

## **Relatório do desenvolvimento de um simulador de máquina Norma na linguagem Java**

**Integrantes:** Léo Henrique Eifert (101614), Gabriel Longhi Quadro (101586)

### **Objetivo**

Visamos ampliar nosso entendimento sobre máquinas, computadores e programas universais através do desenvolvimento do simulador Norma.

### **Introdução**

A dupla optou em realizar o simulador na linguagem Java utilizando a IDE Eclipse. Para o desenvolvimento da interface foi usado JGoodies.

O simulador aceita como entrada um programa monolítico com regras rotuladas e gera uma execução completa do programa.

### **Programa 1 – Executor de programas monolíticos com instruções rotuladas**

A imagem a seguir mostra a interface do programa ao iniciar.

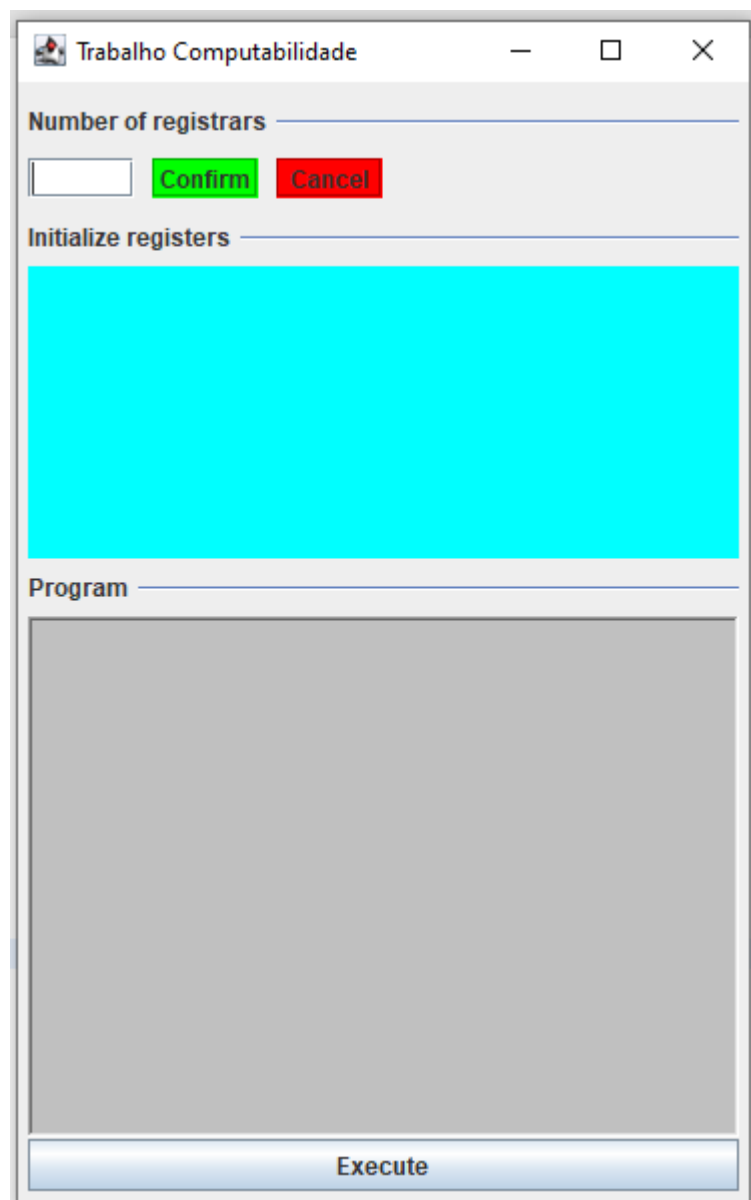


Figura 1: Interface do programa

Para executar o programa, primeiro é necessário informar a quantidade de registradores que serão utilizados (para este trabalho o número de registradores está limitado a 4), após clique em confirm. Em seguida, inicialize-os com seus nomes e valores. Por fim, adicione um programa no campo de texto "Program" e clique no botão "Execute".

## Simulação

→ Registradores:

“a” e “b”, com valores iniciais “2” e “2”

→ Programa:

1: se zero\_b então vá\_para 5 senão vá\_para 2

2: faça ad\_a vá\_para 3

3: faça ad\_a vá\_para 4

4: faça sub\_b vá\_para 1

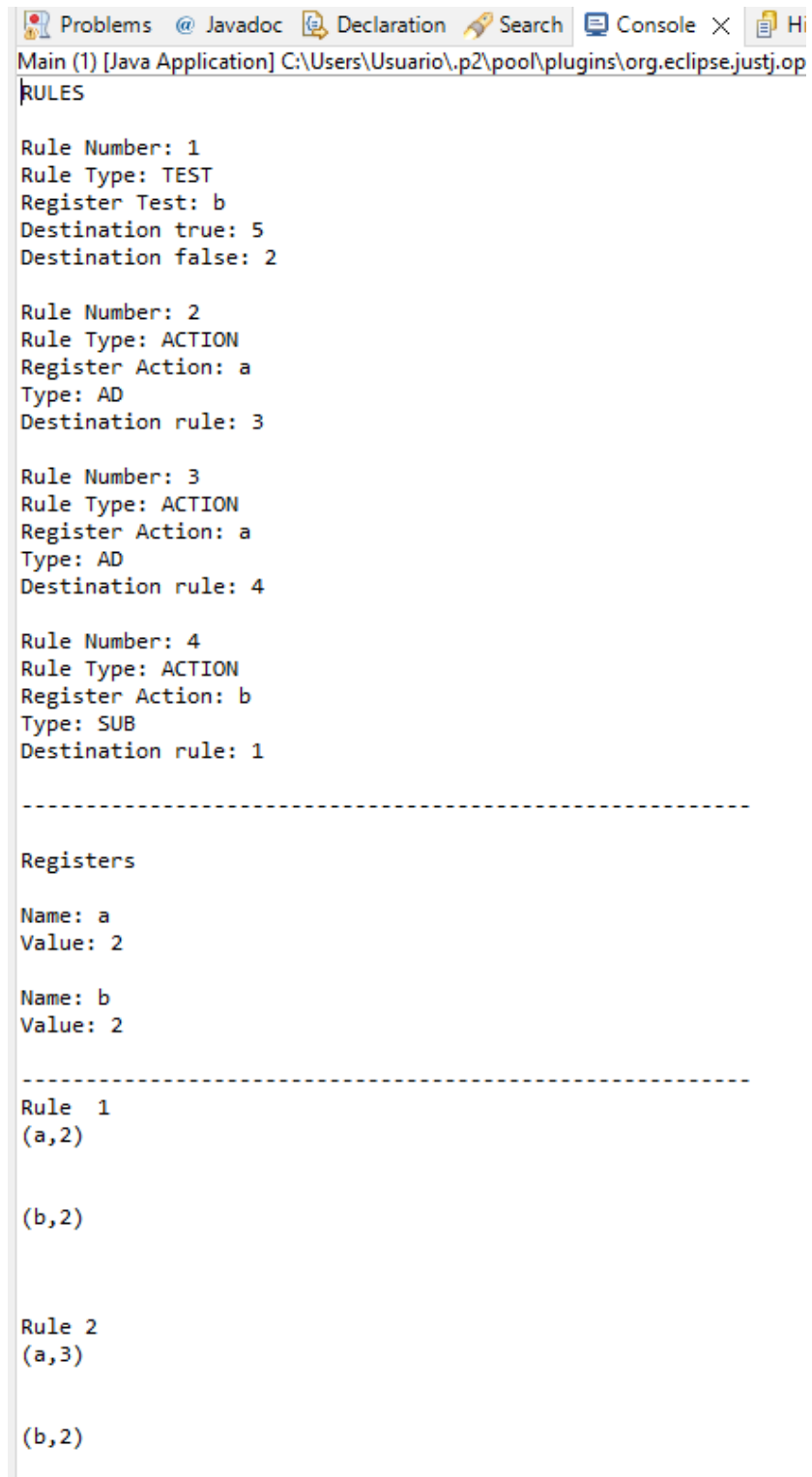
The screenshot shows a window titled "Trabalho Computabilidade" with a standard Windows title bar (minimize, maximize, close buttons). The window is divided into several sections:

- Number of registrars:** A text input field contains the number "2". To its right are two buttons: a green "Confirm" button and a red "Cancel" button.
- Initialize registers:** This section has a light blue background and contains two rows of input fields:
  - Register 1 Name:  Value:
  - Register 2 Name:  Value:
- Program:** A large text area containing the following assembly-like code:

```
1: se zero_b então vá_para 5 senão vá_para 2
2: faça ad_a vá_para 3
3: faça ad_a vá_para 4
4: faça sub_b vá_para 1
```
- Execute:** A large blue button at the bottom of the window.

Figura 2: Simulação

A saída é apresentada no console da aplicação mostrando a computação finita



```
Problems @ Javadoc Declaration Search Console X Hi
Main (1) [Java Application] C:\Users\Usuario\.p2\pool\plugins\org.eclipse.justj.op
RULES

Rule Number: 1
Rule Type: TEST
Register Test: b
Destination true: 5
Destination false: 2

Rule Number: 2
Rule Type: ACTION
Register Action: a
Type: AD
Destination rule: 3

Rule Number: 3
Rule Type: ACTION
Register Action: a
Type: AD
Destination rule: 4

Rule Number: 4
Rule Type: ACTION
Register Action: b
Type: SUB
Destination rule: 1

-----

Registers

Name: a
Value: 2

Name: b
Value: 2

-----

Rule 1
(a,2)

(b,2)

Rule 2
(a,3)

(b,2)
```

```
Rule 3
(a,4)

(b,2)

Rule 4
(a,4)

(b,1)

Rule 1
(a,4)

(b,1)

Rule 2
(a,5)

(b,1)

Rule 3
(a,6)

(b,1)

Rule 4
(a,6)

(b,0)

Rule 1
(a,6)

(b,0)

END
```

Figura 3: Log com a computação completa do programa.

Caso não seja informado o número de registradores é mostrada a seguinte mensagem:

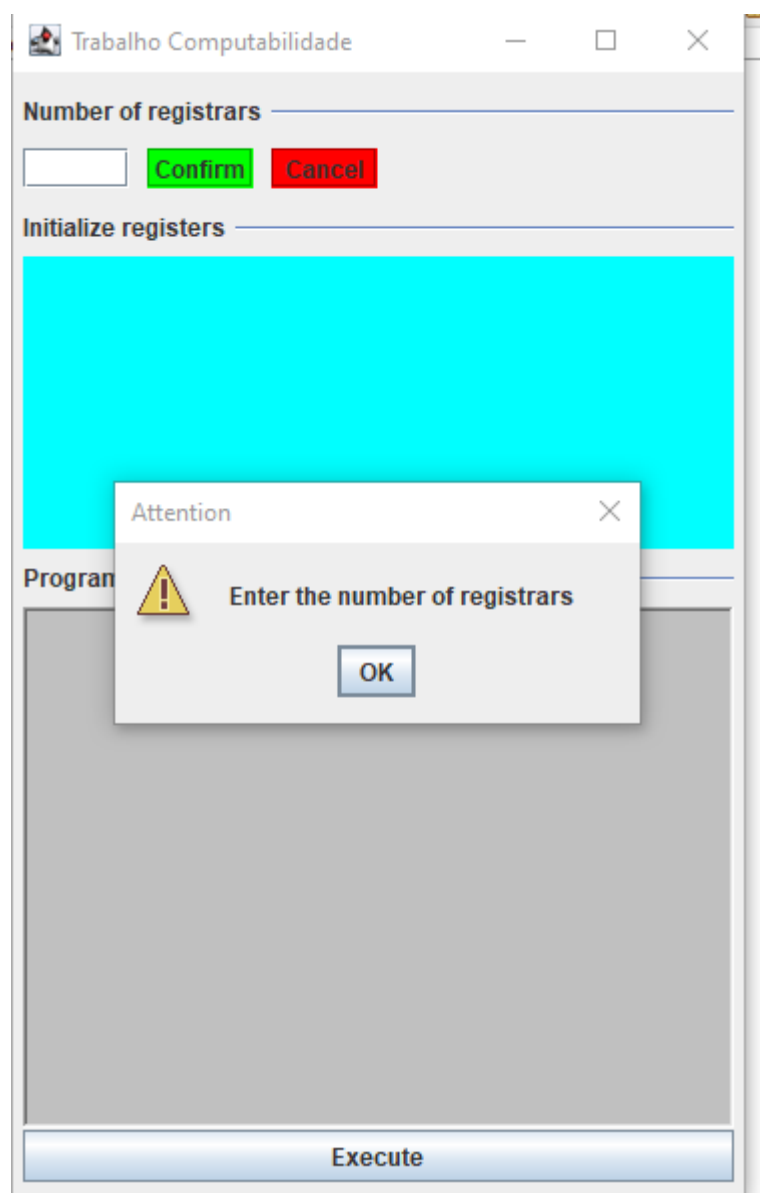


Figura 4: Número de registradores não informado

Caso o usuário digite um número de registradores maior que 4 o programa mostra a seguinte mensagem:

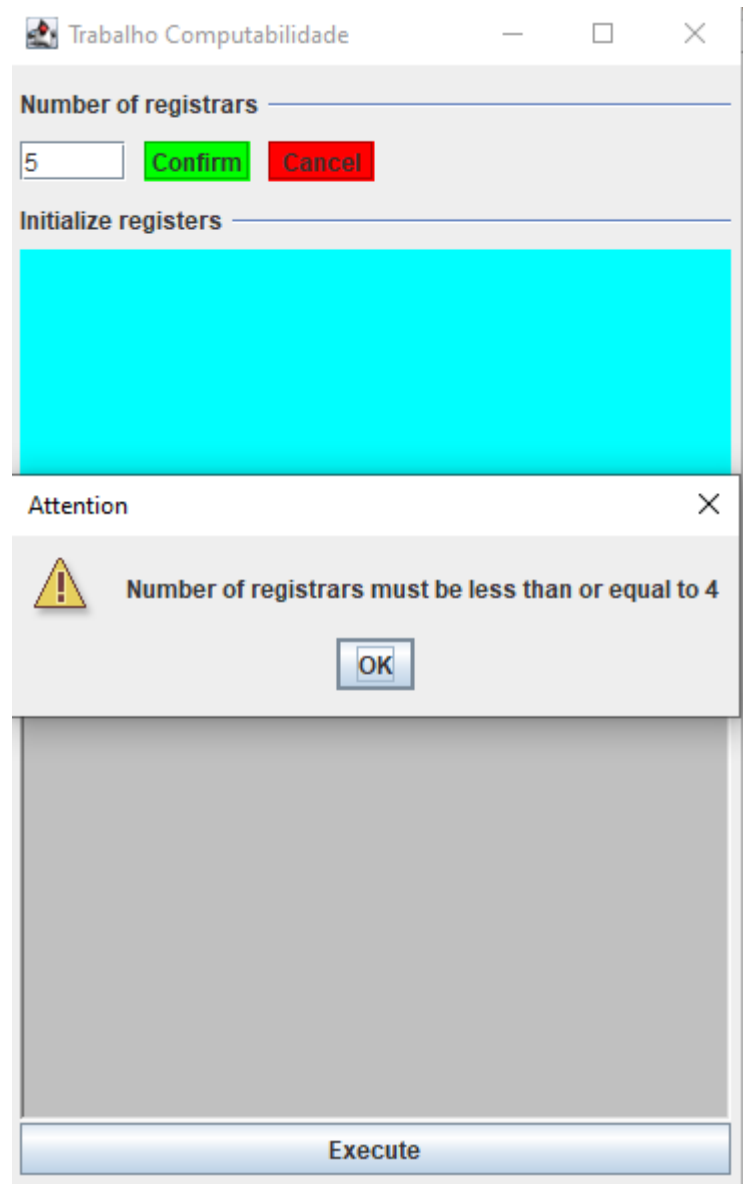


Figura 5: Número de registradores superior a 4.

As estruturas de dados desenvolvidas para a implementação deste programa foram as seguintes:

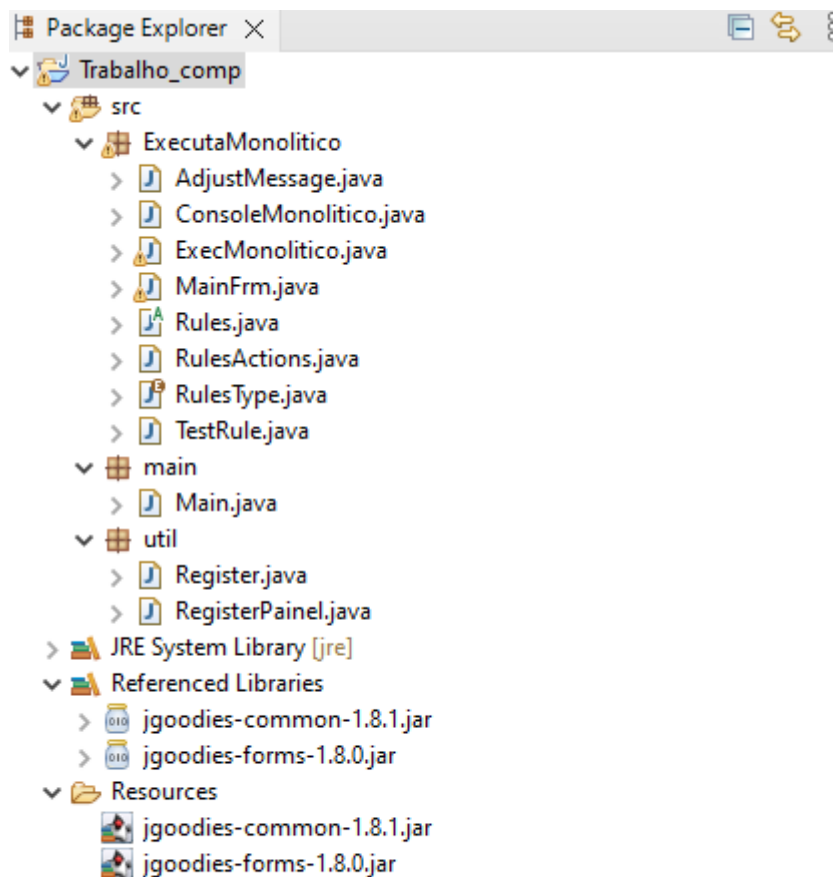


Figura 6: Estrutura do programa

Main: Classe responsável por inicializar o programa. Ela exibe em tela o MainFrm.

MainFrm: Tela principal do programa com a interface gráfica.

ExecMonolitico: Classe responsável por executar as instruções, realizando toda a computação do programa.

RulesActions: Classe responsável por armazenar os dados de uma instrução do tipo operação.

Rule: Classe abstrata criada para implementar as características básicas de uma instrução.

TestRule: Classe responsável por armazenar os dados de uma instrução do tipo teste.

ConsoleMonolitico: Classe responsável por gerar a exibição da computação completa do programa.

RulesType: Classe do tipo Enum que contém os tipos de instrução: TEST e ACTION.

Register: Classe responsável por armazenar os dados de um registrador.

RegisterPainel: Adiciona os componentes para adicionar nome e valor do registrador.



AdjustMessage: Classe responsável por “traduzir” os inputs da interface em dados concretos, gerando as instruções e registradores.

## Lógica de funcionamento

Ao informar o número de registradores e clicar em confirm, o programa irá adicionar os registradores no painel para ser informado nome e valor de cada um a partir da função loadRegisterList().

```
// adiciona os registradores
private void loadRegisterList() {
    // verifica se digitou algo
    if (!"".equals(numReg.getText())) {
        // limite de 4 registradores
        if (Integer.valueOf(numReg.getText()) >= 4) {
            loadRegOk = false;
            JOptionPane.showMessageDialog(form, "Number of registrars must be less than or equal to 4", "Attention",
                JOptionPane.WARNING_MESSAGE);
        } else {
            // converte pra int
            Integer numRegInt = Integer.parseInt(numReg.getText());

            // cria o painel para os registradores
            registrars = new JPanel();
            registrars.setBackground(Color.cyan);
            registrars.setLayout(new BoxLayout(registrars, BoxLayout.Y_AXIS));
            registrars.add(new JLabel(" "));

            // adiciona os registradores
            for (int i = 0; i < numRegInt; i++) {
                registrars.add(new RegisterPainel((i + 1)));
            }
            JPanel jpanel2 = new JPanel();
            jpanel2.setLayout(new BorderLayout());
            jpanel2.setBackground(Color.cyan);
            jpanel2.add(registrars, BorderLayout.CENTER);
            registerPainel.add(jpanel2, BorderLayout.NORTH);

            // havia valor em num de registradores deu tudo ok
            loadRegOk = true;
        }
    } else {
        // não digitou nada
        loadRegOk = false;
        JOptionPane.showMessageDialog(form, "Enter the number of registrars", "Attention",
            JOptionPane.WARNING_MESSAGE);
    }
    SwingUtilities.updateComponentTreeUI(form);
}
```

Figura 07: função loadRegisterList().

Informado os registradores e o programa, ao clicar em execute, o programa irá gerar a lista dos registradores com seu nome e valor a partir da função generateRegisters().

```

// add registradores
private List<Register> generateRegisters() {
    List<Register> result = new ArrayList<Register>();
    for (int i = 1; i < registrars.getComponentCount(); i++) {
        Register r = new Register();
        // add os registers no painel
        RegisterPainel regiterPainel = (RegisterPainel) registrars.getComponent(i);
        // seta nome
        r.setNameReg(regiterPainel.getNmRegistrador().getText());
        // seta valor
        r.setValueRegister(new BigInteger(regiterPainel.getVlRegistrador().getText()));

        result.add(r);
    }
    return result;
}

```

Figura 8: função generateRegisters().

Após se dá início ao tratamento da leitura do programa para armazenar as regras numa lista de rules. Sendo assim é lido cada linha do programa, sendo tratada em generaActions() passando a regra da linha lida. A regra lida é verificada a partir de seu tipo quebrando a string em 2 partes, antes do “:” e após, sendo antes o seu número e após o que é pra ser feito. Tendo o tipo de regra, é feito um switch para teste ou ação para assim recolher os dados de destino, qual registrador afetado etc.

```

// gera as regras
public List<Rules> generateActions(String text) {
    List<Rules> rules = new ArrayList<Rules>();
    // le linha por linha
    List<String> lines = Arrays.asList(text.split("\n"));
    // reúne as regras
    for (String rule : lines) {
        rules.add(generaeActions(rule));
    }
    return rules;
}

private Rules generaeActions(String instrucao) {
    String[] split = instrucao.split(":");
    RulesType ruleType = indentifieRuleTye(split[1]);

    Rules result = null;

    // verifica se é teste ou ação
    switch (ruleType) {
        case ACTION:
            // cria ação
            result = createAction(split);
            break;
        case TEST:
            // cria teste
            result = createTeste(split);
            break;
        default:
    }
    return result;
}

```

Figura 9: Tratamento da mensagem.

A identificação do tipo de regra se dá a partir da quebra da string regra, caso comece com “se”

trata-se de uma regra de test senão é ação.

```
// identifica o tipo da regra
private RulesType indentifieRuleTye(String rule) {
    return "se".equals(rule.substring(1, 3)) ? RulesType.TEST : RulesType.ACTION;
}
```

Figura 10: verifica tipo regra.

Caso a regra trata-se de um teste, será executada a função createTeste(rule), onde a primeira parte da string recebida será o identificador da regra, e a segunda parte o que ela faz. TestRule é a classe responsável por armazenar as informações da regra de tipo teste, numberRule será o número da regra, RulerType o tipo nesse caso teste, variavel r armazena o registrador afetado , rulerTrue indica o destino caso a regra seja true e rulerFalse caso a regra seja false.

```
// caso criar teste
private Rules createTeste(String[] rule) {
    // criação da regra
    Integer numberRule = Integer.valueOf(rule[0]);
    // adiciona o se register zero
    Integer start = rule[1].indexOf("zero_");
    // add o então
    Integer end = rule[1].indexOf("então");
    String r = rule[1].substring(start + 5, end).trim();

    // adiona parte da regra da ocasião vá para
    String subString[] = rule[1].split("vá para ");
    // adiciona o senão do vá para
    Integer end2 = subString[1].indexOf("senão");
    // caso true
    Integer ruleTrue = Integer.valueOf(subString[1].substring(0, end2).trim());
    // caso false
    Integer ruleFalse = Integer.valueOf(subString[2].trim());

    // teste regra
    TestRule testRule = new TestRule();
    // seta numero da regra
    testRule.setNumberRules(numberRule);
    // seta tip pra teste
    testRule.setRulesType(RulesType.TEST);

    testRule.setRegTeste(r);
    // true
    testRule.setNumberRuleTrue(ruleTrue);
    // false
    testRule.setNumberRuleFalse(ruleFalse);
    // retorna o teste da regra
    return testRule;
}
```

Figura 11: organiza caso Test

Caso a regra seja do tipo ação, primeiro pegamos o número da regra igual em teste. Quebramos a sting em “\_” e pegamos o que vem antes para saber o tipo de operação e verificamos se é ad ou sub. Após fazermos o substign para saber o desino e também veificamos qual registrador será afetado, armazenando isso em RulesActions.

```

// caso criar ação
private Rules createAction(String[] rule) {
    // pega número
    Integer numberAction = Integer.valueOf(rule[0]);

    Integer index1 = rule[1].indexOf("_");
    // tipo
    String typeRule = rule[1].substring(index1 - 3, index1).trim();
    // pega se é ad ou sub
    RulesTypeActions operationRuleType = typeRule.equals(RulesTypeActions.AD.getValue()) ? RulesTypeActions.AD
        : RulesTypeActions.SUB;

    Integer index2 = rule[1].indexOf("vá_para");
    // informa onde deve ir
    Integer instrucaoDestino = Integer.valueOf(rule[1].substring(index2 + 7).trim());
    // destino
    String regDestiny = rule[1].substring(index1 + 1, index2).trim();
    // regra ação
    RulesActions result = new RulesActions();
    // número
    result.setNumberRules(numberAction);
    // tipo ação
    result.setRulesType(RulesType.ACTION);
    result.setTypeRuleActions(operationRuleType);
    // destino
    result.setRulesDestination(instrucaoDestino);
    result.setRgOperation(regDestiny);
    return result;
}

```

Figura 12: organiza caso Action

Lido todas as regras do programa, é hora de percorrê-las e atualizar o valor dos registradores em ExecMonolitico na função execMonolitico(). Nela percorremos as regras a partir de um while. Nele registramos a regra que está e verificamos se a regra se trata de um test ou action.

```

public void execMonolitico() {
    //percorre regras
    Rules current = getNumberRule(1);

    // enquanto houver regra
    while (current != null) {
        // registra a regra atual
        logConsole.registerCurrentRule(current);
        Integer next = 0; // proximo
        // verifica tipo
        switch (current.getRulesType()) {
            // executa ação
            case ACTION:
                next = doAction((RulesActions) current);
                break;
            // executa teste
            case TEST:
                next = doTest((TestRule) current);
                break;
            default:
        }

        // seta registrador e valor
        logConsole.registerRegValues(registers);
        // pega próxima
        current = getNumberRule(next);
    }

    // faz o registro de fim de programa
    logConsole.endProgram();

    // programa rodou e acabou
    programRun = true;
}

```

Figura 13: Percorrendo as regras.

Caso a regra seja do tipo test realiza-se a função doTest. Pegamos qual registrar que fazer o

teste e verificar se se é zero ou não e retornamos o número da próxima regra que será feita em `execMonolitico()`.

```
// case test
private Integer doTest(TestRule rule) {
    String numberReg = rule.getRegTeste();
    Register register = registers.stream().filter(f -> f.getNameReg().equals(numberReg)).findFirst().get();
    return register.getValueReg().equals(BigInteger.ZERO) ? rule.getNumberRuleTrue()
        : rule.getNrInstrucaoFalso();
}
```

Figura 14: Caso regra seja teste.

Caso a regra seja do tipo ação será feita a função `doAction()`, a qual pega o registrador, pega o tipo de ação( ad ou sub) , e pega o valor do registrador. É realizado um switch para o tipo da ação, caso ad adiciona +1 no valor do registrador e caso sub -1.

```
// case ação
private Integer doAction(RulesActions current) {
    //operação
    String nmReg = current.getRgOperation();
    //registrador
    Register reg = registers.stream().filter(f -> f.getNameReg().equals(nmReg)).findFirst().get();
    //tipo regra
    RulesTypeActions rulesTypeAction = current.getTypeRuleActions();
    BigInteger vlReg = BigInteger.ZERO;
    switch (rulesTypeAction) {
        // caso ad
        case AD:
            //pega valor do registrador
            vlReg = reg.getValueReg().add(BigInteger.ONE);
            break;
        // caso sub
        case SUB:
            //pega valor do registrador
            vlReg = reg.getValueReg().subtract(BigInteger.ONE);
            break;
        default:
    }

    //seta o valor ou de ad ou de sub
    reg.setValueRegister(vlReg);

    //atualiza atual
    return current.getRulesDestination();
}
```

Figura 15: Caso regra seja action.

Após percorrer todas as regras é mostrado no console o passo a passo da execução do programa com os valores finais dos registradores conforme a figura 3 a partir da classe `ConsoleMonolitico`.

```

// gera o que será impresso no console
public void registerRule(List<Rules> rules) {
    log.append("RULES\n");
    log.append("\n");

    // adiciona as regras
    for (Rules r : rules) {
        // registra nome e tipo da regra
        log.append("Rule Number: " + r.getNumberRules() + "\n");
        log.append("Rule Type: " + r.getRulesType() + "\n");

        // regra ação
        if (r instanceof RulesActions) {
            RulesActions instrucaoOperacao = (RulesActions) r;
            log.append("Register Action: " + instrucaoOperacao.getRgOperation() + "\n");
            log.append("Type: " + instrucaoOperacao.getTypeRuleActions() + "\n");
            log.append("Destination rule: " + instrucaoOperacao.getRulesDestination() + "\n");
        } else { // regra teste
            TestRule instrucaoOperacao = (TestRule) r;
            log.append("Register Test: " + instrucaoOperacao.getRegTeste() + "\n");
            log.append("Destination true: " + instrucaoOperacao.getNumberRuleTrue() + "\n");
            log.append("Destination false: " + instrucaoOperacao.getNrInstrucaoFalso() + "\n");
        }

        log.append("\n");
    }
}

public void printRegisters(List<Register> registers) {
    log.append("-----\n");

    log.append("\n\nRegisters\n\n");

    // print os registradores
    for (Register registrador : registers) {
        // nome registrador
        log.append("Name: " + registrador.getNameReg() + "\n");
        // valor
        log.append("Value: " + registrador.getValueReg() + "\n");

        log.append("\n");
    }
    log.append("-----\n");
}

// registra valores registradores
public void registerRegValues(List<Register> registers) {
    log.append("-----\n");
    //
    for (Register register : registers) {
        log.append("(" + register.getNameReg() + ", " + register.getValueReg()

```

Figura 16: Classe ConsoleMonolitico.