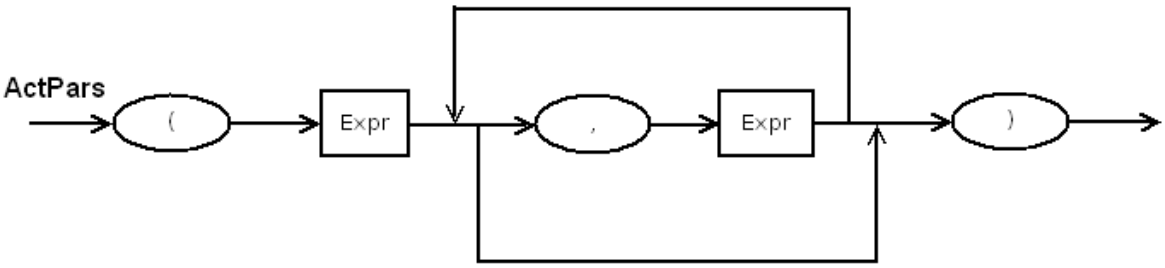

Figura 18 - Grafo sintático de ActPars.

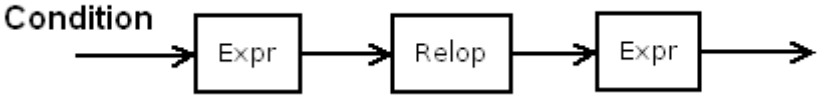

Figura 19 - Grafo sintático de Condition.

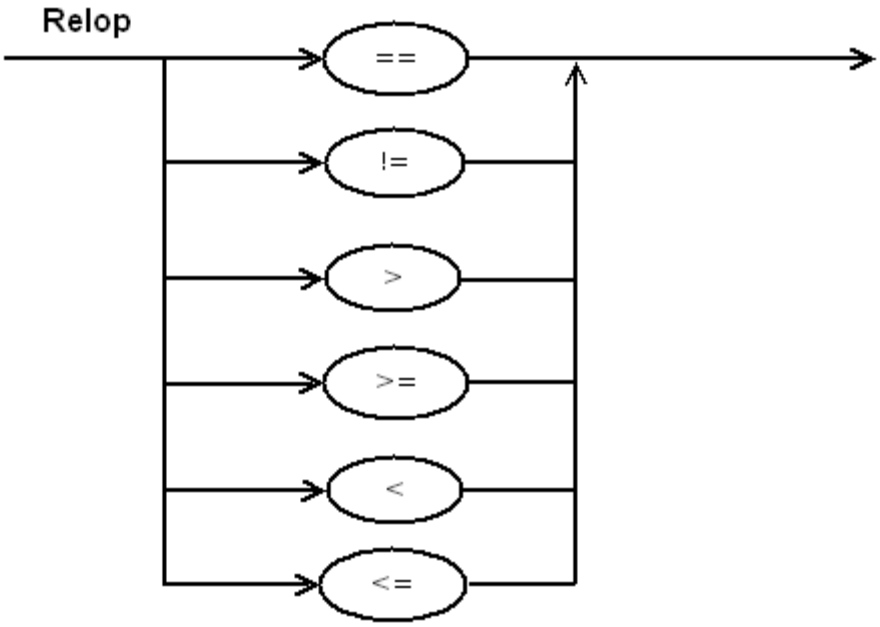

Figura 20 - Grafo sintático de Relop.

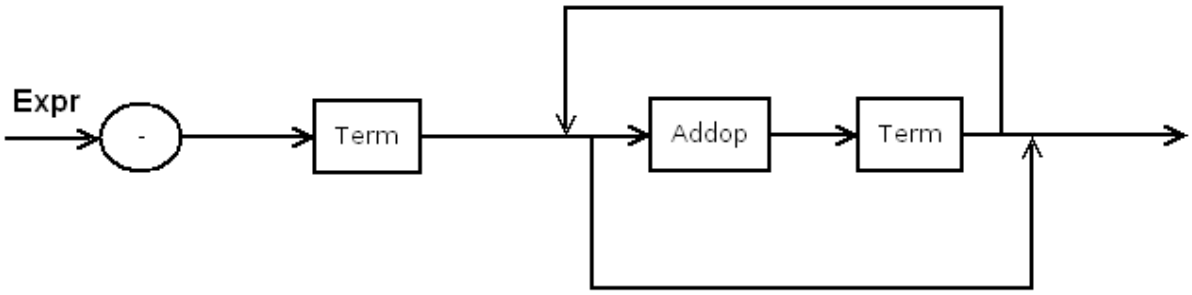

Figura 21 - Grafo sintático de Expr.

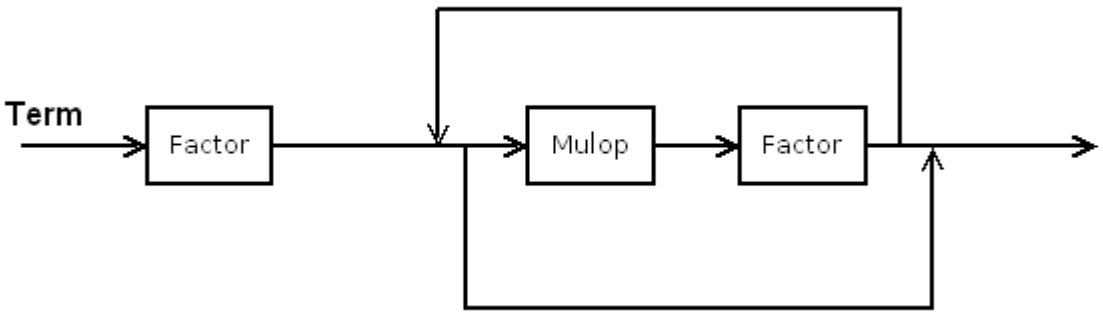

Figura 22 - Grafo sintático de Term.

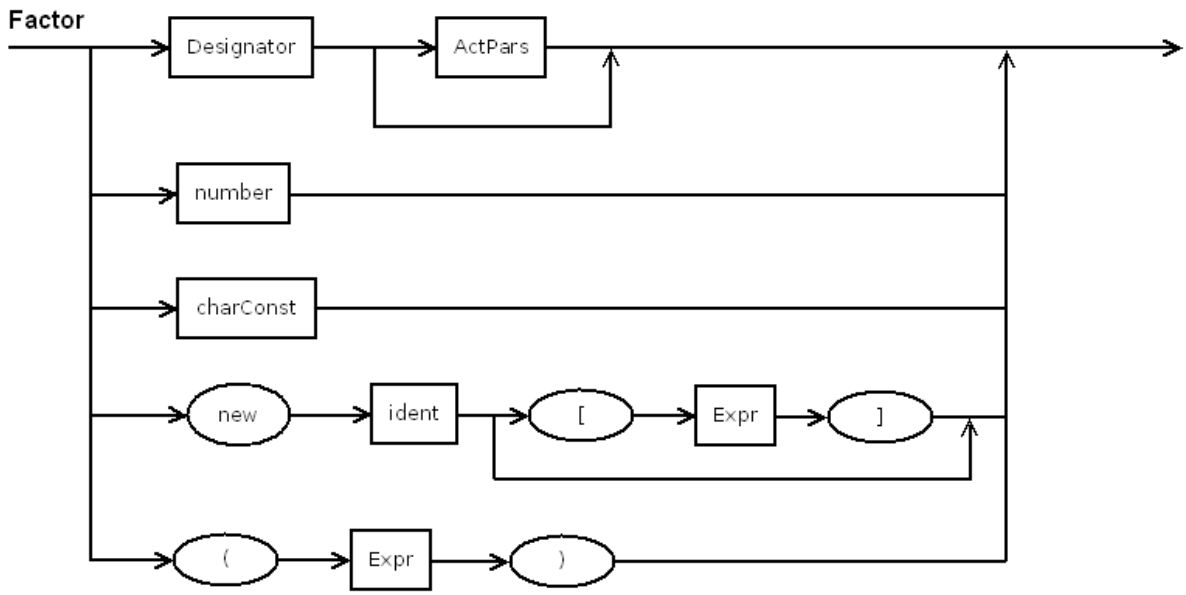

Figura 23 - Grafo sintático de Factor.

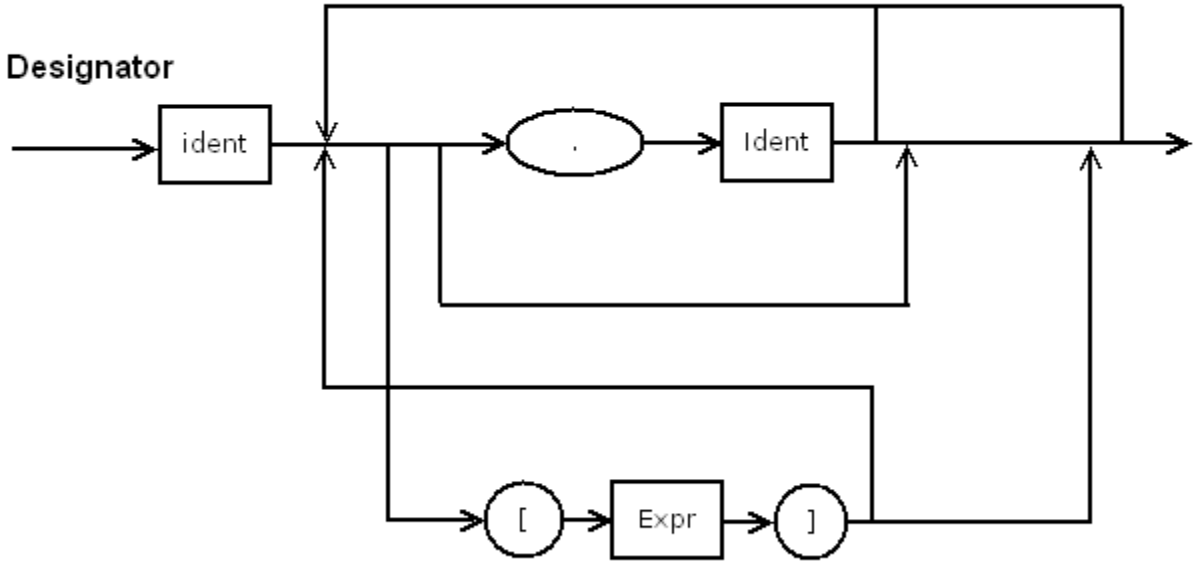

Figura 24 - Grafo sintático de Designator.

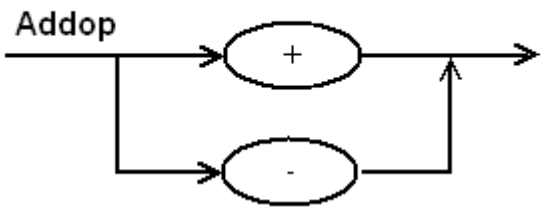

Figura 25 - Grafo sintático de Addop.

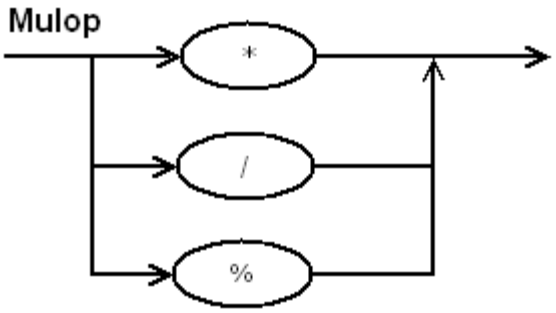

Figura 26 - Grafo sintático de Mulop.

2.3 Detalhes de implementação

Na implementação do analisador sintático foram realizadas algumas modificações estruturais no código base para que houvesse um maior sincronismo com a construção realizada previamente no analisador léxico. Os conjuntos *First* foram convertidos para serem baseados em conjuntos de tabelas Hash, de modo que sua utilização se tornasse mais amigavel e pouco sujeita a erros de programação. O fragmento de código abaixo indica uma pequena parte da construção da tabela Hash dos conjuntos First implementado no projeto:

```
firstMulop = new HashSet();
firstMulop.add(Token.Kind.TIMES);
firstMulop.add(Token.Kind.SLASH);
firstMulop.add(Token.Kind.REMAINDER);

firstActPars = new HashSet();
firstActPars.add(Token.Kind.LEFT_PARENTHESIS);

firstAddop = new HashSet();
firstAddop.add(Token.Kind.PLUS);
firstAddop.add(Token.Kind.MINUS);

firstDesignator = new HashSet();
firstDesignator.add(Token.Kind.IDENTIFIER);

firstFactor = new HashSet(firstDesignator);
firstFactor.add(Token.Kind.NUMBER);
firstFactor.add(Token.Kind.CHARACTER);
firstFactor.add(Token.Kind.NEW);
firstFactor.add(Token.Kind.LEFT_PARENTHESIS);
                    ...
```

Como o analisador léxico determina toda a base necessária para os dados entrantes na fase de Análise Sintática não foi necessária a inserção de estruturas de dados mais complexas.

2.4 Ambiente de testes e resultados

Para auxiliar o desenvolvimento do analisador sintático foi construída uma classe de testes denominada *TestParser*, seu objetivo é realizar baterias de testes sobre quaisquer regras gramaticais da linguagem MicroJava. Seu valor para o desenvolvimento do projeto é inestimável já que o número de possibilidades e níveis de recursão que as regras podem alcançar vão além imaginação humana, o que complica e muito compreender e sanar possíveis erros. Com uma classe que realiza testes sobre as regras pode se modularizar os testes sobre pequenas regiões de código além de criar entradas controladas que irão ajudar a compreender possíveis problemas e certificar sobre seu funcionamento.

A classe TestaParser pode ser vista o fragmento de código abaixo:

```
public class TestParser {

    public static void main(String args[]) {
        if (args.length == 0) {
            executeTests();
        } else {
            for (int i = 0; i < args.length; ++i) {
                reportFile(args[i]);
            }
        }
    }

    private static void reportFile(String filename) {
        try {
            report("File: " + filename,
                    new InputStreamReader(new FileInputStream(filename)));
        } catch (IOException e) {
            System.out.println("-- cannot open file " + filename);
        }
    }

    private static void report(String header, Reader reader) {
        System.out.println(header);
        Parser parser = new Parser(reader);
        parser.parse();
        System.out.println(parser.errors + " errors detected\n");
    }

    private static Parser createParser(String input) {
        Parser parser = new Parser(new StringReader(input));
        parser.showSemanticError = false;
        parser.code.showError = false;
        parser.scan(); // get first token
        return parser;
    }

    private static void executeTests() {
        testActPars();
        testAddop();
        testBlock();
        testClassDecl();
```

```
        testCondition();
        testConstDecl();
        testDesignator();
        testExpr();
        testFactor();
        testFormPars();
        testMethodDecl();
        testMulop();
        testProgram();
        testRelop();
        testStatement();
        testTerm();
        testType();
        testVarDecl();
    }

    private static void testActPars() {
        System.out.println("Test: ActPars");
        createParser("()").ActPars();
        createParser("(2)").ActPars();
        createParser("(-2)").ActPars();
        createParser("(-2+3)").ActPars();
        createParser("(2*3)").ActPars();
        createParser("(-2*3)").ActPars();
        createParser("(2,3*4*5)").ActPars();
        createParser("(-2*3+4)").ActPars();
        createParser("(-2*3+4*5+6+7)").ActPars();
        createParser("(-2*3+4, 5+6+7)").ActPars();
        createParser("(-2*3+4, 5+6+7, 8*9+0)").ActPars();

        System.out.println("Test: ActPars errors");
        createParser("(").ActPars();
        createParser("(+2)").ActPars();
        createParser("(2,)").ActPars();
        createParser("(2,3+4,)").ActPars();
        System.out.println();
    }

    private static void testAddop() {
        System.out.println("Test: Addop");
        createParser("+").Addop();
        createParser("-").Addop();

        System.out.println("Test: Addop errors");
        createParser("1").Addop();
        createParser("=").Addop();
        System.out.println();
    }

    private static void testBlock() {
        System.out.println("Test: Block");
        createParser("{ }").Block();
        createParser("{ ; }").Block();
        createParser("{ return 2; }").Block();
        createParser("{ x = 3; return 2; }").Block();

        System.out.println("Test: Block errors");
        createParser(" ; }").Block();
        createParser("{ ;").Block();
        System.out.println();
    }
```

```
    private static void testClassDecl() {
        System.out.println("Test: ClassDecl");
        createParser("class A { }").ClassDecl();
        createParser("class B { int x; }").ClassDecl();

        System.out.println("Test: ClassDecl errors");
        createParser("C { }").ClassDecl();
        createParser("class { }").ClassDecl();
        createParser("class D }").ClassDecl();
        createParser("class E int y; }").ClassDecl();
        createParser("class F { ").ClassDecl();
        createParser("class G { int z; ").ClassDecl();
        System.out.println();
    }

    private static void testCondition() {
        System.out.println("Test: Condition");
        createParser("2 == 3").Condition();

        System.out.println("Test: Condition errors");
        createParser("  > 4").Condition();
        createParser("5   6").Condition();
        createParser("7 <  ").Condition();
        System.out.println();
    }

    private static void testConstDecl() {
        System.out.println("Test: ConstDecl");
        createParser("final int  x = 2;").ConstDecl();
        createParser("final int  y = 'Y';").ConstDecl();
        createParser("final char w = 3;").ConstDecl();
        createParser("final char z = 'Z';").ConstDecl();

        System.out.println("Test: ConstDecl errors");
        createParser("      int  c = 1;").ConstDecl();
        createParser("final      d = 2;").ConstDecl();
        createParser("final int    = 3;").ConstDecl();
        createParser("final char e   'E';").ConstDecl();
        createParser("final char f =  ;").ConstDecl();
        createParser("final char g = 5 ").ConstDecl();
        System.out.println();
    }

    private static void testDesignator() {
        System.out.println("Test: Designator");
        createParser("x").Designator();
        createParser("x.y").Designator();
        createParser("x.y.z.w").Designator();
        createParser("x[2]").Designator();
        createParser("x[3][4]").Designator();
        createParser("x[5].y[6].z[7][8]").Designator();

        System.out.println("Test: Designator errors");
        createParser("x. ").Designator();
        createParser(" .y").Designator();
        createParser("x. .z.w").Designator();
        createParser("x[]").Designator();
        createParser("x[3 [4]").Designator();
        System.out.println();
    }
```

```
    private static void testExpr() {
        System.out.println("Test: Expr");
        createParser("2").Expr();
        createParser("-2 + 3").Expr();
        createParser("2 + 3 * 4 + 5 / 6 * (7 + 8) % 9").Expr();
        createParser("-(x + ((6 + 3) / 3) + y) * z").Expr();
        createParser("-x + 'y' * z").Expr();
        createParser(" - 'y' ").Expr();
        createParser(" - '\t' + 3 * 'z' % '\n' ").Expr();
        createParser(" -'\n'[1].wtf ").Expr();

        System.out.println("Test: Expr errors");
        createParser("2+").Expr();
        createParser("x++").Expr();
        createParser("--i").Expr();
        createParser("(3 'z')").Expr();
        createParser("(a / (b + c) * e").Expr();
        System.out.println();
    }

    private static void testFactor() {
        System.out.println("Test: Factor");
        createParser("x").Factor();
        createParser("2").Factor();
        createParser(" 'c' ").Factor();
        createParser("new var").Factor();
        createParser("new char[size]").Factor();
        createParser("new a[2 * t + '&']").Factor();
        createParser("(abc + 2 * 3)").Factor();

        System.out.println("Test: Factor errors");
        createParser("new").Factor();
        createParser("new int[]").Factor();
        createParser("new char[size").Factor();
        createParser("new a[2 * t + ]").Factor();
        createParser("(abc (-2 + 3) * 4").Factor();
        System.out.println();
    }

    private static void testFormPars() {
        System.out.println("Test: FormPars");
        createParser("int x").FormPars();
        createParser("char c, char[] d").FormPars();
        createParser("int[] y, char[] e, int z, char f").FormPars();

        System.out.println("Test: FormPars errors");
        createParser("int x,").FormPars();
        createParser("char c, char[]").FormPars();
        System.out.println();
    }

    private static void testMethodDecl() {
        System.out.println("Test: MethodDecl");
        createParser("void f() { }").MethodDecl();
        createParser("int g() { if ('\n' == 2) return ; }").MethodDecl();
        createParser("char[] h(int n) { return new array[n];
}").MethodDecl();
        createParser("int F(int a, int b) int i, j; { return a*i + b*j;
}").MethodDecl();
```

```
        System.out.println("Test: MethodDecl errors");
        createParser("int g() if ('\n' == 2) return ; }").MethodDecl();
        createParser("int gg() { ").MethodDecl();
        createParser("char[] int n) { }").MethodDecl();
        createParser("int F(int a, int b) int i, j { }").MethodDecl();
        System.out.println();
    }

    private static void testMulop() {
        System.out.println("Test: Mulop");
        createParser("*").Mulop();
        createParser("/").Mulop();
        createParser("%").Mulop();

        System.out.println("Test: Mulop errors");
        createParser("+").Mulop();
        createParser("$").Mulop();
        System.out.println();
    }

    private static void testProgram() {
        System.out.println("Test: Program");
        createParser("program P { }").Program();
        createParser("program Main { void main() {;} }").Program();
        createParser("program ABC {"
                + " void f() {;}"
                + " int g(int x) { return x*2; } }").Program();
        createParser("program R "
                + " final int x = 3;"
                + " class CL { }"
                + " CL obj;"
                + " { }").Program();

        System.out.println("Test: Program errors");
        createParser("P { }").Program();
        createParser("program { }").Program();
        createParser("program C }").Program();
        createParser("program D {").Program();
        createParser("program P1 { void main() { }").Program();
        createParser("program P2 int x { void main() { } }").Program();
        System.out.println();
    }

    private static void testRelop() {
        System.out.println("Test: Relop");
        createParser("==").Relop();
        createParser("!=").Relop();
        createParser("> ").Relop();
        createParser(">=").Relop();
        createParser("< ").Relop();
        createParser("<=").Relop();

        System.out.println("Test: Relop errors");
        createParser("!").Relop();
        createParser("=!").Relop();
        createParser("+").Relop();
        createParser("--").Relop();
        createParser("*").Relop();
        System.out.println();
    }
```

```
    private static void testStatement() {
        System.out.println("Test: Statement");
        createParser(";").Statement();
        createParser("{ ; }").Statement();
        createParser("{ { { ; } } }").Statement();
        createParser("x = 2;").Statement();
        createParser("f();").Statement();
        createParser("g(2, 'c', 3 * 4);").Statement();
        createParser("if (1 == 2) y = 1; else y = 0;").Statement();
        createParser("if (3 == 3) w = 1; else ;").Statement();
        createParser("while (0 > '1') ;").Statement();
        createParser("return ;").Statement();
        createParser("return 2 * '3' + 4;").Statement();
        createParser("read(x);").Statement();
        createParser("print(2);").Statement();
        createParser("print(3, 4);").Statement();

        System.out.println("Test: Statement errors");
        createParser("f(;").Statement();
        createParser("g(2, 'c';").Statement();
        createParser("if (1 == 2) y = 1  else y = 0;").Statement();
        createParser("if (3 == 3) w = 1; else ").Statement();
        createParser("return").Statement();
        createParser("return 2 * '3' + ;").Statement();
        createParser("read(1);").Statement();
        createParser("read(x, y);").Statement();
        createParser("print(3, 4, 5);").Statement();
        System.out.println();
    }

    private static void testTerm() {
        System.out.println("Test: Term");
        createParser("2").Term();
        createParser("3 * 4").Term();
        createParser("5 * (6 + 7) / 8 % 9").Term();

        System.out.println("Test: Term errors");
        createParser("2 %").Term();
        createParser("* 4").Term();
        createParser("5 * (6 / 7 8)").Term();
        createParser("11 / (12 % 2").Term();
        System.out.println();
    }

    private static void testType() {
        System.out.println("Test: Type");
        createParser("int").Type();
        createParser("int[]").Type();
        createParser("char").Type();
        createParser("char[]").Type();
        createParser("xyz").Type();

        System.out.println("Test: Type errors");
        createParser("int[").Type();
        createParser("int[2]").Type();
        createParser("void").Type();
        createParser("345").Type();
        System.out.println();
    }

    private static void testVarDecl() {
```

```
        System.out.println("Test: VarDecl");
        createParser("int x;").VarDecl();
        createParser("int x, y;").VarDecl();
        createParser("int x, x, x;").VarDecl();

        System.out.println("Test: VarDecl errors");
        createParser("int x").VarDecl();
        createParser("int x y;").VarDecl();
        createParser("int x, ,y;").VarDecl();
        System.out.println();
    }

}
```

Cada um dos métodos de testes são especializados em testar determinadas regras da gramática. Por exemplo, o método testType realiza testes sobre a regra Type definida na gramática da linguagem MicroJava, por padrão foram colocados algumas entradas que devem gerar sucesso e logo em seguida possíveis casos de erro. Sua execução permite analisar o que cada uma dessas entradas obteve como saída do analisador sintático.

Caso o TestParser seja executado com a bateria de testes sobre o analisador sintático atual obteriamos o seguinte relatório de resultados gerados pelo sistema:

```
Test: ActPars
Test: ActPars errors
Error: line   1, column   2: Expected ')'.
Error: line   1, column   2: Expected ')'.
Error: line   1, column   4: Identifier not found.
Error: line   1, column   8: Identifier not found.

Test: Addop
Test: Addop errors
Error: line   1, column   1: expected '-'or '+'
Error: line   1, column   1: expected '-'or '+'

Test: Block
Test: Block errors
Error: line   1, column   2: Expected: '{'.
Error: line   1, column   4: Expected: '}'.

Test: ClassDecl
Test: ClassDecl errors
Error: line   1, column   1: Class definition not found. Expected 'class'.
Error: line   1, column   7: Class identifier not valid.
Error: line   1, column   9: Class left brace not found.
Error: line   1, column   9: Class left brace not found.
Error: line   1, column  11: End of file not expected, close class
declaration braces.
Error: line   1, column  18: End of file not expected, close class
declaration braces.

Test: Condition
Test: Condition errors
Error: line   1, column   3: Identifier not found.
Error: line   1, column   5: Invalid condition operator. Expected any of
these: == != > >= < <=
```

```
Error: line   1, column   6: Identifier not found.

Test: ConstDecl
Test: ConstDecl errors
Error: line   1, column   7: Expected final keyword.
Error: line   1, column  14: Expected identifier name.
Error: line   1, column  14: Expected identifier name.
Error: line   1, column  16: Expected '='.
Error: line   1, column  17: Expected number or character.
Error: line   1, column  18: Expected ';' at end of declaration.

Test: Designator
Test: Designator errors
Error: line   1, column   4: Expected identifier.
Error: line   1, column   2: Identifier not found.
Error: line   1, column   4: Expected identifier.
Error: line   1, column   3: Identifier not found.
Error: line   1, column   5: Expected ']'.

Test: Expr
Test: Expr errors
Error: line   1, column   3: Identifier not found.
Error: line   1, column   3: Identifier not found.
Error: line   1, column   4: Identifier not found.
Error: line   1, column   2: Identifier not found.
Error: line   1, column   4: Expected ')'.
Error: line   1, column  17: Expected ')'.

Test: Factor
Test: Factor errors
Error: line   1, column   4: Expected identifier.
Error: line   1, column   9: Identifier not found.
Error: line   1, column  14: Expected ']'.
Error: line   1, column  15: Identifier not found.
Error: line   1, column  18: Expected ')'.

Test: FormPars
Test: FormPars errors
Error: line   1, column   7: Expected type identifier.
Error: line   1, column  15: Expected identifier.

Test: MethodDecl
Test: MethodDecl errors
Error: line   1, column   9: Expected: '{'.
Error: line   1, column  12: Expected: '}'.
Error: line   1, column  12: Expected '('.
Error: line   1, column  13: Expected identifier.
Error: line   1, column  30: Expected ';' at the end of variable
declaration.

Test: Mulop
Test: Mulop errors
Error: line   1, column   1: expected '*', '/' or '%'
Error: line   1, column   1: expected '*', '/' or '%'

Test: Program
Test: Program errors
Error: line   1, column   1: Expected 'program'.
Error: line   1, column   9: Program name not declared.
Error: line   1, column  11: Expected '{'.
Error: line   1, column  12: Expected '}'.
```

```
Error: line   1, column  29: Expected '}'.
Error: line   1, column  18: Expected ';' at the end of variable
declaration.

Test: Relop
Test: Relop errors
Error: line   1, column   1: Invalid condition operator. Expected any of
these: == != > >= < <=
Error: line   1, column   1: Invalid condition operator. Expected any of
these: == != > >= < <=
Error: line   1, column   1: Invalid condition operator. Expected any of
these: == != > >= < <=
Error: line   1, column   1: Invalid condition operator. Expected any of
these: == != > >= < <=
Error: line   1, column   1: Invalid condition operator. Expected any of
these: == != > >= < <=

Test: Statement
Test: Statement errors
Error: line   1, column   3: Expected ')'.
Error: line   1, column   9: Expected ')'.
Error: line   1, column  20: Expected ';'
Error: line   1, column  25: Expected ';'
Error: line   1, column   7: Expected ';'
Error: line   1, column  18: Identifier not found.
Error: line   1, column   6: Identifier not found.
Error: line   1, column   7: Expected ')'
Error: line   1, column  11: Expected ')'

Test: Term
Test: Term errors
Error: line   1, column   4: Identifier not found.
Error: line   1, column   1: Identifier not found.
Error: line   1, column  12: Expected ')'.
Error: line   1, column  13: Expected ')'.

Test: Type
Test: Type errors
Error: line   1, column   5: Expected ']'.
Error: line   1, column   5: Expected ']'.
Error: line   1, column   1: Expected type identifier.
Error: line   1, column   1: Expected type identifier.

Test: VarDecl
Test: VarDecl errors
Error: line   1, column   6: Expected ';' at the end of variable
declaration.
Error: line   1, column   7: Expected ',' between 'x' and 'y'.
Error: line   1, column   8: Expected an identifier after ',' at variable
declaration.
```

*Obs: As entradas que obtiveram exito são automáticamente removidas do resultado já que no geral se pretende encontrar informações sobre as mensagens de erro, o que apenas comprometeria sua visualização.

Averiguando cuidadosamente a saída dos testes, se percebe que o analisador sintático obteve exito sobre todas as entradas da bateria de testes, seja aceitando entradas válidas e rejeitando entradas incorretas. Vale ressaltar que a classe

TestParser foi utilizada exaustivamente em conjunto com a construção de cada uma das regras o que permitiu um processo de correção fino.

3.0 Análise Semântica

A Análise Semântica deve verificar se a semântica de uma seqüência *tokens* sintaticamente corretos é válida. Por semântica deve se entender a natureza lógica de uma seqüência de *tokens*, ou seja, aquela que faz sentido ser utilizada.

3.1 Definições Semânticas

Na análise semântica diferentemente das etapas descritas anteriormente não existe uma especificação formal que descreva de forma completa e total todas as características semânticas de uma linguagem. Esta compreensão fica mais a cargo do projetista, que ao ler os manuais da linguagem e anotações de sua especificação os interpreta e as define no compilador. Como auxilio, a documentação do MicroJava disponibilizou um documento denominado “Handsout” que inferiu uma série de características que ajudaram a compreensão e direcionamento na construção do analisador semântico.

3.2 Detalhes de implementação

O analisador semântico foi implementado sobre o módulo de análise sintática, o que acabou por reaproveitar toda sua infra-estrutura previamente montado. Desta maneira não foi necessário uma adição de estruturas relevantes, exceto aquelas que foram disponibilizadas como base para o projeto, tais como a Tabela Sintática(TS). O processo de Análise Sintática está intimamente ligado ao gerenciamento da TS e este é feito adicionando-se declarações à tabela e realizando buscas quando identificadores quaisquer aparecem. Quando identificadores utilizados não são encontrados ocorrem erros semânticos devido a não tipificação do identificador, já quando um identificador é encontrado na tabela outros tipos de verificações são possíveis e dependem de cada um dos casos que se quer analisar. Por exemplo, em expressões de atribuição além de os identificadores estarem presentes na TS estes devem pertencer a tipos compatíveis, do contrário erros semânticos serão disparados.

3.3 Ambiente de testes e resultados

Nesta etapa o nível de complexidade aumentou devido tanto ao acumulo de implementações realizadas em fases anteriores como ao modo pelo qual a implementação foi feita, ao ter que realizar adições de código sobre o módulo sintático, isto corroborou e muito para o surgimento de erros e uma classe de testes se tornou ainda mais fundamental para a construção do analisador semântico. Como nas etapas anteriores foi construída uma classe para realização de baterias de testes sobre cada uma das regras da linguagem com o objetivo de averiguar se o analisador semântico estava funcionando corretamente. A implementa cão da classe TestParserSemantic segue abaixo:

```
public class TestParserSemantic {

    public static void main(String args[]) {
        if (args.length == 0) {
            executeTests();
        } else {
            for (int i = 0; i < args.length; ++i) {
                reportFile(args[i]);
            }
        }
    }

    private static void reportFile(String filename) {
        try {
            report("File: " + filename,
                    new InputStreamReader(new FileInputStream(filename)));
        } catch (IOException e) {
            System.out.println("-- cannot open file " + filename);
        }
    }

    private static void report(String header, Reader reader) {
        System.out.println(header);
        Parser parser = new Parser(reader);
        parser.parse();
        System.out.println(parser.errors + " errors detected\n");
    }

    private static Parser createParser(String input) {
        Parser parser = new Parser(new StringReader(input));
        parser.showSemanticError = true;
        parser.code.showError = false;
        parser.scan(); // get first token
        return parser;
    }

    private static void executeTests() {
        testClassDecl();
        testCondition();
        testConstDecl();
        testDesignator();
        testExpr();
        testFactor();
        testFormPars();
```

```
        testMethodDecl();
        testProgram();
        testStatement();
        testTerm();
        testType();
        testVarDecl();
    }

    private static void testClassDecl() {
        System.out.println("Test: ClassDecl");
        createParser("class A {}").ClassDecl();
        createParser("class B { int b; }").ClassDecl();
        createParser("class C { int c1; char[] c2; }").ClassDecl();
        createParser("class D {} class E { D d;
}").ClassDecl().ClassDecl();
        createParser("class F { F[] f; }").ClassDecl();

        System.out.println("Test: ClassDecl errors");
        createParser("class int {}").ClassDecl();
        createParser("class char {}").ClassDecl();
        createParser("class H { int H; }").ClassDecl();
        createParser("class J { int j; char j; }").ClassDecl();
        // invalid type (X not declared)
        createParser("class K { X k; }").ClassDecl();
        // atribute name is the same of the type of variable
        createParser("class L {} class M { L L;
}").ClassDecl().ClassDecl();
        createParser("class N { N N; }").ClassDecl();
        System.out.println();
    }

    private static void testCondition() {
        System.out.println("Test: Condition");
        createParser("2 == 3").Condition();
        createParser("int[] a, b; a == b").VarDecl().Condition();
        createParser("char[] c, d; c != d").VarDecl().Condition();
        createParser("class E{} E e, f; e == f   e != f")
                .ClassDecl().VarDecl().Condition().Condition();

        System.out.println("Test: Condition errors");
        createParser("2 + 3 == 'a'").Condition();
        createParser("int[] ia, ib; ia > ib").VarDecl().Condition();
        createParser("class X{} X x, y; x <=
y").ClassDecl().VarDecl().Condition();
        createParser("class Z{} Z z; int[] r; r == z")
                .ClassDecl().VarDecl().VarDecl().Condition();
        System.out.println();
    }

    private static void testConstDecl() {
        System.out.println("Test: ConstDecl");
        createParser("final int b = 2;").ConstDecl();
        createParser("final char c = 'c';").ConstDecl();

        System.out.println("Test: ConstDecl errors");
        createParser("final int d = 'd';").ConstDecl();
        createParser("final char e = 1;").ConstDecl();
        // array is not accept
        createParser("final int[] f = 1;").ConstDecl();
        createParser("final char[] g = 'g';").ConstDecl();
        System.out.println();
```

```
    }

    private static void testDesignator() {
        System.out.println("Test: Designator");
        createParser("class A {} A a; a = new A; a")
                .ClassDecl().VarDecl().Statement().Designator();
        createParser("class B { int bb; } B b; b = new B; b.bb")
                .ClassDecl().VarDecl().Statement().Designator();
        createParser("class C { char c1; int c2; } C c; c = new C; c.c1
c.c2")

.ClassDecl().VarDecl().Statement().Designator().Designator();
        createParser("int[] d; d[2]").VarDecl().Designator();

        System.out.println("Test: Designator errors");
        createParser("int x; x.y").VarDecl().Designator();
        createParser("class M { int m; } M.m").ClassDecl().Designator();
        System.out.println();
    }

    private static void testExpr() {
        System.out.println("Test: Expr");
        createParser("2 + 3").Expr();
        createParser("-4").Expr();
        createParser("-5 * 6").Expr();
        createParser("int a; a * 6").VarDecl().Expr();
        createParser("int b, c, d; b * 6 + c * d").VarDecl().Expr();

        System.out.println("Test: Expr errors");
        createParser("-'a'").Expr();
        createParser("-2 * 'b'").Expr();
        createParser("-3 * 'c' * 4").Expr();
        createParser("-5 * 'd' * 6 + 7").Expr();
        createParser("char e;  e - 'f' % 'g'").VarDecl().Expr();
        createParser("void m(){}  m + 8").MethodDecl().Expr();
        System.out.println();
    }

    private static void testFactor() {
        System.out.println("Test: Factor");
        // new
        createParser("new int[2]").Factor();
        createParser("int size; new char[size]").VarDecl().Factor();
        createParser("class A {}  new A").ClassDecl().Factor();
        createParser("class B {}  new B[2]").ClassDecl().Factor();
        // designator: variable
        createParser("int i; i").VarDecl().Factor();
        // designator: method */
        createParser("int  m1(int i, int j) {}    m1(2,
3)").MethodDecl().Factor();

        System.out.println("Test: Factor errors");
        // new
        createParser("new int").Factor();
        createParser("new char").Factor();
        createParser("new X").Factor();
        createParser("new Y[2]").Factor();
        createParser("class G {}  new G['b']").ClassDecl().Factor();
        // designator: variable
        createParser("int w; w()").VarDecl().Factor();
        // designator: method
```

```
        createParser("int  me1() {}    me1(1)").MethodDecl().Factor();
        createParser("int  me2(int i, int j) {}
me2(2)").MethodDecl().Factor();
        createParser("char me3(char c) {}
me3(3)").MethodDecl().Factor();
        createParser("char me4(int i, int j, char k) {}    me4(4, 'c',
6)").MethodDecl().Factor();
        // void methods (procedures) cannot be used in a expression
context.
        // Factor is always called in an expression context (Expr -> Term
-> Factor)
        createParser("void m5() {}    m5()").MethodDecl().Factor();
        createParser("void m6(char c) {}
m6('c')").MethodDecl().Factor();
        System.out.println();
    }

    private static void testFormPars() {
        System.out.println("Test: FormPars");
        createParser("int i, char c, int[] ia, char[] ca").FormPars();
        createParser("class A {}  A a, A[] aa").ClassDecl().FormPars();

        System.out.println("Test: FormPars errors");
        createParser("int char, int char").FormPars();
        createParser("B b, C[] c").FormPars();
        System.out.println();
    }

    private static void testMethodDecl() {
        System.out.println("Test: MethodDecl");
        createParser("int[] a(int[] ia, char[] ca) int i; char c;
{}").MethodDecl();
        createParser("void b() int i; char c; {}").MethodDecl();
        createParser("class C {} \n C c() {}").ClassDecl().MethodDecl();
        createParser("class D {} \n D[] d(D dp) D dv;
{}").ClassDecl().MethodDecl();

        System.out.println("Test: MethodDecl errors");
        createParser("int char() {}").MethodDecl();
        createParser("void m(G g) {}").MethodDecl();
        System.out.println();
    }

    private static void testProgram() {
        System.out.println("Test: Program");
        createParser("program P { void main() {} }").Program();
        createParser("program ABC {"
                + "     void f() {}"
                + "     void main() { return ; }"
                + "     int g(int x) { return x*2; }"
                + "   }").Program();

        System.out.println("Test: Program errors");
        createParser("program E1 { }").Program();
        createParser("program E2 { int main() { } }").Program();
        createParser("program E3 { void main(int i) { } }").Program();
        System.out.println();
    }

    private static void testStatement() {
        System.out.println("Test: Statement");
```

```
        // Designator "=" Expr ";"
        createParser("int b; b = 2;").VarDecl().Statement();
        createParser("char c, d; c = d;").VarDecl().Statement();
        createParser("int[] e; e[3] = 3;").VarDecl().Statement();
        createParser("class A { int i; } A a; a.i =
4;").ClassDecl().VarDecl().Statement();
        createParser("int[] ia; ia = new int[5];").VarDecl().Statement();
        createParser("char[] ca; ca = new
char[6];").VarDecl().Statement();
        // Designator ActPars ";"
        createParser("void f() {} f();").MethodDecl().Statement();
        //  "print" "(" Expr ["," number] ")" ";"
        createParser("print(2); print('c');").Statement().Statement();
        createParser("print(3, 4);").Statement();
        // "read" "(" Designator ")" ";"
        createParser("int i; read(i);").VarDecl().Statement();
        createParser("char c; read(c);").VarDecl().Statement();
        createParser("char[] ca; read(ca[2]);").VarDecl().Statement();
        createParser("class C {int i;} C o;
read(o.i);").ClassDecl().VarDecl().Statement();
        //  "return" [Expr] ";"
        createParser("void f() { return ;}").MethodDecl();
        createParser("int g() { return 2;}").MethodDecl();
        createParser("char h() { return 'c';}").MethodDecl();

        System.out.println("Test: Statement errors");
        // Designator "=" Expr
        createParser("int b; b = 'c';").VarDecl().Statement();
        createParser("char[] c; c[4] = 4;").VarDecl().Statement();
        createParser("class A { int i; } A a; a.i =
'4';").ClassDecl().VarDecl().Statement();
        createParser("int[] ia; ia = new char[5];").VarDecl().Statement();
        createParser("char[] ca; ca = new int[6];").VarDecl().Statement();
        // Designator ActPars
        createParser("int g; g();").VarDecl().Statement();
        //  "print" "(" Expr ["," number] ")" ";"
        createParser("class C {}  C c;
print(c);").ClassDecl().VarDecl().Statement();
        createParser("print(5, 'd');").Statement();
        // "read" "(" Designator ")" ";"
        createParser("class C {}  C c;
read(c);").ClassDecl().VarDecl().Statement();
        //  "return" [Expr] ";"
        createParser("void p() { return 1; }").MethodDecl();
        createParser("int q() { return 'c';}").MethodDecl();
        createParser("char r() { return 2;}").MethodDecl();
        createParser("int s() { return ;}").MethodDecl();
        System.out.println();
    }

    private static void testTerm() {
        System.out.println("Test: Term");
        createParser("2 * 3").Term();
        createParser("4").Term();
        createParser("5 * (-6)").Term();
        createParser("'a'").Term();
        createParser("int a, b; a * (-7) * b").VarDecl().Term();
        createParser("char a; a").VarDecl().Term();

        System.out.println("Test: Term errors");
        createParser("2 * 'b'").Term();
```

```
        createParser("'c' * 3").Term();
        createParser("char d; d * 'e'").VarDecl().Term();
        System.out.println();
    }

    private static void testType() {
        System.out.println("Test: Type");
        createParser("int").Type();
        createParser("int[]").Type();
        createParser("char").Type();
        createParser("char[]").Type();
        createParser("class A {}  A").ClassDecl().Type();

        System.out.println("Test: Type errors");
        createParser("class").Type();
        createParser("MyClass").Type();
        System.out.println();
    }

    private static void testVarDecl() {
        System.out.println("Test: VarDecl");
        createParser("int x;").VarDecl();
        createParser("int x, y;").VarDecl();
        createParser("int x, X;").VarDecl(); // case-sensitive variables
        createParser("int x, y, z, X, Y, Z;").VarDecl();

        System.out.println("Test: VarDecl errors");
        createParser("int int;").VarDecl();
        createParser("int char;").VarDecl();
        createParser("char int;").VarDecl();
        createParser("char char;").VarDecl();
        createParser("int s, s;").VarDecl();
        createParser("int t, char, t, int;").VarDecl();
        createParser("int x, w, p, q, r, w;").VarDecl();
        System.out.println();
    }

}
```

Cada um dos métodos de teste realiza verificações especificas quanto à semântica de cada uma das regras da linguagem implementada. Por exemplo, o método *testTerm* realiza testes semânticos sobre a regra Term, cuja finalidade é descrever operações de multiplicação, divisão, resto entre outras. Quando submetidos aos testes, a regra Term deverá disparar erros semânticos adequados as entradas dos testes.

Com a execução da bateria de testes atual sobre o analisador semântico foi obtido o seguinte relatório de resultados:

```
Test: ClassDecl
Test: ClassDecl errors
Error: line   1, column  11: Semantic: 'int' is not a non-keyword
identifier.
Error: line   1, column  12: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column  16: Semantic: 'H' is a classname. Expected non-
```

```
classname identifier.
Error: line   1, column  24: Semantic: 'j' was already declared.
Error: line   1, column  13: Semantic: 'X' is not a keyword or a
classname.
Error: line   1, column  14: Semantic: Invalid type for 'k'.
Error: line   1, column  25: Semantic: 'L' is a classname. Expected non-
classname identifier.
Error: line   1, column  14: Semantic: 'N' is a classname. Expected non-
classname identifier.

Test: Condition
Test: Condition errors
Error: line   1, column  13: Semantic: Incompatible types: int char
Error: line   1, column  31: Semantic: Incompatible types: r z

Test: ConstDecl
Test: ConstDecl errors
Error: line   1, column  18: Semantic: Expected int value.
Error: line   1, column  17: Semantic: Expected char value.
Error: line   1, column  10: Expected identifier name.
Error: line   1, column  11: Expected identifier name.

Test: Designator
Test: Designator errors
Error: line   1, column  10: Semantic: 'x' is not an object.
Error: line   1, column  21: Semantic: 'M' is a classname. Expected non-
classname identifier.

Test: Expr
Test: Expr errors
Error: line   1, column   5: Semantic: Got 'a' (char). Expected int type.
Error: line   1, column   9: Semantic: Got 'b' (char). Expected int type.
Error: line   1, column  10: Semantic: Got 'c' (char). Expected int type.
Error: line   1, column  10: Semantic: Got 'd' (char). Expected int type.
Error: line   1, column  12: Semantic: Got e (char). Expected int type.
Error: line   1, column  18: Semantic: Got 'f' (char). Expected int type.
Error: line   1, column  23: Semantic: Got 'g' (char). Expected int type.
Error: line   1, column  15: Semantic: Got m (method). Expected int type.

Test: Factor
Test: Factor errors
Error: line   1, column   8: Semantic: Class type expected
Error: line   1, column   9: Semantic: Class type expected
Error: line   1, column   6: Semantic: 'X' not declared.
Error: line   1, column   6: Semantic: 'Y' not declared.
Error: line   1, column  22: Semantic: Got 'b' (char). Expected int type.
Error: line   1, column   9: Semantic: 'w' is not method.
Error: line   1, column  23: Semantic: Wrong number of parameters (1) for
'me1'. Expected 0 parameters.
Error: line   1, column  35: Semantic: Wrong number of parameters (1) for
'me2'. Expected 2 parameters.
Error: line   1, column  29: Semantic: Incompatible parameter type. Got
int. Expected char.
Error: line   1, column  51: Semantic: Incompatible parameter type. Got
char. Expected int.
Error: line   1, column  51: Semantic: Incompatible parameter type. Got
int. Expected char.
Error: line   1, column  21: Semantic: procedure called as a function
Error: line   1, column  30: Semantic: procedure called as a function

Test: FormPars
```

```
Test: FormPars errors
Error: line   1, column   9: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column  19: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column   3: Semantic: 'B' is not a keyword or a
classname.
Error: line   1, column   4: Semantic: Invalid type for 'b'.
Error: line   1, column   7: Semantic: 'C' is not a keyword or a
classname.
Error: line   1, column  11: Semantic: Invalid type for 'c'.

Test: MethodDecl
Test: MethodDecl errors
Error: line   1, column   9: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column  10: Semantic: 'G' is not a keyword or a
classname.
Error: line   1, column  11: Semantic: Invalid type for 'g'.

Test: Program
Test: Program errors
Error: line   1, column  14: Semantic: Program must contain a 'main'
method.
Error: line   1, column  29: Semantic: 'main' method must be void and not
have parameters.
Error: line   1, column  35: Semantic: 'main' method must be void and not
have parameters.

Test: Statement
Test: Statement errors
Error: line   1, column  15: Semantic: Incompatible types: b (int)    char
Error: line   1, column  19: Semantic: Incompatible types: c (char[])
int
Error: line   1, column  34: Semantic: Incompatible types: i (int)    char
Error: line   1, column  27: Semantic: Incompatible types: ia (int[])
char[]
Error: line   1, column  27: Semantic: Incompatible types: ca (char[])
int[]
Error: line   1, column   9: Semantic: 'g' is not a method.
Error: line   1, column  26: Semantic: Expected int or char.
Error: line   1, column  10: Expected a number.
Error: line   1, column  24: Semantic: Expected int or char.
Error: line   1, column  20: Semantic: void method 'p' must not return a
value
Error: line   1, column  21: Semantic: Got char. Expected int.
Error: line   1, column  20: Semantic: Got int. Expected char.
Error: line   1, column  18: Semantic: Method 's' must return void.

Test: Term
Test: Term errors
Error: line   1, column   8: Semantic: Got 'b' (char). Expected int type.
Error: line   1, column   5: Semantic: Got 'c' (char). Expected int type.
Error: line   1, column  11: Semantic: Got d (char). Expected int type.
Error: line   1, column  16: Semantic: Got 'e' (char). Expected int type.

Test: Type
Test: Type errors
Error: line   1, column   1: Expected type identifier.
Error: line   1, column   8: Semantic: 'MyClass' is not a keyword or a
classname.
```

```
Test: VarDecl
Test: VarDecl errors
Error: line   1, column   8: Semantic: 'int' is not a non-keyword
identifier.
Error: line   1, column   9: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column   9: Semantic: 'int' is not a non-keyword
identifier.
Error: line   1, column  10: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column   9: Semantic: 's' was already declared.
Error: line   1, column  12: Semantic: 'char' is not a non-keyword
identifier.
Error: line   1, column  15: Semantic: 't' was already declared.
Error: line   1, column  20: Semantic: 'int' is not a non-keyword
identifier.
Error: line   1, column  21: Semantic: 'w' was already declared.
```

Uma análise cuidadosa permite verificar que todos os testes foram realizados com sucesso, o que nos permite afirmar que a etapa está implementada com um sistema estável e altamente confiavel fornecendo assim a base para a etapa seguinte.

4.0 Geração de Código

A Geração de Código consiste na etapa de gerar o código intermediario ou alvo no processo de compilação. O código intermediario é normalmente gerado por se tratar de uma linguagem simples para que métodos de otimização sejam mais facilmente concebíveis. No entanto, nesta implementação serão gerados os códigos na linguagem alvo diretamente sem que haja uma etapa intermediaria utilizada em otimizações. A linguagem alvo é definida pela maquina virtual do MicroJava, e trata-se de uma versão bem parecida ao P-Código amplamente utilizado como linguagem intermediária por compiladores da linguagem Pascal.

4.1 Definições da Máquina Virtual

A Máquina Virtual cujo o código alvo deve ser compreendido é baseado em pilhas. Existem diferentes tipos de pilhas que conferem toda a infra-estrutura necessária para a execução dos programas, isto é, as chamadas de função estão baseadas na pilha *pstack* que armazena os chamados registros de ativação de cada uma das funções em execução. A pilha *estack* tem a função de armazenar os valores e endereços temporários de regiões da *pstack, heap* e *região global* de forma que expressões possam ser calculadas e armazenadas novamente em cada uma das regiões supracitadas. Vale ressaltar que na maior parte do tempo o foco é a utilização da pilha *estack* para a computação de expressões e subsequente armazenamentos nas regiões que servem inicialmente como fonte de dados.

4.2 Detalhes de implementação

A implementação manteve a mesma estrutura previamente construída, adicionando as classes de auxilio disponibilizadas para o projeto. Houve apenas uma etapa de simplificação de procedimentos que se tornaram redundantes com a adição do código para implementação do processo de geração de código.

4.3 Ambiente de testes e resultados

Nesta etapa ao contrário das anteriores onde o foco era testar exatamente o módulo que estava se desenvolvendo, optou-se por realizar baterias de testes completas que testavam não só a geração do código objeto, mas todo o compilador. A metodologia empregada foi ir construindo pequenos programas que utilizassem sempre as caracteristicas implementadas mais recentemente na geração de código e corrigindo evetuais problemas. O processo de testes teve em média a execução de ao menos uma centena de pequenos programas que se comportaram bem com o término de toda a implementação. Com o compilador estável pode se tornar à prova toda a implementação desenvolvendo programas relativamente complexos com a linguagem MicroJava, dentre eles destacamos os programas listados abaixo:

Algoritmo de ordenação Bubble Sort: bubblesort.mj

```
program P
{
    void BubbleSort(int[] v, int n)
        int i, j, aux;
    {
        i = n;
        while(i>0){
            j = 0;
            while(j<i-1){
                if(v[j] > v[j+1]){
                    aux = v[j];
                    v[j] = v[j+1];
                    v[j+1] = aux;
                }
                j = j + 1;
            }
            i = i - 1;
        }
    }

    void PrintVector(int[] v, int n)
        int i;
    {
        i = 0;
        while(i<n){
            print(v[i]);
            print('\n');
            i = i + 1;
        }
    }

    void main()
        int[] a;
        int n, i, j;
    {
        print('n');
        print(':');
        print('\n');
        read(n);
        a = new int[n];

        while(i < n) {
            read(j);
            a[i] = j;
            i = i + 1;
        }

        print('n');
        print('o');
        print('t');
        print('_');
        print('o');
        print('r');
        print('d');
        print('\n');

        PrintVector(a, n);

        print('o');
        print('r');
```

```
        print('d');
        print('\n');

        BubbleSort(a, n);
        PrintVector(a, n);
    }
}
```

Algoritmo de ordenação Insertion Sort: insertionsort.mj

```
program P
{
    void InsertionSort(int[] v, int n)
        int i, j, current, continue;
    {
        i = 1;
        while (i < n) {
            current = v[i];
            j = i - 1;

            continue = 1;
            while (continue == 1) {
                if (j >= 0) {
                    if (v[j] > current) {
                        v[j+1] = v[j];
                        j = j - 1;
                    } else
                        continue = 0;
                } else
                    continue = 0;
            }
            v[j+1] = current;
            i = i + 1;
        }
    }

    void PrintVector(int[] v, int n)
        int i;
    {
        i = 0;
        while(i < n) {
            print(v[i]);
            print('\n');
            i = i + 1;
        }
    }

    void main()
        int[] a;
        int n, i, j;
    {
        print('n');
        print(':');
        print('\n');
        read(n);

        a = new int[n];
        i = 0;
        while(i < n) {
```

```
            read(j);
            a[i] = j;
            i = i + 1;
        }

        InsertionSort(a, n);
        PrintVector(a, n);
    }
}
```

5.0 MicroJava IDE - Um Ambiente Integrado de Desenvolvimento para o MicroJava

Um ambiente de desenvolvimento integrado provê ferramentas para que o desenvolvedor obtenha indices de produção maximizados com a utilização de menus e operações automatizadas que podem ser realizadas em um único ambiente. Pensando nas vantagens que uma IDE propicia, foi desenvolvido um software integrado de apoio ao desenvolvimento para a linguagem MicroJava, denominado MicroJava IDE(MIDE). Este software foi criado especialmente para trabalhar com a linguagem MicroJava em um ambiente de multi-trabalho onde é possível editar, compilar e executar multiplos arquivos fontes com apoio de um identificador que compreende a sintaxe e aplica técnicas de *highlight* facilitando a visualização e interpretação de códigos.
O MIDE foi desenvolvido totalmente em C++ com o apoio da biblioteca de interface gráfica QT amplamente utilizada por softwares livres, sua versão atual é compativel com os sistemas operacionais Windows, Linux e Mac. A Figura 27 mostra o MIDE sendo executado a esquerda em uma distribuição Linux Ubuntu e a direita no Windows XP.

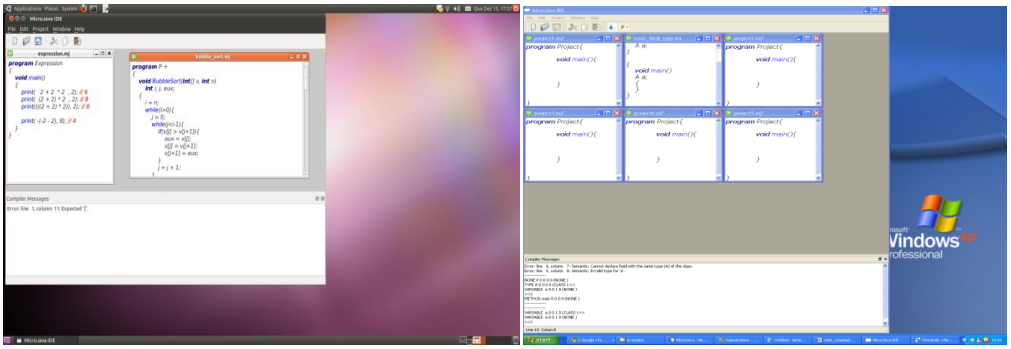

Figura 27 – O MIDE sendo executando no Linux Ubuntu e Windows Xp.

5.1 Menus do MIDE

Esta seção tem o cunho de introduzir de forma resumida as funcionalidades do MIDE, tentando passar sua idéia de trabalho básica para que qualquer desenvolvedor se sinta familirizado com o ambiente.

Ao executar o MIDE a primeira tela a ser visualizada corresponde a Figura 28, onde o usuário possuí a sua disposição as opções contidas no menu na parte superior da janela.

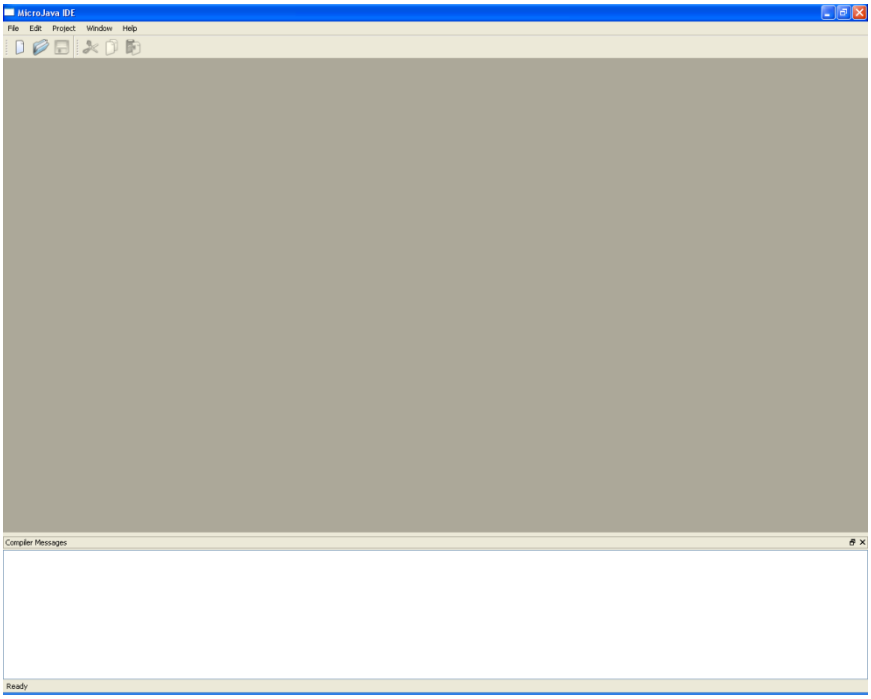

Figura 28 - Tela principal do MIDE.

A primeira operação a ser efetuada deve sempre partir do menu File mostrado na Figura 29, onde são possíveis criar novos arquivos fontes a partir do zero, abrir códigos fonte previamente editados ou mesmo salvar as produções atuais, além de efetuar mudanças no layout que podem facilitar o ambiente de trabalho.

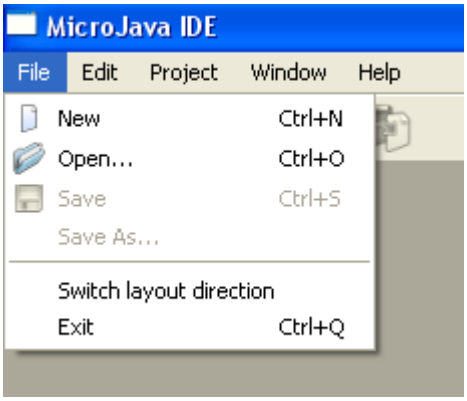

Figura 29 - Menu File do MIDE.

Deve se perceber que os atalhos para os menus estão explicitados à frente de cada uma das opções, ou seja, para que um novo código fonte seja editado na área de trabalho a opção *New* pode ser invocada através do atalho de teclado *"Ctrl+N".*

O menu Edit na Figura 30 contém as operações básicas de edição, onde podem ser realizados cópias, recortes, cola e restauração de fragmentos de códigos, tarefas utilizadas geralmente bastante utilizadas.

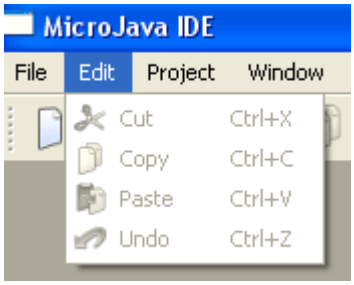

Figura 30 - Menu Edit do MIDE.

O menu Project na Figura 31 possui as duas funções fundamentas na MIDE, a opção de compilação e execução que são disparadas automaticamente através dos menus, ou mais comumente pelos atalhos de teclados.

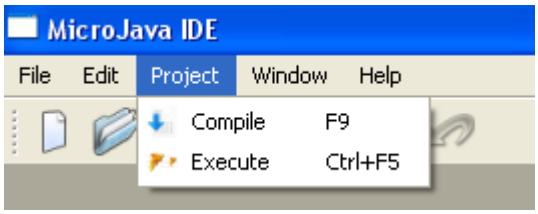

Figura 31 - Menu Project do MIDE.

O menu *Window* mostrado na Figura 32 oferece opções de distribuição das janelas de trabalho de forma que múltiplos códigos fontes possas ser mais bem visualizados e editados paralelamente.

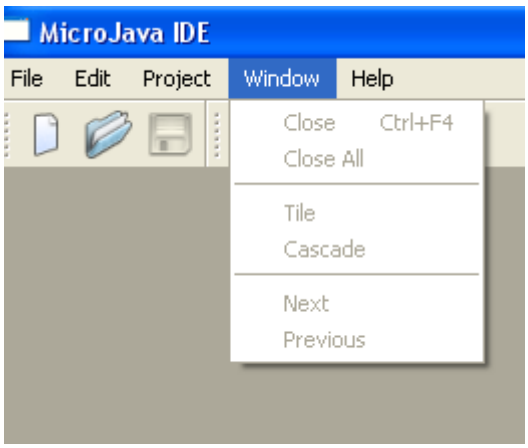

Figura 32 - Menu Window do MIDE.

O menu *Help* na Figura 33 disponibiliza informações à respeito dos desenvolvedores do sistemas através da opção *About*.

Figura 33 - Menu Help do MIDE.

5.2 Sistema Highlight

Conhecendo os menus abordemos agora o sistema de *highlight* que realiza um reconhecimento sintático da linguagem MicroJava a fim de colorir adequadamente as palavras chaves pertencentes a linguagem, o que permite uma compreensão de código mais rapida por parte dos desenvolvedores.

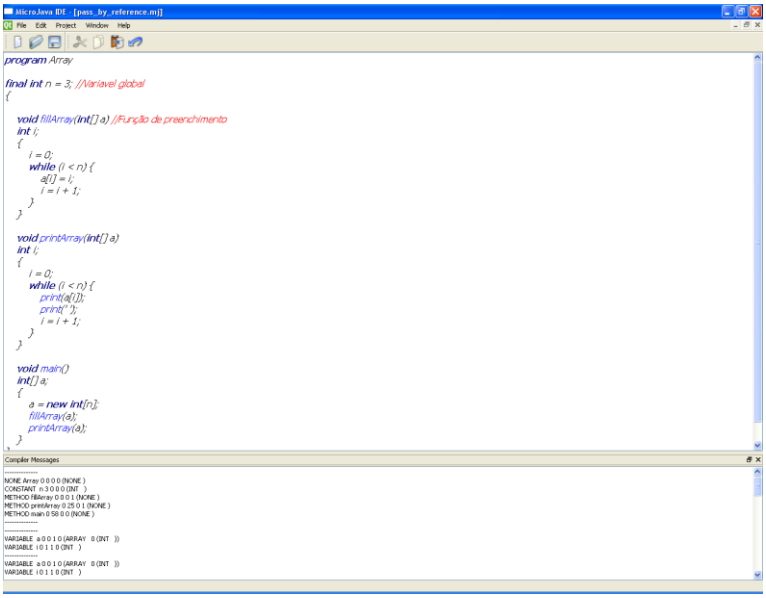

Figura 34 – Sistema highlight.

A Figura 34 mostra o um projeto que explora as funcionalidades de passagem por referência do MicroJava, seu código está sendo analisado e colorido adequadamente levando em conta a sintaxe da linguagem. Os comentários são destacado em vermelho, os métodos são destacados de cor azul clara e as demais palavras chaves de azul escuro.

5.3 Desenvolvendo com o MIDE

Como o MIDE deve ser o mais amigavel possível ao desenvolvedor ele proporciona algumas ferramentas que facilitam a busca por erros sintáticos. Por exemplo, na Figura 35 foi desenvolvido propositalmente um código fonte que não respeita sintáticamente a linguagem MicroJava, se o desenvolvedor disparar o processo de compilação com *"Ctrl + F9"* a janela inferior denominada *Compiler Messages* irá mostrar os possíveis problemas indicando a linha e coluna onde estes erros foram identificados. Com o cursor do mouse o desenvolvedor pode achar esta linha rápidamente visualizando sua posição no *toolbar* contido na parte inferior da janela principal.

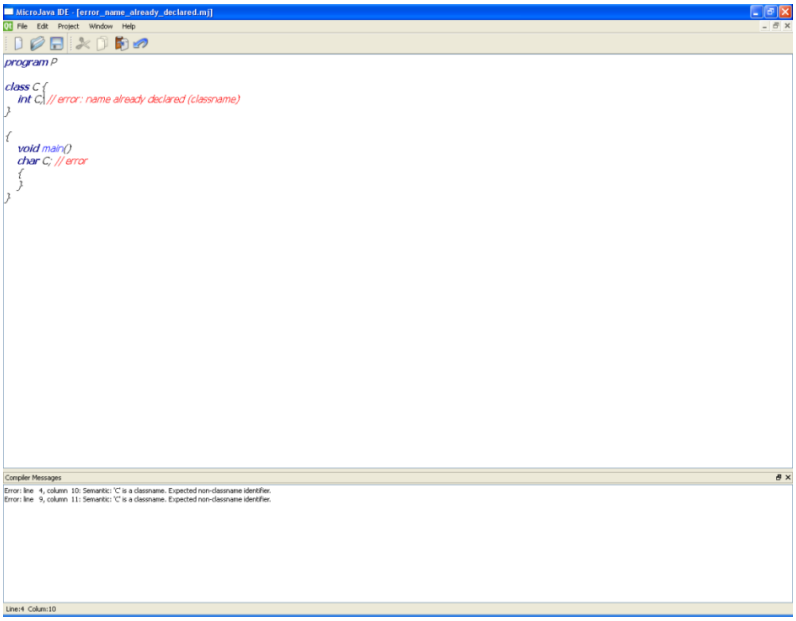

Figura 35 - Janela Compiler Messages e encontro de erros sintáticos.

O MIDE ainda proporciona um ambiente de trabalho multi-projeto onde são possíveis abrir simultaneamente um número ilimitado de projetos e editá-los paralelamente, podendo realizar a compilação e execução de cada uma delas a qualquer instante. A Figura 36 mostra dois projetos que implementam os algoritmos de ordenação *Bubble Sort* e *Insertion Sort* em paralelo.

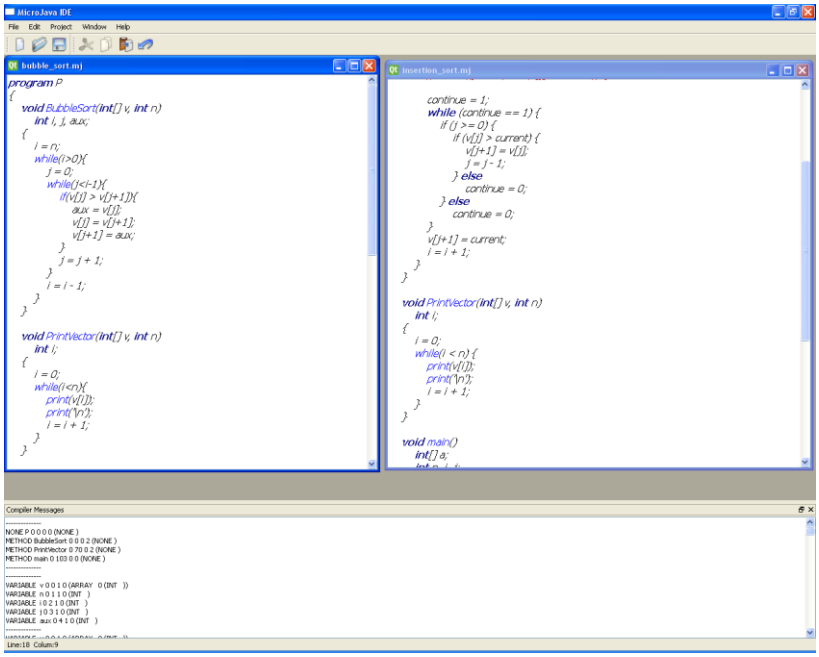

Figura 36 - Ambiente de trabalho multi-projeto.

Com a ativação da janela do projeto Bubble Sort e o disparo da compilação e execução podemos efetivamente visualizar a execução do programa como mostrado na Figura 37.

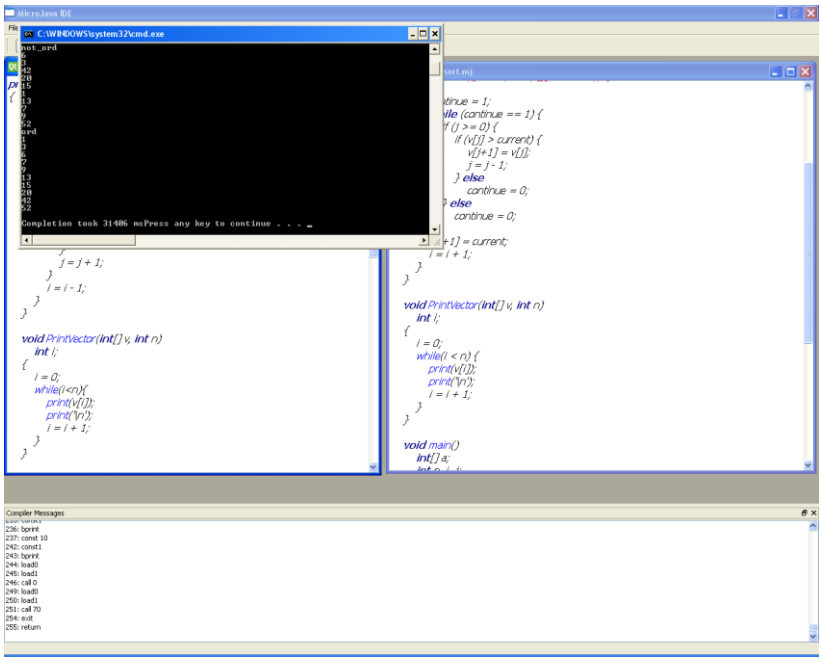

Figura 37 - Execução do Bubble Sort.

A execução do programa é realizada externamente ao MIDE em um console separado onde o usuário pode ter maior liberdade e contar com a infra-estrutura do sistema.

6.0 Conclusão

O desenvolvimento do compilador para a linguagem MicroJava e do ambiente de desenvolvimento MIDE foram de extrema importância para uma compreensão total dos fundamentos que orientam a teoria dos compiladores, já que grande parte dos métodos e algoritmos discutidos em sala de aula foram executados e postos à prova no decorrer deste trabalho. O compilador foi implementado de forma integral e testado exaustivamente com o apoio do MIDE, obtendo sucesso na avaliação e geração de código para todas as entradas testadas, o que garantiu uma implementação estável.