UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

AMBIENTE PARA AUXILIAR O DESENVOLVIMENTO DE PROGRAMAS MONOLÍTICOS

OLIVER MÁRIO DA SILVA

BLUMENAU
2004

2004/2-39

OLIVER MÁRIO DA SILVA

AMBIENTE PARA AUXILIAR O DESENVOLVIMENTO DE PROGRAMAS MONOLÍTICOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva - Orientador

BLUMENAU
2004

2004/2-39

AMBIENTE PARA AUXILIAR O DESENVOLVIMENTO DE PROGRAMAS MONOLÍTICOS

Por

OLIVER MÁRIO DA SILVA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. José Roque Voltolini da Silva – Orientador, FURB
Membro:	Prof. Joyce Martins, FURB
Membro:	Prof. Dr. Mauro Marcelo Mattos, FURB

Blumenau, 15 de dezembro de 2004.

Não me preocupa que não exerça um cargo, o que me preocupa é como me tornar capaz de um. Não me preocupa o não ser conhecido, mas procuro tornar-me digno de ser conhecido.

Confúcio

Dedico este trabalho a minha esposa Cristina, que me apoiou nos momentos difíceis e compreendeu minha ausência durante todo o tempo dedicado à conclusão deste curso.

AGRADECIMENTOS

A Deus, pelo dom da vida e pela saúde que me concede a cada dia.

Aos meus pais Wilson e Marlene que sempre procuraram me educar da melhor maneira possível, com muito amor e carinho.

As minhas irmãs Poliana e Daniela, que sempre estiveram ao meu lado.

A todos os amigos, que me ajudaram e apoiaram durante todo este curso.

Ao meu orientador, José R. V. da Silva, pelo auxílio prestado durante a elaboração do trabalho.

RESUMO

Este trabalho consiste na criação de um ambiente para auxiliar o desenvolvimento de programas monolíticos, utilizando somente registradores que comportem números naturais (mesma estrutura de dados da máquina *Number TheOretic Register MAchine* - NORMA). Como foco principal, tem-se a aplicação de um novo algoritmo proposto por José R. V. da Silva (SILVA, 2004) para a transformação de um programa monolítico descrito na forma de instruções rotuladas em instruções rotuladas compostas. A especificação da linguagem monolítica para o ambiente foi feita utilizando-se a notação *Backus-Naur Form* (BNF). Para a especificação do ambiente utilizou-se a análise orientada a objetos com a *Unified Modeling Language* (UML), através dos diagramas de casos de uso, de classes e de seqüência. A implementação do ambiente foi feita no ambiente de programação Delphi 7.0, utilizando classes criadas pelo Gerador de Analisadores Léxicos e Sintáticos (GALS). Através das especificações da BNF da linguagem, o GALS gerou todas as classes para a análise léxica e sintática.

Palavras chaves: Compiladores; Computabilidade; Programas Monolíticos.

ABSTRACT

This work consists of the creation of an environment to assist the development of monolithic programs, only using registers that hold natural numbers (same data structure of the machine Number TheOretic Register MAchine - NORMA). As main focus, it is had application of a new algorithm considered for Jose R. V. da Silva (SILVA, 2004) for the transformation of a described monolithic program in the form of instructions labeled in labeled instructions composed. The specification of the monolithic language for the environment was made using it notation Backus-Naur Form (BNF). For the specification of the environment it was used the objects oriented analysis with Unified Modeling Language (UML), through the diagrams of cases of use, of class and of sequence. The implementation of the environment was made in the environment of programming Delphi 7,0, using class created for the Analyzer Generator for Lexicon and Syntactic (GALS). Through the specifications of the BNF of the language, the GALS generated all the class for the lexicon and syntactic analysis.

Key Words: Compilers; Computability; Monolithic Programs.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SÍMBOLOS

$\in$ - pertence

$\notin$ - não pertence

$\subseteq$ - contido

ε - epsílon

$\cup$ - união

$\equiv$ - equivalente

$\surd$ - operação vazia

ω – rótulo que identifica uma instrução de ciclo infinito

SUMÁRIO

1 INTRODUÇÃO

Desenvolver programas de computadores livres de erros é algo que envolve alguns fatores. Seguir algum modelo de requisitos e projeto de software com certeza é um fator fortíssimo para se obter um bom resultado. A etapa de implementação de um programa consiste basicamente na construção do código fonte, sua compilação e execução. Durante a implementação, demanda-se um esforço para que se escreva um código sem erros. Mesmo que se esteja utilizando um ambiente de programação avançado, exige-se muito da experiência e conhecimento dos desenvolvedores. Ainda assim os erros são praticamente inevitáveis.

Quando o código fonte de um programa passa pelo compilador, são revelados os erros de sintaxe e alguns de semântica. Se o programa não apresentar nenhum erro detectável, o mesmo é compilado com sucesso e pode ser executado. Os chamados “erros de lógica” (ou semânticos) ainda não foram totalmente solucionados pelos compiladores atuais. Alguns compiladores já conseguem identificar alguns erros simples de lógica como: variáveis não inicializadas, estruturas de repetição mal definidas (somente algumas), entre outros.

Visto o problema descrito, desenvolveu-se um ambiente, que além de detectar os erros mais comuns (léxicos e sintáticos) em programas monolíticos (programas baseados em desvios condicionais e incondicionais), também detecta “erros de lógica” encontrados em programas. Estes erros podem ocorrer na estrutura estática ou dinâmica de um programa. Aqui desenvolveu-se apenas a detecção de alguns “erros de lógica” na estrutura estática, tais como: detecção de ciclos infinitos e instruções mortas (inatingíveis).

O erro conhecido como ciclo infinito acontece em uma estrutura de repetição cuja condição foi mal definida. Esta falha irá fazer com que as instruções da estrutura de repetição sejam executadas infinitamente. Já as instruções mortas são conjuntos de instruções que jamais serão executadas.

Estes dois erros ou propriedades como serão chamados, serão verificados em cima de programas monolíticos, que primeiramente serão transformados em instruções rotuladas compostas, utilizando um algoritmo proposto em Silva (2004).

Ainda, o processo de detecção para os erros mencionados prepara os programas para verificar a sua equivalência com outros. Visto isto, o ambiente também possibilita a realização da verificação da equivalência entre dois programas monolíticos.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho foi o de desenvolver um ambiente para auxiliar o desenvolvimento de programas monolíticos.

Os objetivos específicos são:

a) disponibilizar analisadores léxico e sintático para programas monolíticos;
b) disponibilizar um módulo para conversão de programas monolíticos na forma de instruções rotuladas em programas monolíticos na forma de instruções rotuladas compostas;
c) disponibilizar um módulo para verificar as seguintes propriedades:
   - existência de ciclos infinitos,
   - identificação de instruções mortas;
d) disponibilizar uma opção para verificar a equivalência entre dois programas monolíticos;
e) disponibilizar um módulo para interpretar programas monolíticos.

1.2 ESTRUTURA DO TRABALHO

No presente capítulo é apresentada uma introdução sobre o trabalho, assim como os seus objetivos. O segundo capítulo abrange toda a fundamentação teórica do trabalho, apresentando definições sobre máquina, programas e suas propriedades, computação e função computada, equivalência forte de programas, instruções rotuladas compostas, propriedades identificáveis em um programa monolítico, otimização estrutural de um programa monolítico, máquina NORMA e também um trabalho correlato. No terceiro capítulo é apresentado o desenvolvimento do ambiente, através do levantamento dos requisitos do ambiente, da definição da linguagem e sua especificação através da BNF, da especificação do ambiente usando Orientação a Objetos (OO) com a UML, através dos diagramas de casos de uso, de classes e de seqüência, da implementação e dos resultados e discussões obtidos ao final do trabalho. No quarto e último capítulo são apresentadas as conclusões do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentadas as definições sobre máquina, programas e suas propriedades, computação e função computada, equivalência forte de programas, instruções rotuladas compostas, propriedades identificáveis em um programa monolítico, otimização estrutural de um programa monolítico, máquina NORMA, compiladores, a ferramenta GALS e também um trabalho correlato.

Este capítulo é fortemente baseado no livro de Diverio e Menezes (2003).

2.1 MÁQUINA

Segundo Diverio e Menezes (2003, p. 20-21), o objetivo de uma máquina é suprir todas as informações necessárias para que a computação de um programa possa ser descrita. Cabe a máquina suprir o significado aos identificadores das operações e testes. Os identificadores de operação e teste interpretados pela máquina devem ser associados a uma transformação na estrutura de memória e a uma função verdade, respectivamente. A máquina deve descrever o armazenamento ou recuperação de informações na estrutura de memória.

Uma máquina é uma 7-upla representada por $M = (V, X, Y, \pi_X, \pi_Y, \pi_F, \pi_T)$ (DIVERIO; MENEZES, 2003 p. 21) onde:

a) V : conjunto de valores de memória;
b) X : conjunto de valores de entrada;
c) Y : conjunto de valores de saída;
d) π_X : função de entrada tal que $\pi_X: X \rightarrow V$;
e) π_Y : função de saída tal que $\pi_Y: V \rightarrow Y$;
f) π_F : conjunto de interpretações de operações onde, para cada identificador de operação F interpretado por M, existe uma única função: $\pi_F: V \rightarrow V$ em π_F;
g) π_T : conjunto de interpretação de testes tal que, para cada identificador de teste T interpretado por M, existe uma única função: $\pi_{T:} V \rightarrow$ {verdadeiro, falso} em π_T.

2.2 PROGRAMA

Segundo Diverio e Menezes (2003, p. 10), um programa pode ser descrito como um conjunto estruturado de instruções que capacitam uma máquina a aplicar sucessivamente certas operações básicas e testes em uma parte determinada dos dados iniciais fornecidos, até

que esses dados tenham se transformado numa forma desejável. Um programa deve explicitar como as operações ou teste devem ser compostos, ou seja, deve possuir uma estrutura de controle de operações e testes. Existem três formas de estruturação do controle, as quais serão expostas nos próximos três tópicos, como tipos de programas.

2.2.1 PROGRAMA MONOLÍTICO

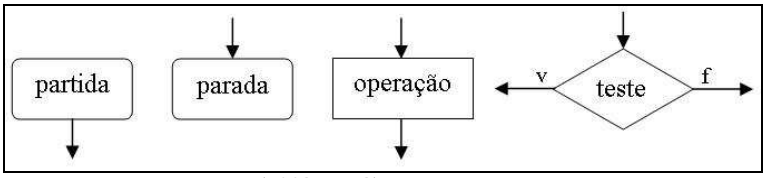
Segundo Diverio e Menezes (2003, p. 12), um programa monolítico é estruturado usando desvios condicionais e incondicionais, não fazendo uso explícito de mecanismos auxiliares de programação que permitam uma melhor estruturação do controle como iteração, subdivisão ou recursão. Um programa monolítico é um par ordenado P = (I, r) onde I é um conjunto finito de instruções rotuladas e r é o rótulo inicial em I. O Quadro 1 apresenta um exemplo de programa monolítico, que tem somente uma instrução, a qual é uma operação vazia representada pelo símbolo √.

$P2 = (\{r_1:$ faça √ vá_para $r_2\},\ r_1)$
onde r_2 é um rótulo final

Fonte: Diverio e Menezes (2003, p. 15)

Quadro 1 - Programa monolítico P2

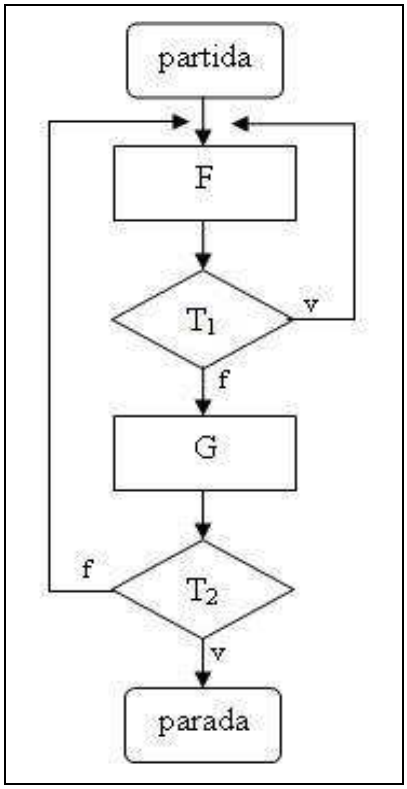
Programas monolíticos podem ser representados de duas formas: através de fluxogramas ou através de instruções rotuladas. Fluxogramas são construídos a partir de componentes elementares denominados *Partida*, *Parada*, operação e teste (Figura 1) (DIVERIO; MENEZES, 2003, p. 12).



Fonte: Diverio e Menezes (2003, p. 13)

Figura 1 - Componentes elementares de um fluxograma

Na Figura 2 tem-se o exemplo de um programa monolítico representado através de fluxograma.



Fonte: Diverio e Menezes (2003, p. 14)
Figura 2 - Fluxograma

Segundo Diverio e Menezes (2003, p. 14), a definição formal de programas monolíticos é melhor descrita usando a notação de instruções rotuladas do que diagramas. No Quadro 2 tem-se um programa monolítico escrito em forma de instruções rotuladas, o qual é equivalente ao descrito na Figura 2.

```
1: faça F vá_para 2
2: se T1 então vá_para 1 senão vá_para 3
3: faça G vá_para 4
4: se T2 então vá_para 5 senão vá_para 1
```

Fonte: Diverio e Menezes (2003, p. 14)
Quadro 2 - Instruções rotuladas

Na instrução rotulada 4, existe um desvio para o rótulo 5. Porem o rótulo 5 não existe no conjunto de instruções rotuladas. Sempre que houver um desvio para um rótulo que não exista no conjunto de instruções rotuladas, subentende-se que é um rótulo final. (DIVERIO; MENEZES, 2003, p. 13).

2.2.2 PROGRAMA ITERATIVO

Segundo Diverio e Menezes (2003, p. 15-16), a noção de programa com estruturas de controle iterativas surgiu na tentativa de solucionar os problemas decorrentes da dificuldade

de entendimento e manutenção de programas monolíticos, onde existe uma grande liberdade para definir desvios incondicionais, ocasionando as “quebras de lógica”. A idéia é substituir os desvios incondicionais por estruturas de controle de repetição, o que resulta em uma melhor estruturação dos desvios.

Programas iterativos possuem mecanismos de controle de iteração de trechos do programa, não permitindo o uso de desvios incondicionais. No Quadro 3 tem-se um exemplo de programa iterativo, onde T_1, T_2 e T_3 são identificadores de teste e F e G são operações.

```
(se T1
então enquanto T2
      faça  (até T3
            faça (F;G))
senão √)
```

Fonte: adaptado de Diverio e Menezes (2003, p. 18)

Quadro 3 - Programa iterativo

Um programa iterativo também pode ser representado através de diagramas de Nassi-Schneidermann (NASSI; SCHNEIDERMANN, 1973). A Figura 3 apresenta o programa iterativo representado no Quadro 3, através do diagrama de Nassi-Schneiderman.

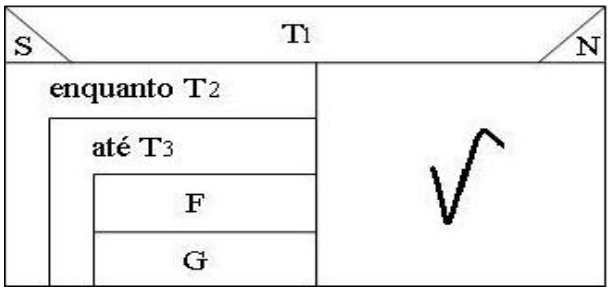


Figura 3 - Programa iterativo representado através do diagrama de Nassi-Schneiderman

2.2.3 PROGRAMA RECURSIVO

Segundo Diverio e Menezes (2003, p. 19), um programa recursivo P tem a forma: P é E_0 onde R_1 def E_1 (R_1 é uma sub-rotina definida pela expressão de sub-rotina E_1), R_2 def E_2, ... R_n def E_n; onde (supondo que $K \in \{1,2,...,n\}$):

a) E_0 é a expressão inicial, a qual é uma expressão de sub-rotinas;
b) E_k é a expressão que define R_k, ou seja, a expressão que define a sub-rotina identificada por R_k.

Para cada identificador de sub-rotina referenciado em alguma expressão, existe uma expressão que o define. A operação vazia √ também é uma expressão de sub-rotina e pode constituir um programa recursivo.

A recursão é uma forma indutiva de definir programas. Possui mecanismos de estruturação em sub-rotinas recursivas. Não possibilita o uso de desvios incondicionais. (DIVERIO; MENEZES, 2003, p. 18) No Quadro 4 tem-se um exemplo de programa recursivo, onde R e S são sub-rotinas.

```
P é R;S onde
      R def F;(se T então G;S),
      S def (se T então √ senão F;R)
```

Fonte: Diverio e Menezes (2003, p. 20)

Quadro 4 - Programa recursivo

2.3 COMPUTAÇÃO E FUNÇÃO COMPUTADA

Segundo Diverio e Menezes (2003, p. 23), a computação de um programa monolítico é um histórico das instruções executadas e o correspondente valor de memória. O histórico é representado na forma de uma cadeia de pares, onde:

a) cada par reflete um estado da máquina para o programa, ou seja, a instrução a ser executada e o valor corrente da memória;
b) a cadeia reflete uma seqüência de estados possíveis a partir do estado inicial.

A função computada por um programa monolítico sobre uma máquina corresponde a uma noção intuitiva (DIVERIO; MENEZES, 2003, p. 29), ou seja:

a) a computação inicia na instrução identificada pelo rótulo inicial com a memória contendo o valor inicial resultante da aplicação da função de entrada sobre o dado fornecido;
b) executa-se passo a passo, testes e operações, na ordem definida pelo programa, até atingir o rótulo final, onde o programa pára;
c) o valor da função computada pelo programa é o valor resultante da aplicação da função de saída ao valor da memória quando o programa atinge o rótulo final. Se um programa não atingir um rótulo final, ele entrará num estado de ciclo infinito ou seja, ele jamais irá terminar e a função computada é indefinida.

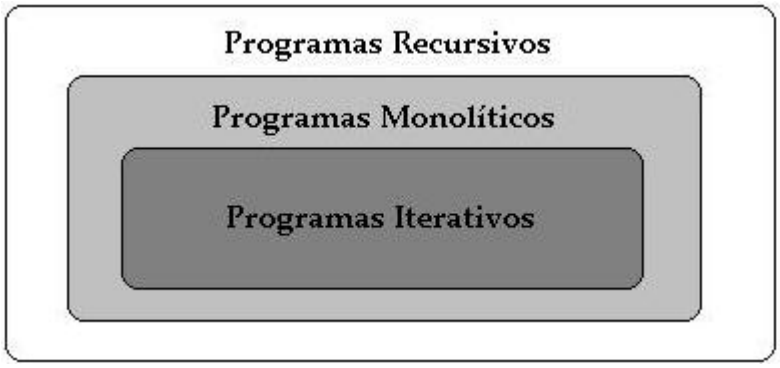
2.4 EQUIVALÊNCIA FORTE DE PROGRAMAS

Segundo Diverio e Menezes (2003, p. 31-32), um par de programas pertence à relação de equivalência se as correspondentes funções computadas coincidem para qualquer máquina.

Considerar a relação de equivalência forte de programas é importante por várias razões:

a) permite identificar diferentes programas em uma mesma classe de equivalência, ou seja, permite identificar diferentes programas cujas funções computadas coincidem para qualquer máquina;
b) as funções computadas por programas fortemente equivalentes têm a propriedade de que as mesmas operações são efetuadas na mesma ordem, independente do significado dos mesmos;
c) possibilita analisar a complexidade estrutural de programas.

Conforme ilustrado na Figura 4, todo programa iterativo pode ser transformado em monolítico, assim como todo monolítico pode ser transformado em recursivo, para qualquer máquina. O inverso não necessariamente é verdadeiro, pois existe uma hierarquia de classes de programas.



Fonte: Diverio e Menezes (2003, p. 35)

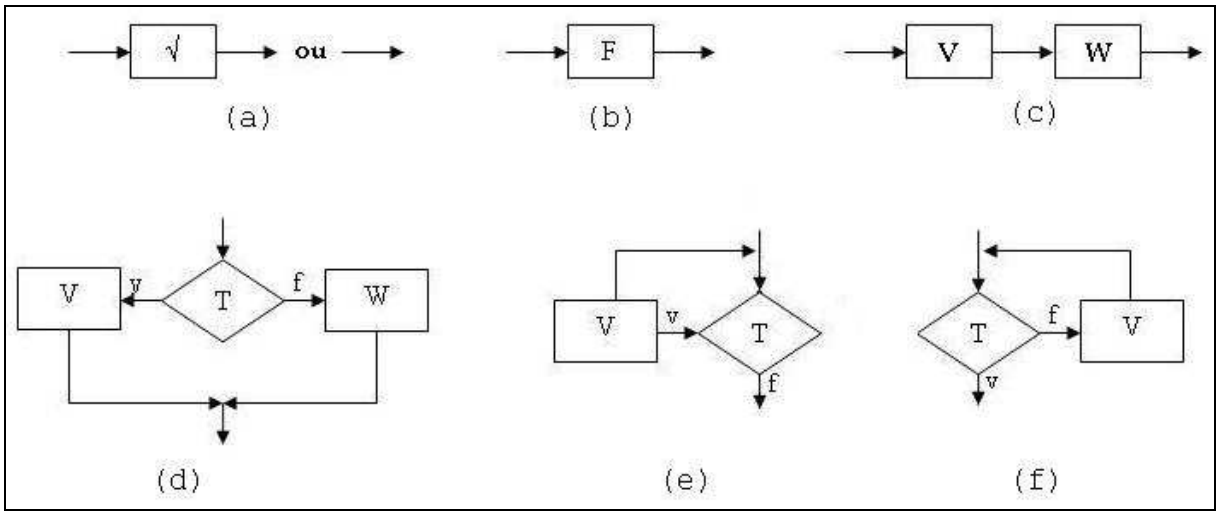
Figura 4 - Hierarquia induzida pela relação equivalência forte de programas

2.4.1 REGRAS DE CONVERSÃO DE PROGRAMAS ITERATIVOS PARA MONOLÍTICOS

Conforme foi citado anteriormente, é possível converter qualquer programa iterativo em monolítico. Assim:

a) a operação vazia √ de um programa iterativo corresponde ao fluxograma da Figura 5(a) em um programa monolítico;

b) cada identificador de operação F do programa iterativo corresponde ao fluxograma da Figura 5(b);
c) supondo que V e W são programas iterativos, então a composição seqüencial “V, W” é o correspondente fluxograma ilustrado na Figura 5 (c);
d) supondo que T é um identificador de teste, a composição condicional “se T então V senão W” corresponde ao fluxograma ilustrado na Figura 5(d);
e) a composição “enquanto T faça (V)” corresponde ao fluxograma ilustrado na Figura 5(e);
f) a composição “até T faça (V)” corresponde ao fluxograma ilustrado na Figura 5(f).



Fonte: adaptado de Diverio e Menezes (2003, p. 35-36)

Figura 5 – Equivalência forte de programas: transformação de iterativo para monolítico

Os componentes elementares de *Partida* e *Parada* ilustrados na Figura 1 devem ser adicionados ao fluxograma.

2.4.2 REGRAS DE CONVERSÃO DE PROGRAMAS MONOLÍTICOS PARA RECURSIVOS

Segundo Diverio e Menezes (2003, p. 36), para qualquer programa monolítico (P_m) existe um programa recursivo (P_r).

Seja P_m um programa monolítico qualquer onde $L = \{r_1, r_2, ..., r_n\}$ é o correspondente conjunto de rótulos. Então P_r é um programa recursivo construído a partir de P_m e é tal que: P_r é R_1 onde R_1 def E_1, R_2 def E_2, ..., R_k def √ . Para $K \in \{1, 2, ..., n-1\}$, E_k é como segue:

a) o*peração*: se r_k é da forma: [r_k: faça F vá_para r_k’] então E_k é a seguinte expressão de sub-rotinas: [F;R_k’];

b) t*este*: se R_k é da forma [r_k: se T então vá_para r_k' senão vá_para r_k"] então E_k é a seguinte expressão de sub-rotinas: [(se T então R_k' senão R_k")].

2.5 INSTRUÇÕES ROTULADAS COMPOSTAS

Uma instrução rotulada composta é uma seqüência de símbolos (DIVERIO; MENEZES, 2003, p. 46) conforme exemplificado no Quadro 5 (supondo que F e G são identificadores de operação e que T é o único identificador de teste), onde r_2 e r_3 são ditos rótulos sucessores de r_1 e r_1 é dito rótulo antecessor de r_2 e r_3.

r_1: se T então faça F vá_para r_2 senão faça G vá_para r_3

Fonte: Diverio e Menezes (2003, p. 46)

Quadro 5 - Instrução rotulada composta

Um programa monolítico representado através de instruções rotuladas compostas P é um par ordenado P = (I,r), onde:

a) I é o conjunto de instruções rotuladas compostas o qual é finito;
b) r é o rótulo inicial o qual distingue a instrução rotulada inicial em I.

Não existem duas instruções diferentes com um mesmo rótulo e, um rótulo referenciado por alguma instrução o qual não é associado a qualquer instrução rotulada é dito um rótulo final.

Considerando um único identificador de teste, a instrução rotulada composta exemplificada no Quadro 5 pode ser abreviada, conforme ilustrado no Quadro 6.

r_1: (F, r_2), (G, r_3)

Fonte: Diverio e Menezes (2003, p. 46)

Quadro 6 - Instrução rotulada composta simplificada

Todo programa monolítico representado através de instruções rotuladas pode ser convertido para instruções rotuladas compostas.

Segundo Diverio e Menezes (2003, p. 14), instruções rotuladas compostas possuem uma única forma, diferentemente das instruções rotuladas que podem ser de duas formas: operação ou teste. Uma instrução rotulada composta combina ambas em uma única forma. No Quadro 7 tem-se o exemplo de um programa monolítico com instruções rotuladas compostas onde: F e G são operações, ciclo é um ciclo infinito, parada é o término do programa, o

símbolo ε indica que não existe uma próxima instrução a ser executada, e o símbolo ω é o rótulo que identifica a instrução de ciclo infinito.

1:	(G, 2), (F, 3)
2:	(G, 2), (F, 3)
3:	(F, 4), (G, 5)
4:	(F, 4), (G, 5)
5:	(F, 6), (ciclo, ω)
6:	(parada, ε), (G, 7)
7:	(G, 7), (G, 7)
ω :	(ciclo, ω), (ciclo, ω)

Fonte: Diverio e Menezes (2003, p. 49)

Quadro 7 - Conjunto de instruções rotuladas compostas

A execução de um programa na forma de instruções rotuladas compostas é da seguinte forma:

a) para cada instrução rotulada composta, faz-se um teste;
   - se o teste for verdadeiro:
     - executa-se a operação da instrução verdadeira,
     - executa-se o desvio da instrução verdadeira,
   - se o teste for falso:
     - executa-se a operação da instrução falsa,
     - executa-se o desvio da instrução falsa;
b) repetir o primeiro passo até que seja encontrada uma instrução de parada ou de ciclo infinito;
c) a execução de um programa deve ser feita após a sua simplificação. A simplificação de programas monolíticos será apresentada mais adiante.

2.5.1 REGRAS DE CONVERSÃO DE PROGRAMAS MONOLÍTICOS PARA INSTRUÇÕES ROTULADAS COMPOSTAS

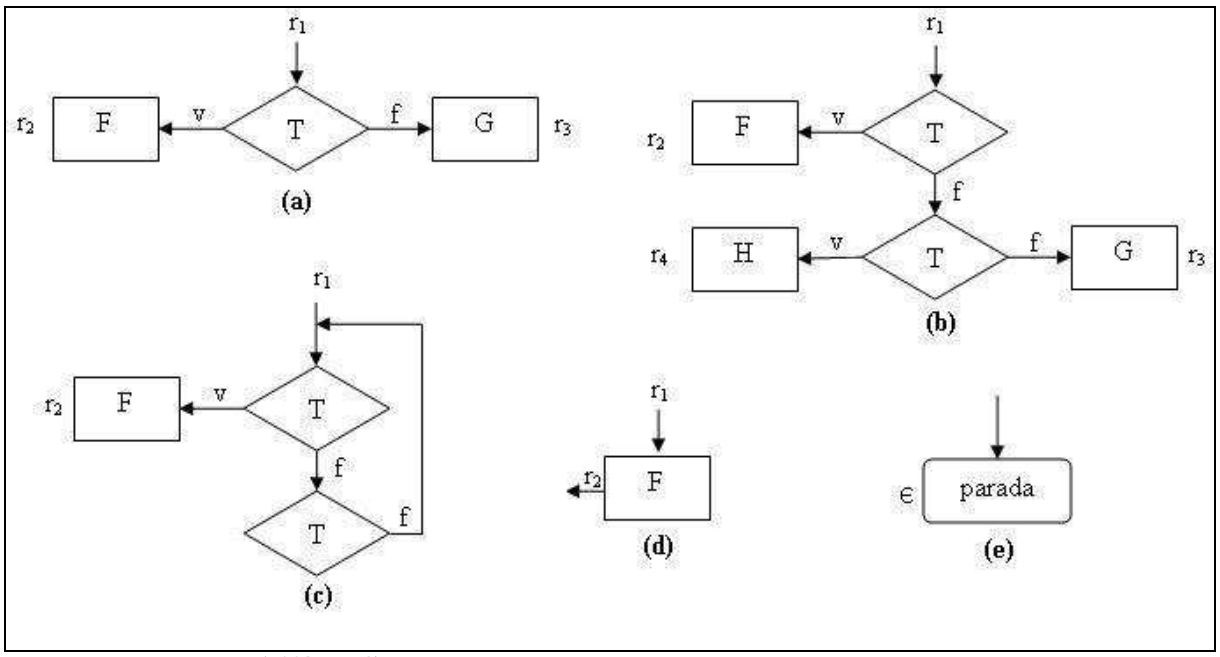
Conforme visto anteriormente, todo programa monolítico representado através de instruções rotuladas pode ser convertido para instruções rotuladas compostas. A seguir são apresentados dois algoritmos para conversão de instruções rotuladas em instruções rotuladas compostas. No primeiro algoritmo, descrito em Diverio e Menezes (2003, p. 42), o programa representado através de instruções rotuladas deve primeiramente ser transformado em um fluxograma, para depois ser convertido em um programa monolítico na forma de instruções rotuladas compostas. No segundo algoritmo, proposto em Silva (2004), o programa

monolítico com instruções rotuladas é convertido diretamente em um programa monolítico na forma de instruções rotuladas compostas.

2.5.1.1 ALGORITMO DESCRITO EM DIVERIO E MENEZES

Segundo Diverio e Menezes (2003, p. 42), os componentes elementares de partida, parada e operação são chamados de nós. O algoritmo para conversão de um fluxograma P em um programa monolítico P' constituído por instruções rotuladas compostas é da seguinte forma:

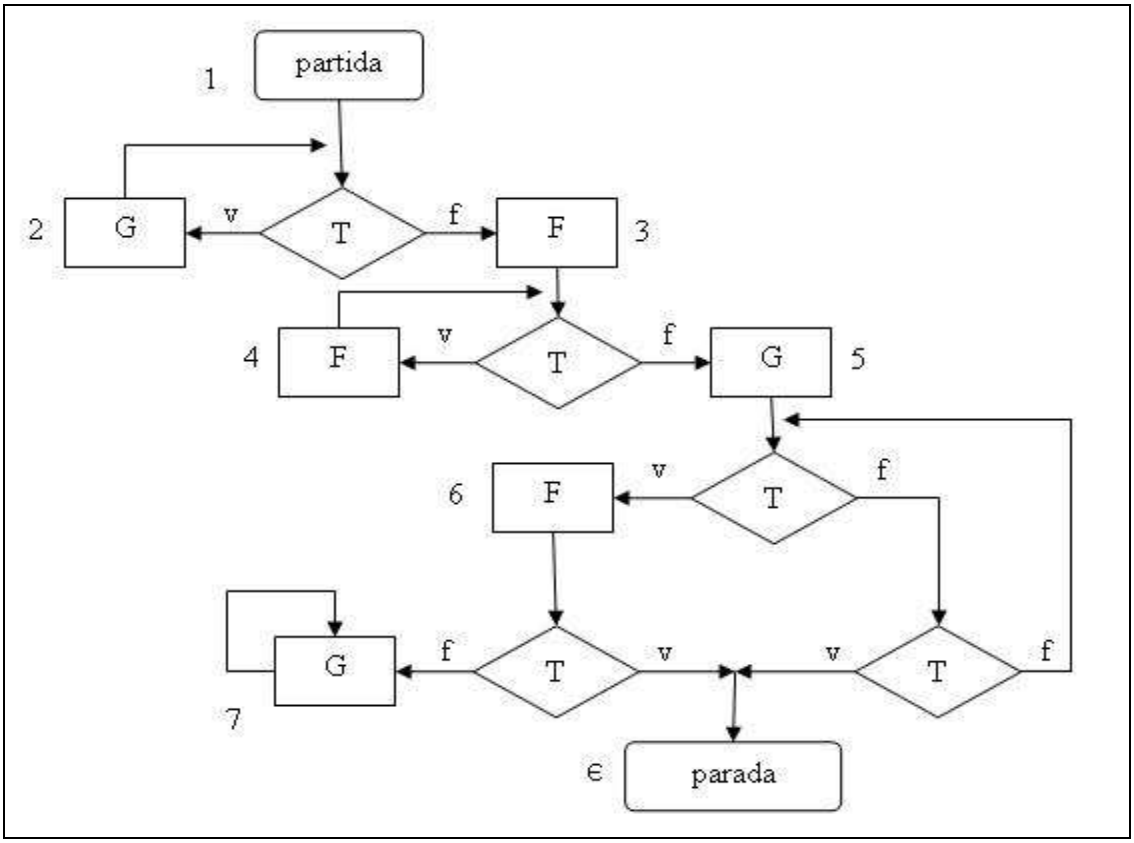
a) rotulação de nós: rotula-se cada nó do fluxograma, supondo que só exista um único nó de parada. O rótulo correspondente ao nó de partida é o rótulo inicial do programa P'. A rotulação dos nós de um fluxograma usa números naturais, fixando o número 1 para o nó de partida. O nó de parada deve ser rotulado com o símbolo ε;
b) instruções rotuladas compostas: a construção de uma instrução rotulada composta parte do nó de partida e segue o caminho do fluxograma. Dependendo do próximo componente tem-se que:
   - para um teste (Figura 6(a)), a correspondente instrução rotulada composta é a seguinte: [r_1: (F, r_2), (G, r_3)],
   - para uma operação (Figura 6(d)), a correspondente instrução rotulada composta é a seguinte : [r_1: (F, r_2), (F, r_2)],
   - para uma parada (Figura 6(e)), a correspondente instrução rotulada composta é a seguinte: [r: (parada, ε), (parada, ε)],
   - para testes encadeados (Figura 6(b)), segue-se o fluxo até que seja encontrado um nó, resultando na seguinte instrução rotulada composta: [r_1: (F, r_2), (G, r_3)],
   - para testes encadeados em ciclo infinito (Figura 6 (c)) a correspondente instrução rotulada composta é a seguinte:[r_1: (F, r_2), (ciclo, ω)]. Quando da ocorrência deste caso, deve ser incluída uma instrução rotulada composta correspondente ao ciclo infinito: [ω: (ciclo, ω), (ciclo, ω)].



Fonte: Diverio e Menezes (2003, p. 48)

Figura 6 - Fluxograma

A Figura 7 apresenta um fluxograma rotulado. O respectivo programa na forma de instruções rotuladas compostas é apresentado no Quadro 7.



Fonte: Diverio e Menezes (2003, p. 49)

Figura 7 - Fluxograma com os nós rotulados

2.5.1.2 ALGORITMO PROPOSTO POR SILVA

O algoritmo proposto por Silva (2004) transforma um programa monolítico na forma de instruções rotuladas diretamente em um programa monolítico na forma de instruções rotuladas compostas.

Para exemplificar o algoritmo, será tomando como exemplo o Programa Monolítico na forma de Instruções Rotuladas (PMIR) ilustrado no Quadro 8.

```
1: faça G vá_para 2
2: se T então vá_para 3 senão vá_para 5
3: faça F vá_para 4
4: se T então vá_para 10 senão vá_para 7
5: faca G vá_para 6
6: se T então vá_para 7 senão vá_para 8
7: faça H vá_para 10
8: faça F vá_para 9
9: faça G vá_para 1
```

Quadro 8 - Programa Monolítico na Forma de Instruções Rotuladas (PMIR)

O algoritmo para transformar um PMIR em um Programa Monolítico na forma de Instruções Rotuladas Compostas (PMIRC) obedece dois passos. No primeiro passo cria-se a "tabela de transformação" (Tabela 1) e no segundo passo cria-se o programa monolítico na forma de instruções rotuladas compostas a partir da "tabela de transformação".

Para executar o primeiro passo, procede-se da seguinte forma:

a) criar uma tabela com 4 colunas (C11, C12, C13 e C14) e *x* linhas (*x* é o número de linhas do PMIR mais uma). Identificar a primeira coluna (C11) de "linhas", a segunda (C12) de "rótulos do PMIRC", a terceira (C13) de "rótulos do PMIR" e a quarta (C14) de "instruções do PMIR";
b) numerar as linhas começando de 1 (C11);
c) inserir o PMIR a partir da segunda linha, separando os rótulos do PMIR na coluna C13 e as instruções correspondentes na coluna C14;
d) numerar a coluna C12 iniciando a primeira linha com 1. Continuar numerando (seqüencialmente) apenas as linhas desta coluna onde existir instrução do tipo operação (faça), deixando as linhas com instruções de teste (se...) em branco.

Tabela 1 - Tabela de transformação

Linhas	rótulos do PMIRC	rótulos do PMIR	instruções do PMIR
C11	C12	C13	C14
1	1		
2	2	1:	faça G vá_para 2
3	-	2:	se T então vá_para 3 senão vá_para 5
4	3	3:	faça F vá_para 4
5	-	4:	se T então vá_para 10 senão vá_para 7
6	4	5:	faça G vá_para 6
7	-	6:	se T então vá_para 7 senão vá_para 8
8	5	7:	faça H vá_para 10
9	6	8:	faça F vá_para 9
10	7	9:	faça G vá_para 1

Fonte: Silva (2004)

Para executar o segundo passo, procede-se da seguinte forma:

a) passo 1 - criar uma nova tabela (Tabela 2) com 5 colunas (C21, C22, C23, C24 e C25). Identificar as colunas da tabela (cabeçalhos), sendo a primeira coluna (C21) com “rótulos do PMIRC”; a segunda (C22) com “separador (literal ‘:’)”; a terceira (C23) com “instrução PMIRC - opção ‘verdadeira’ (então) - (Oper,rc’)”; a quarta (C24) com “separador (literal ‘,’)” e a quinta (C25) com “instrução PMIRC - opção ‘falsa’ (senão) - (Oper,rc’’)”. Quando da inserção de uma nova linha na tabela, a coluna C22 deve ser preenchida sempre com “:” (dois pontos) e a coluna C24 deve ser preenchida sempre com “,” (vírgula);

b) passo 2 - para construir uma instrução rotulada composta, iniciar a partir do rótulo do PMIRC (C12) da linha 1 da Tabela 1, o qual será sempre um e é o nó inicial do PMIRC (C21 da primeira linha);

c) passo 3 - percorrer o PMIR a partir do rótulo 1 da coluna “rótulos do PMIR” (C13);

d) passo 4 - dependendo da instrução do PMIR, têm-se:

   - rótulo sem instrução associada (determina parada no PMIR): a instrução rotulada composta é (parada, ε) na C23 e (parada, ε) na C25;
   - OPERAÇÃO (r: faca G vá_para r’): a instrução rotulada composta é rc na C21, (G,rc’) na C23 e (G,rc”) na C25; onde: rc (C21) é o rótulo do PMIRC da linha em questão já anotado (tabela 2); G (C23 e C25) é a operação e rc’ e rc” (C23 e C25

respectivamente) é o rótulo da coluna PMIRC (C12) da mesma linha da instrução do PMIR (C14 da Tabela 1);

- TESTE (r: se T então vá_para rc'senão vá_para rc"): segue o fluxo do PMIR, inicialmente para a opção verdade ("então") para determinar a C23 e após para a opção falsa ("senão") para determinar a C25, até que seja encontrado:
  - rótulo sem instrução associada (determina parada no PMIR): a instrução rotulada composta para a opção é (parada, ε),
  - operação: a instrução para a opção é (Oper,rc'/rc"), onde Oper é a operação em questão (C14) e rc´/rc´´ é o rótulo da coluna PMIRC (C12) da mesma linha da instrução do PMIR (C14 da Tabela 1),
  - teste que fecha o ciclo: quando percorrendo o PMIR (C13 e C14), chega-se em uma instrução de teste já verificada (não encontra-se instrução de operação ou parada). Neste caso, a instrução rotulada composta para a opção é (ciclo, ω);

e) passo 5 - a cada rótulo determinado nas colunas na C23 e C25 ainda não descrito na coluna C21, acrescentá-los na mesma (C21), exceto para o rótulo ω e ε;

f) passo 6 - pegar o próximo rótulo (C21) para o qual as colunas C23 e C25 ainda não foram definidas as instruções. Caso não existir, ir para o passo 9;

g) passo 7 - através do C21, acessar a C12 que coincide com o mesmo. Pegar o rótulo do comando vá_para (C14) para acessar a próxima instrução do PMIR;

h) passo 8 - voltar para o passo 4;

i) passo 9 – caso existir pelo menos um rótulo ω nas colunas C23 ou C25, acrescentar uma nova linha, identificando a C21 com ω e na C23 e C25 colocar a instrução (ciclo,ω) e finalizar o processo.

Tabela 2 - Tabela do PMIRC

rótulos do PMIRC (rc)	separador (literal ':')	instrução PMIRC - opção 'verdadeira' (então) - (Oper,rc')	separador (literal ',')	instrução PMIRC - opção 'falsa' (senão) - (Oper,rc'')
C21	**C22**	**C23**	**C24**	**C25**
1	:	(G, 2)	,	(G, 2)
2	:	(F, 3)	,	(G, 4)
3	:	(parada, ε)	,	(H, 5)
4	:	(H, 5)	,	(F, 6)
5	:	(parada, ε)	,	(parada, ε)
6	:	(G, 7)	,	(G, 7)
7	:	(G, 2)	,	(G, 2)

Fonte: Silva (2004)

2.6 PROPRIEDADES IDENTIFICÁVEIS DE UM PROGRAMA MONOLÍTICO

Sobre a estrutura estática de programas monolíticos com instruções rotuladas compostas, pode-se verificar propriedades importantes, as quais são: instruções mortas (inatingíveis) e ciclos infinitos.

Para a identificação destas propriedades, define-se uma seqüência finita de conjuntos.

Segundo Diverio e Menezes (2003, p. 51), uma seqüência de conjuntos $A_0A_1...$ é dita:

a) uma cadeia de conjuntos se, para qualquer $k \geq 0$, $A_k \subseteq A_{k+1}$;
b) uma cadeia finita de conjuntos é uma cadeia de conjuntos onde existe n, para todo $k \geq 0$, tal que $A_n = A_k = A_{k+1}$. Neste caso define-se o limite da cadeia finita de conjuntos como: $\lim A_k = A_n$.

2.6.1 INSTRUÇÕES MORTAS EM UM PROGRAMA MONOLÍTICO

Instruções mortas são aquelas que jamais serão atingidas a partir do estado inicial.

A identificação de uma instrução morta inicia-se a partir da instrução de "início", determinando os seus sucessores. Por exclusão, uma instrução que não possui antecessor é considerada uma instrução morta.

Seja I um conjunto de n instruções rotuladas. Seja $A_0A_1...$ uma seqüência de conjuntos de rótulos definida como segue: A_0 = { <Rótulo Inicial> }; $A_{K+1} = A_K$ U {r | r é rótulo de

instrução sucessora de alguma instrução rotulada por A_K}. Então A_0A_1... é uma cadeia finita de conjuntos e, para qualquer rótulo r de instrução de I, tem-se que, se $r \notin \lim A_k$, a instrução rotulada por r pode ser eliminada (instrução morta).

Considerando o exemplo do Quadro 9, a aplicação do algoritmo será a seguinte:

$A_0 = \{ 1 \}$
$A_1 = \{1, 2 \}$
$A_2 = \{1, 2, 3 \}$
$A_3 = \{1, 2, 3, 6\}$
$A_4 = \{1, 2, 3, 6\}$

Logo:

$\lim A_k = \{1, 2, 3, 6\}$
as instruções rotuladas 4 e 5 podem ser eliminadas, pois os rótulos 4 e 5 $\notin \lim A_k$

```
1: faça F vá_para 2
2: se T1 então vá_para 1 senão vá_para 3
3: faça G vá_para 6
4: faça F vá_para 5
5: faça G vá_para 6
6: se T2 então vá_para 7 senão vá_para 1
```

Fonte: adaptado de Diverio e Menezes (2003, p. 14)

Quadro 9 - Programa monolítico com instruções mortas

2.6.2 CICLOS INFINITOS EM UM PROGRAMA MONOLÍTICO

Ciclos infinitos são instruções que jamais alcançarão um rótulo final. Segundo Diverio e Menezes (2003, p. 51-52), a identificação dos ciclos infinitos inicia-se a partir da instrução de “parada”, determinando os seus antecessores. Por exclusão, uma instrução que não é antecessora de “parada” determina um ciclo infinito.

Seja I um conjunto de n instruções rotuladas compostas. Seja A_0A_1... uma seqüência de conjuntos de rótulos definida como segue: $A_0 = \{ \varepsilon \}$; $A_{K+1} = A_K \cup \{r \mid r$ é rótulo de instrução antecessora de alguma instrução rotulada por $A_K\}$. Então A_0A_1... é uma cadeia finita de conjuntos e, para qualquer rótulo r de instrução de I, tem-se que: $(I, r) \equiv (I, \omega)$ se, e somente se, $r \notin \lim A_k$ (DIVERIO; MENEZES, 2003, p. 51).

Pode-se tomar como exemplo o conjunto I de instruções rotuladas compostas do Quadro 7. A correspondente cadeia finita de conjuntos é a seguinte:

$A_0 = \{ \epsilon \}$
$A_1 = \{6, \epsilon \}$
$A_2 = \{5, 6, \epsilon \}$
$A_3 = \{3, 4, 5, 6, \epsilon \}$

$A_4 = \{1, 2, 3, 4, 5, 6, \epsilon\}$
$A_5 = \{1, 2, 3, 4, 5, 6, \epsilon\}$

Logo:

$\lim A_k = \{1, 2, 3, 4, 5, 6, \epsilon\}$
$(I, 7) \equiv (I, \omega)$, pois $7 \notin \lim A_k$

2.7 OTIMIZAÇÃO ESTRUTURAL DE UM PROGRAMA MONOLÍTICO

Através da identificação das instruções mortas e dos ciclos infinitos, um programa monolítico pode ser otimizado.

As instruções mortas identificadas no conjunto de instruções rotuladas podem ser excluídas, pois como elas jamais serão alcançadas, não terão nenhuma utilidade no programa. Se as instruções mortas não forem excluídas antes da transformação do programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas, durante a transformação, as instruções mortas serão automaticamente ignoradas, pois os dois algoritmos de transformação apresentados não irão converter os rótulos mortos. O Quadro 10 apresenta o programa com instruções rotuladas apresentado no Quadro 9, porém sem as instruções mortas identificadas.

```
1: faça F vá_para 2
2: se T1 então vá_para 1 senão vá_para 3
3: faça G vá_para 6
6: se T2 então vá_para 7 senão vá_para 1
```

Fonte: adaptado de Diverio e Menezes (2003, p. 14)

Quadro 10 - Programa monolítico sem instruções mortas

Segundo Diverio e Menezes (2003, p. 52), após identificados os ciclos infinitos, sendo I o conjunto finito de instruções rotuladas compostas, o algoritmo para simplificação dos ciclos infinitos é o seguinte:

a) os rótulos que não pertencerem ao $\lim A_k$ devem ser excluídos de I;
b) todas as referências em I aos rótulos excluídos devem ser substituídas pela instrução (ciclo, ω);
c) $I = I \cup [\omega: (ciclo, \omega), (ciclo, \omega)]$.

O conjunto de instruções rotuladas compostas apresentado no Quadro 11 corresponde à simplificação do conjunto de instruções rotuladas compostas do Quadro 7.

```
1: (G, 2), (F, 3)
2: (G, 2), (F, 3)
3: (F, 4), (G, 5)
4: (F, 4), (G, 5)
5: (F, 6), (ciclo, w)
6: (parada, e), (ciclo, w)
w: (ciclo, w), (ciclo, w)
```

Fonte: Diverio e Menezes (2003, p. 52)

Quadro 11 - Conjunto de instruções rotuladas compostas simplificadas

2.8 EQUIVALÊNCIA FORTE DE PROGRAMAS MONOLÍTICOS

Após a otimização de um programa monolítico na forma de instruções rotuladas compostas, pode-se verificar sua equivalência com outro programa monolítico.

Para a verificação da equivalência deve-se definir o que são rótulos consistentes e o que são rótulos equivalentes fortemente:

a) rótulos consistentes: seja I um conjunto de instruções rotuladas compostas e simplificadas. Sejam r e s dois rótulos de instruções em I (diferentes de ε). Supondo que as instruções rotuladas por r e s são da forma apresentada no Quadro 12 então, r e s são rótulos consistentes se, e somente se $F_1 = G_1$ e $F_2 = G_2$ (DIVERIO; MENEZES, 203, p. 53).

```
r:  (F1, r1), (F2, r2)
s:  (G1, s1), (G2, s2)
```

Fonte: Diverio e Menezes (2003, p. 53)

Quadro 12 - Rótulos consistentes

b) rótulos equivalentes fortemente: seja I um conjunto de instruções rotuladas compostas e simplificadas. Sejam r e s dois rótulos de instruções em I então, r e s são rótulos equivalentes fortemente se, e somente se (DIVERIO; MENEZES, 2003, p. 53):
   - ou $r = s = \varepsilon$,
   - ou r e s são ambos diferentes de ε e consistentes;

Segundo Diverio e Menezes (2003, p. 53), a determinação dos rótulos equivalentes fortemente deve ser feita da seguinte forma: sendo I um conjunto de instruções rotuladas compostas simplificadas, r e s dois rótulos de instruções de I, define-se a seqüência de conjuntos B_0B_1... como segue:

a) $B_0 = \{(r, s)\}$;
b) $B_{k+1} = \{(r'', s'') \mid r''$ e s'' são rótulos sucessores de r' e s', respectivamente, $(r', s') \in B_k$ e $(r'', s'') \notin B_I$ e $_I \in \{0, 1,...,K\}\}$.

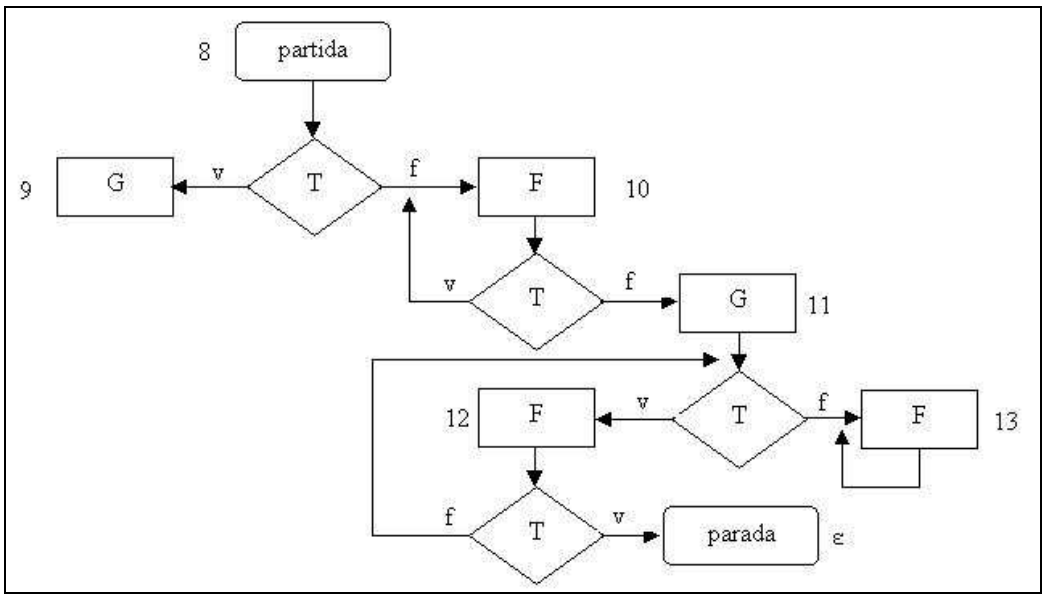
Então $B_0B_1...$ é uma seqüência de conjuntos que converge para o conjunto vazio e r, s são rótulos equivalentes fortemente se, e somente se, qualquer par de B_k é constituído por rótulos consistentes (DIVERIO; MENEZES, 2003, p. 53).

Para fazer a verificação da equivalência entre dois programas monolíticos é necessário fazer a união disjunta dos dois programas monolíticos na forma de instruções rotuladas compostas. Segundo Diverio e Menezes (2003, p. 50-51), a união disjunta garante que todos os elementos dos dois conjuntos (programas) constituem um conjunto resultante, mesmo possuindo a mesma identificação. Por exemplo, os conjuntos A = {a, x} e B = {b, x}, o conjunto resultante da união disjunta é o seguinte: $\{a, x_A, b, x_B\}$.

Sendo Q = (I_Q, q) e R = (I_R, r) dois programas monolíticos na forma de instruções rotuladas compostas, deve-se antes da união disjunta dos dois programas fazer a re-rotulação das instruções rotuladas. O rótulo inicial de R deverá ser o último rótulo de Q mais 1 ou seja, se o último rótulo de Q foi 7, o primeiro rótulo de R deverá ser 8.

Considerando os programas monolíticos representados na forma de fluxograma (Figura 7 e Figura 8), transformados em instruções rotuladas compostas (Quadro 7 e Quadro 13) e simplificados (Quadro 11 e Quadro 14), a união disjunta dos dois programas é apresentada no Quadro 15.

Sendo Q = (I_Q, q) e R = (I_R, r) dois programas monolíticos na forma de instruções rotuladas compostas simplificadas e sejam Pq = (I, q) e Pr = (I, r) programas monolíticos simplificados onde I é o conjunto resultante da união disjunta de I_Q e I_R. Então Pq **≡** Pr se, e somente se, Q **≡** R.



Fonte: adaptado de Diverio e Menezes (2003, p. 54)

Figura 8 - Programa monolítico na forma de fluxograma rotulado

```
8: (G, 9), (F, 10)
9: (G, 9), (F, 10)
10: (F, 10), (G, 11)
11: (F, 12), (F, 13)
12: (parada, ε), (F, 13)
13: (F, 13), (F, 13)
```

Fonte: Diverio e Menezes (2003, p. 55)

Quadro 13 - Instruções rotuladas compostas: verificação de equivalência

```
8:   (G, 9), (F, 10)
9:   (G, 9), (F, 10)
10:  (F, 10), (G, 11)
11:  (F, 12), (ciclo, w)
12:  (parada, ε), (ciclo, w)
w:   (ciclo, w), (ciclo, w)
```

Fonte: Diverio e Menezes (2003, p. 55)

Quadro 14 - Instruções rotuladas compostas simplificadas: verificação de equivalência

```
1:   (G, 2), (F, 3)
2:   (G, 2), (F, 3)
3:   (F, 4), (G, 5)
4:   (F, 4), (G, 5)
5:   (F, 6), (ciclo, w)
6:   (parada, ε), (ciclo, w)
8:   (G, 9), (F, 10)
9:   (G, 9), (F, 10)
10:  (F, 10), (G, 11)
11:  (F, 12), (ciclo, w)
12:  (parada, ε), (ciclo, w)
w:   (ciclo, w), (ciclo, w)
```

Fonte: Diverio e Menezes (2003, p. 55)

Quadro 15 - União disjunta (instruções rotuladas compostas simplificadas)

Para verificar se Q ≡ R (Quadros 11 e 14) são equivalentes, considerando a união disjunta apresentada no Quadro 15, é necessário verificar se (Pq, 1) ≡ (Pr, 8).

Como as instruções dos rótulos 1 e 8 são equivalentes fortemente então $B_0 = \{(1, 8)\}$. Verificando-se a equivalência entre os sucessores dos rótulos 1 e 8, têm-se a cadeia de conjuntos apresentado no Quadro 16, onde o conjunto B5 = ∅. Logo, (I, 1) ≡ (I, 8) e, portanto, Q ≡ R.

$B_0 = \{(1, 8)\}$	pares de rótulos equivalentes fortemente
$B_1 = \{(2, 9), (3, 10)\}$	pares de rótulos equivalentes fortemente
$B_2 = \{(4, 10), (5, 11)\}$	pares de rótulos equivalentes fortemente
$B_3 = \{(6, 12), (w, w)\}$	pares de rótulos equivalentes fortemente
$B_4 = \{(\varepsilon , \varepsilon)\}$	pares de rótulos equivalentes fortemente
$B_5 = \varnothing$	

Fonte: adaptado de Diverio e Menezes (2003, p. 56)

Quadro 16 - Pares de rótulos equivalentes fortemente

2.9 CODIFICAÇÃO DE CONJUNTOS ESTRUTURADOS

Segundo Diverio e Menezes (203, p. 68), elementos de tipos de dados estruturados são representados como números naturais. Para um dado conjunto estruturado X, define-se uma função injetora c: X → N, ou seja, uma função que para todo x,y ∈ X tem-se que: se c(x) = c(y), então x = y. Para este caso, o número natural c(x) é a codificação do elemento estruturado x.

As n-uplas naturais também podem ser representadas através de conjuntos estruturados, definindo uma função injetora: c: $N^n \rightarrow N$. Um número natural, pelo teorema fundamental da aritmética, pode ser decomposto em seus fatores primos. Supondo os n primeiros números primos denotados por $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ e assim sucessivamente então, a codificação c: $N^n \rightarrow N$ definida no Quadro 17 é unívoca (supondo que $(x_1\ x_2, x_3, ..., x_n) \in N^n$ e que o símbolo * denota a operação de multiplicação nos naturais) (DIVERIO; MENEZES, 2003, p. 69).

$$c(x_1, x_2,\ldots,x_n) = p_1^{x1} * p_2^{x2} * \ldots p_n^{xn}$$

Fonte: adaptado de Diverio e Menezes (2003, p. 69)

Quadro 17 - Codificação de n-uplas naturais

Esta codificação não constitui uma função bijetora, ou seja, nem todo número natural corresponde a uma n-upla. Entretanto, todo número natural decomponível nos n primeiros números primos corresponde a uma n-upla (DIVERIO; MENEZES, 2003, p. 69).

Segundo Diverio e Menezes (2003, p. 69), programas monolíticos na forma de instruções rotuladas também podem ser representados através da codificação de conjuntos estruturados. Um programa monolítico pode ser codificado como um número natural onde, as instruções rotuladas de operação do tipo: "r_1: faça F_k vá_para r_2" e de teste do tipo: "r_1: se T_k então vá_para r_2 senão vá_para r_3" podem ser denotadas pelas quadras ordenadas (onde a primeira componente identifica o tipo da instrução):

a) operação: (0, k, r2, r2):
b) teste: (1, k, r2, r3)

Supondo o seguinte número: p = (2^{150}) * (3^{105}), portanto o programa possui duas instruções rotuladas correspondentes aos números 150 e 105. Decompondo ambos os números em fatores primos tem-se que: $150 = 2^1 * 3^1 * 5^2 * 7^0$ e $105 = 2^0 * 3^1 * 5^1 * 7^1$, o que corresponde as quádruplas: (1, 1, 2, 0) e (0, 1, 1, 1). Logo, as instruções rotuladas decodificadas são como apresentado no Quadro 18 (DIVERIO; MENEZES, 2003, p. 70).

1: se T_1 então vá_para 2 senão vá_para 0
2: faça F_1 vá_para 1

Fonte: Diverio e Menezes (2003, p. 70)

Quadro 18 - Instruções rotuladas decodificadas

2.10 MÁQUINA NORMA

A máquina NORMA, proposta em Bird (1976, p. 50), é uma máquina de registradores. Possui um conjunto infinito de registradores naturais como memória e somente três operações sobre cada registrador:

a) adição do valor um;
b) subtração do valor um;
c) teste se o valor armazenado é zero.

Segundo Diverio e Menezes (2003, p. 72), a máquina NORMA é universal e extremamente simples, com poder computacional de qualquer computador moderno.

2.10.1 OPERAÇÕES E TESTES

Com as operações e o teste definidos na máquina NORMA, pode-se exemplificar outras operações e testes tais como:

a) atribuição de valor a um registrador: a atribuição [A:=0] pode ser obtida através do programa iterativo descrito no Quadro 19. Esta atribuição poderá ser usada como uma macro, facilitando a definição de atribuição de um valor qualquer a um registrador. No Quadro 20 é apresentado um programa iterativo que faz a atribuição: [A:= n], com n = 3 ;

```
Até A = 0
Faça (A:= A - 1)
```

Fonte: Diverio e Menezes (2003, p. 73)
Quadro 19 - Macro A := 0

```
A := 0;
A := A + 1;
A := A + 1;
A := A + 1
```

Fonte: Diverio e Menezes (2003, p. 73)
Quadro 20 - Macro A := 3

b) adição de dois registradores: a macro [A:= A + B] pode ser obtida através do programa iterativo descrito no Quadro 21. Esta macro zera o valor do registrador B. Se for necessário preservar o valor do registrador B, deve-se utilizar a macro [A:= A + B usando C], conforme descrito no Quadro 22;

```
até B = 0
faça (A:= A + 1; B:= B - 1)
```

Fonte: Diverio e Menezes (2003, p. 74)

Quadro 21 - Macro A := A + B

```
C:= 0
até B = 0
faça (A:= A + 1; C:= C + 1; B:= B - 1);
até C = 0
faça (B:= B + 1; C:= C - 1
```

Fonte: Diverio e Menezes (2003, p. 74)

Quadro 22 - Macro A := A + B usando C

c) multiplicação de dois registradores: a macro [A:= A x B usando C, D] descrita no Quadro 23, executa multiplicação de A por B, usando dois registradores de trabalho C e D;

```
C:= 0
até A = 0
faça (c:= c + 1; A:= A - 1);
até C = 0
faça (A:= A + A usando D; C:= C -1)
```

Fonte: Diverio e Menezes (2003, p. 75)

Quadro 23 - A:= A x B usando C, D

d) teste se o valor de um registrador é um número primo: a macro [teste_primo(A) usando C] descrita no Quadro 24 verifica se o valor de um registrador é um número primo, onde teste_mod(A, C) é um teste que retorna o valor verdadeiro, se o resto da divisão inteira do conteúdo de A por C é zero, e o valor falso caso contrário.

```
(se A = 0
então falso
senão C:= A;
      C:= C - 1;
       (se C = 0
      então verdadeiro
      senão até teste_mod(A,C)
            faça (C:= C -1)
            C:=C - 1;
             (se C = 0
            então verdadeiro
            senão falso)))
```

Fonte: Diverio e Menezes (2003, p. 75)

Quadro 24 - Macro teste_primo(A) usando C

Além dos exemplos citados e exemplificados anteriormente, pode-se citar outros como: fatorial de um número, potenciação, teste “menor”, teste “menor ou igual”, entre outros.

2.10.2 VALORES NUMÉRICOS

A máquina NORMA utiliza somente números naturais, porém através deles pode-se definir outros tipos numéricos, como por exemplo inteiros, racionais, entre outros.

Segundo Diverio e Menezes (2003, p. 76), um valor inteiro m pode ser representado através de um par ordenado (s, |m|), onde :

a) |m| é a magnitude dada pelo valor absoluto de m;

b) s é o sinal de m (se m < 0, então s = 1 senão s = 0).

Uma forma para representar o par ordenado definido para um valor inteiro na máquina NORMA é a utilização de dois registradores: um para o sinal e outro para a magnitude (valores absolutos). A simulação da operação inteira [A:= A + 1], supondo que A denota o par de registradores A_1 (sinal) e A_2 (magnitude), pode ser realizada pelo programa iterativo apresentado do Quadro 25 (DIVERIO; MENEZES, 2003, p. 76).

```
(se   A1 = 0
então A2:=A2 + 1
senão A2:=A2 - 1;
      (se    A2 = 0
       então A1:= A1 - 1
       senão √))
```

Fonte: Diverio e Menezes (2003, p. 77)

Quadro 25 - Operação inteira

Um valor racional r pode ser definido como um par ordenado (a, b), tal que b > 0 e r = a/b. O Quadro 26 mostra a definição das operações de adição, subtração, multiplicação, divisão e teste de igualdade.

```
(a, b) + (c, d) = (a*d + b*c, b*d)
(a, b) - (c, d) = (a*d - b*c, b*d)
(a, b) x (c, d) = (a*c, b*d)
(a, b) ÷ (c, d) = (a*d, b*c) com c ≠ 0
(a, b) = (c, d) se, e somente se, a*d = b*c
```

Fonte; Diverio e Menezes (2003, p. 77)

Quadro 26 - Operações sobre números racionais

2.10.3 DADOS ESTRUTURADOS

Pode-se definir a partir da máquina NORMA, estruturas do tipo arranjo unidimensional e multidimensional (DIVERIO; MENEZES, 2003, p. 78).

Segundo Diverio e Menezes (2003, p. 78), uma estrutura do tipo arranjo unidimensional pode ser definida por um único registrador, usando a codificação de n-uplas naturais. O arranjo não necessita ter tamanho máximo (número de posições indexáveis) predefinido. Em uma estrutura do tipo arranjo pode-se indexar as posições de forma direta (número natural) ou indireta (conteúdo de um registrador). Em ambos os casos, devem ser definidas as operações de adição e subtração do valor 1, e também do teste se o valor é zero, supondo que:

a) o arranjo unidimensional é implementado usando o registrador A;
b) p_n denota o n-ésimo número primo;
c) o teste deve retornar o valor verdadeiro, se o conteúdo do registrador C é um divisor do conteúdo do registrador A e falso, caso contrário.

A indexação direta realizada pelas macros $ad_{A(n)}$ usando C, $sub_{A(n)}$ usando C e $zero_{A(n)}$ usando C onde, $A_{(n)}$ denota a n-ésima posição do arranjo podem ser definidas pelos programas iterativos dos Quadros 27, 28, 29 respectivamente (DIVERIO; MENEZES, 2003, p. 78).

```
C:= Pn;
A:= A x C
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 27 - Programa iterativo $ad_{A(n)}$ usando C

```
C:=pn;
(se   teste_div(A,C)
então A:=A/C
senão √)
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 28 - Programa iterativo $sub_{A(n)}$ usando C

```
C:=pn;
(se   teste_div(A,C)
então falso
senão verdadeiro)
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 29 - Programa iterativo $zero_{A(n)}$ usando C

Segundo Diverio e Menezes (2003, p. 79), a indexação indireta realizada pelas macros $ad_{A(B)}$ usando C, $sub_{A(B)}$ usando C e $zero_{A(B)}$ usando C onde, A(B) denota a b-ésima posição do arranjo A e b é o conteúdo do registrador B, podem ser definidas pelos programas iterativos dos Quadros 30, 31 e 32 respectivamente.

```
C:= primo(B);
A:= A x C
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 30 - Programa iterativo $ad_{A(B)}$ usando C

```
C:=primo(B);
(se   teste_div(A,C)
então A:=A/C
senão √)
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 31 - Programa iterativo $sub_{A(B)}$ usando C

```
C:=primo(B);
(se   teste_div(A,C)
então falso
senão verdadeiro)
```

Fonte: Diverio e Menezes (2003, p. 79)
Quadro 32 - Programa iterativo $zero_{A(B)}$ usando C

2.10.4 ENDEREÇAMENTO INDIRETO E RECURSÃO

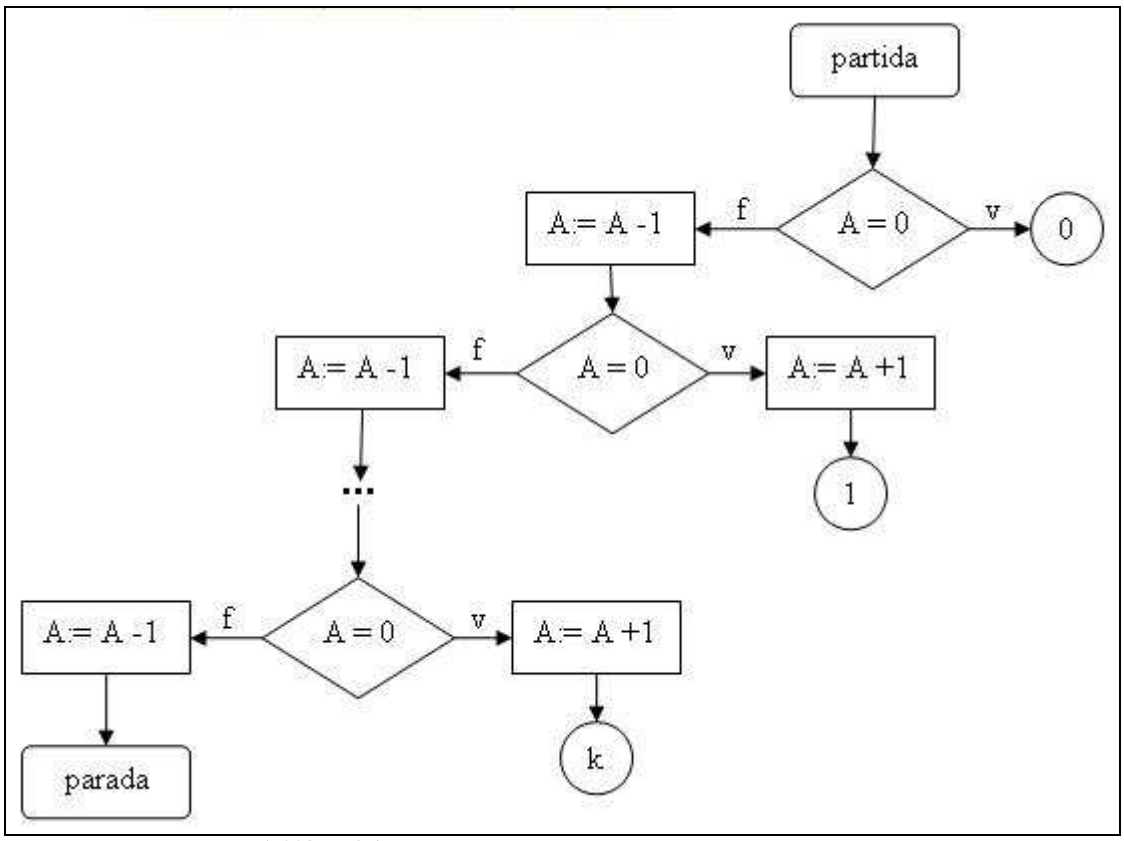
Em programas monolíticos pode-se utilizar desvios através de endereçamento indireto (determinado pelo conteúdo de um registrador). Uma operação com endereçamento indireto conforme o Quadro 33, onde A é um registrador, pode ser definida pelo programa monolítico ilustrado no Quadro 34, onde a macro End_A, ilustrada na Figura 9, trata o endereçamento indireto de A, onde cada circunferência rotulada representa um desvio incondicional para a correspondente instrução (DIVERIO; MENEZES, 2003, p. 81).

```
r: faça F vá_para A
```

Fonte: Diverio e Menezes (2003, p. 81)
Quadro 33 - Endereçamento indireto

```
r: faça F vá_para End_A
```

Fonte: Diverio e Menezes (2003, p. 81)
Quadro 34 - Endereçamento indireto utilizando macro



Fonte: Diverio e Menezes (2003, p. 82)

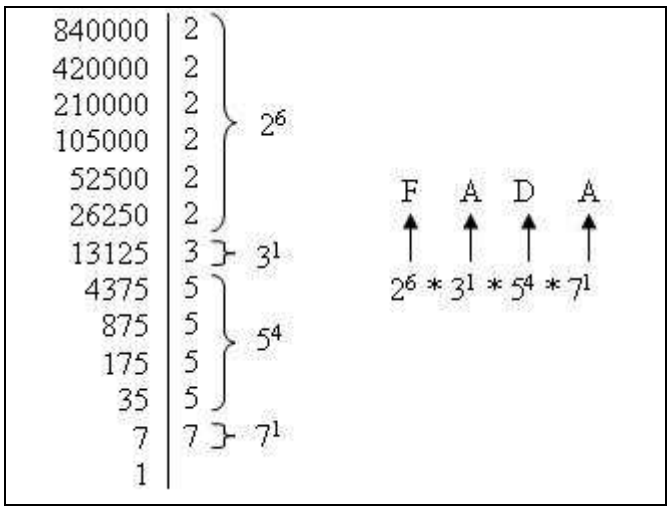
Figura 9 - Fluxograma para tratar endereçamento indireto

2.10.5 CADEIAS DE CARACTERES

Cadeias de caracteres podem ser representados na máquina NORMA, através da codificação de n-uplas naturais. Supondo que cada caractere equivale a um número natural (A=1, B=2, C=3, D=4,...), pode-se tomar como exemplo a palavra “FADA” (Quadro 35); onde a letra “F” equivale ao número 6, a letra “A” ao número 1 e a letra “D” ao número 4. Então, tem-se a seguinte quádrupla: “FADA” = $2^6 * 3^1 * 5^4 * 7^1$. Logo, a palavra “FADA” equivale ao número natural 840000. Se a decomposição em fatores primos for feita sobre o número natural 840000, conforme apresentado no Quadro 36, retorna-se a palavra “FADA”.

F = 6 A = 1 D = 4 A = 1	“FADA” = $2^6 * 3^1 * 5^4 * 7^1$ “FADA” = 64 * 3 * 625 * 7 “FADA” = 840000

Quadro 35 - Codificação da palavra "FADA"



Quadro 36 - Decodificação da palavra "FADA"

2.11 COMPILADORES

Segundo Aho, Sethi e Ullman (1995, p. 1-5) um compilador é um programa que lê um programa escrito em uma linguagem (linguagem fonte) e o traduz num programa equivalente em uma outra linguagem (linguagem alvo). O compilador relata ao seu usuário a presença de erros no programa fonte. Um compilador opera em várias fases, cada uma das quais transforma o programa fonte de uma representação para outra. Uma decomposição comum de um compilador é apresentada na Figura 10.

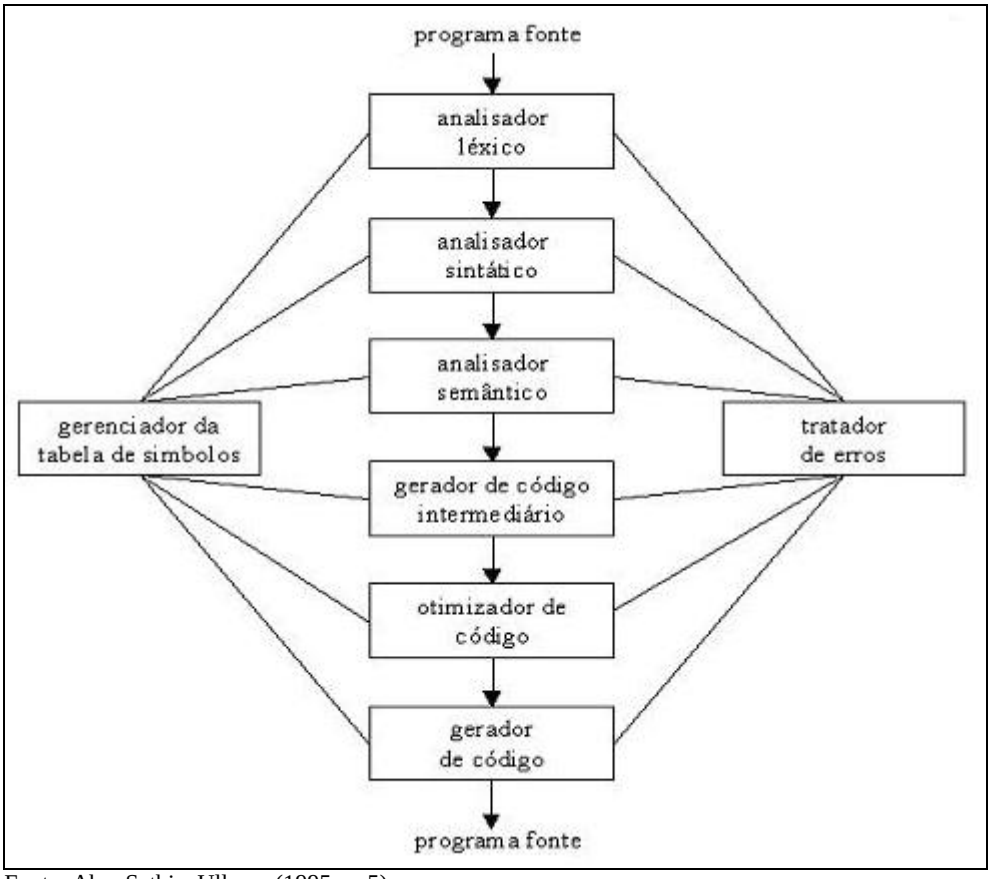
As fases de um compilador são:

a) análise léxica: a principal tarefa de um analisador léxico segundo Aho, Sethi e Ullman (1995, p. 38) é a de ler os caracteres de entrada e produzir uma seqüência de *tokens* que será utilizada para a análise sintática. O analisador léxico ao fazer a varredura do arquivo fonte deve desconsiderar os comentários, os caracteres em branco e verificar se os *tokens* lidos são válidos;
b) análise sintática: a análise sintática valida a estrutura gramatical do programa fonte, verificando se a mesma está em conformidade com as regras gramaticais especificadas para a linguagem;
c) análise semântica: a função do analisador semântico é verificar se as estruturas sintáticas montadas na etapa da análise sintática possuem sentido. Nesta fase do compilador são verificados alguns itens como: identificadores não declarados, incompatibilidade de tipos, declaração de funções, procedimentos, variáveis, etc;
d) gerador de código intermediário: é a fase em que se dá início à montagem do código objeto. Utilizando o resultado produzido pelas fases anteriores (análises

sintática e semântica), uma seqüência de código é gerada. Essa seqüência de código pode sofrer mudanças durante sua geração;

e) otimizador de código: segundo Aho, Sethi e Ullman (1995, p. 7), esta fase procura melhorar o código intermediário de tal forma que venha resultar em um código de máquina mais rápido em tempo de execução;

f) gerador de código: é a fase final de um compilador, onde segundo Aho, Sethi e Ullman (1995, p. 7) ocorre a geração do código alvo, consistindo normalmente de código de máquina relocável ou código de montagem.



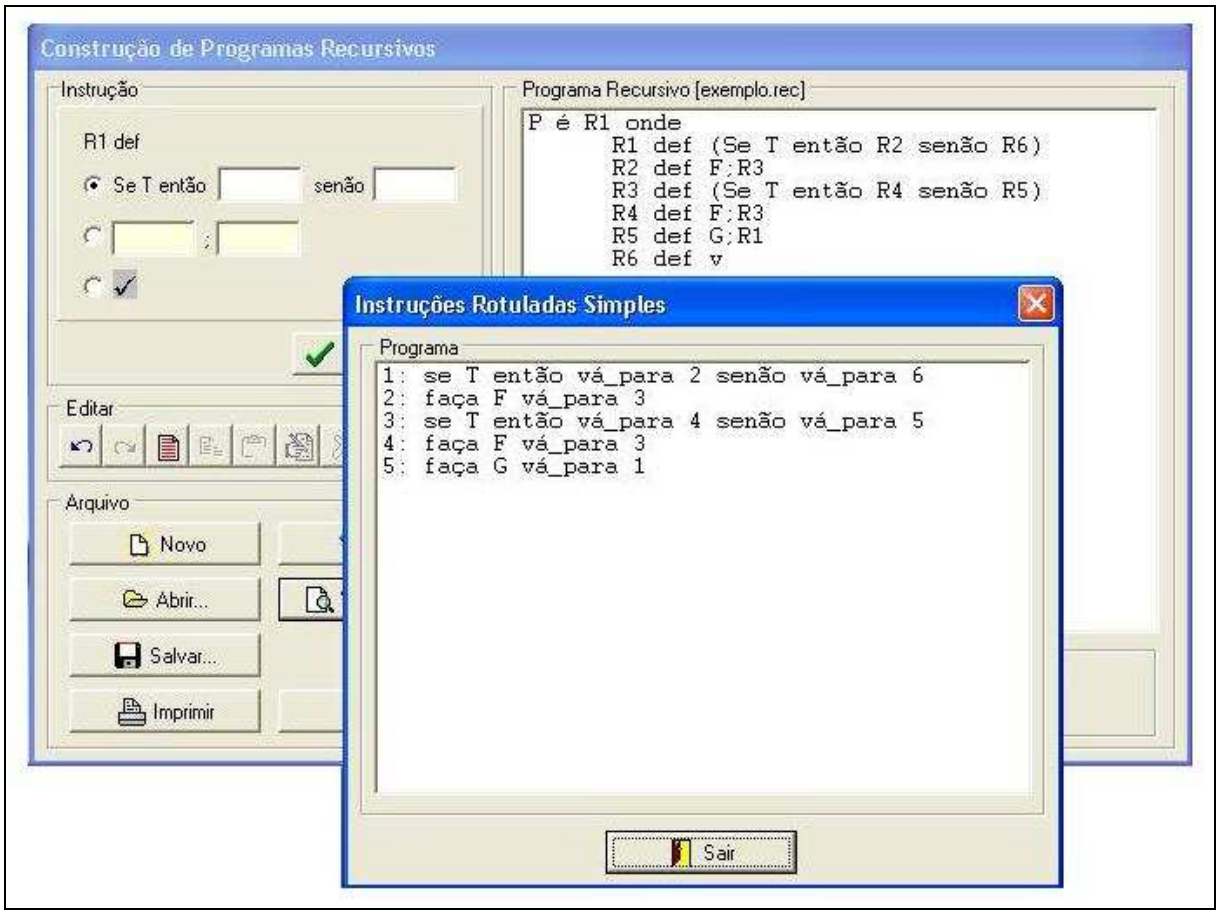
Fonte: Aho, Sethi e Ullman (1995, p. 5)

Figura 10 - Fases de um Compilador

2.12 TRABALHOS CORRELATOS

Fernandes et al (2004) apresenta um protótipo para converter programas monolíticos na forma de instruções rotuladas em programa recursivo, além de possibilitar a simplificação do mesmo. O ambiente permite criar ou editar um programa recursivo, com opções para adicionar novas instruções, removê-las, desfazer ou refazer operações e imprimir o programa recursivo simplificado ou não.

O ambiente permite ainda a verificação da equivalência entre dois programas, podendo-se entrar com os programas na forma de fluxograma, instruções rotuladas simples ou instruções rotuladas compostas. Um exemplo da interface do ambiente é apresentado na Figura 11.



Fonte: Fernandes (2004, p. 9)

Figura 11 - Software Programas Recursivos: tela de visualização do programa monolítico

3 DESENVOLVIMENTO DO PROTÓTIPO

As seções deste capítulo têm como objetivo descrever as ferramentas utilizadas, os requisitos do sistema, a análise do sistema usando OO com a UML, bem como a descrição da implementação e sua a operacionalidade.

3.1 FERRAMENTAS UTILIZADAS

Para a especificação do ambiente utilizou-se a ferramenta *Rational Rose*, a qual é uma ferramenta bastante conhecida e não será detalhada, assim como a ambiente de programação Delphi 7.0, utilizado na fase de implementação.

Para a especificação da linguagem criada, utilizou-se a ferramenta GALS, que é apresentada a seguir.

3.1.1 GERADOR DE ANALISADORES LÉXICOS E SINTÁTICOS (GALS)

Segundo Aho, Sethi e Ullman (1995, p. 10), logo após a escrita dos primeiros compiladores, surgiram sistemas como compiladores de compiladores, geradores de compiladores e sistemas de escrita de tradutores, que auxiliam no processo de escrita de compiladores.

O GALS é uma ferramenta que possibilita a geração de código dos analisadores léxico e sintático, para o ambiente Delphi, Java e C++ (GESSER, 2003). Para que possam ser gerados os analisadores léxico e sintático, é preciso entrar com informações em quatro campos:

a) “Definições Regulares”: as definições regulares deverão seguir a seguinte forma: [identificador] : [expressão regular] onde, cada linha pode conter apenas uma definição regular. As definições regulares declaradas poderão ser utilizadas em outras expressões regulares, utilizando seu identificador entre “{” e “}”;
b) “Tokens”: os tokens são todas as palavras ou símbolos terminais que serão utilizados na especificação da gramática;
c) “Não Terminais”: é uma lista contendo todos os símbolos não-terminais especificados na gramática. O primeiro símbolo da lista é considerado como símbolo inicial da gramática;

d) “Gramática”: é a declaração das produções, baseada na notação BNF. Os símbolos não-terminais devem ser previamente declarados. O uso de um símbolo não-terminal não declarado gera um erro semântico.

Algumas opções podem ser configuradas, como exemplo, o analisador sintático pode ser do tipo ascendente ou descendente. Antes da geração de código, a ferramenta possibilita a verificação de erros e de símbolos inúteis. Um simulador pode ser executado na própria ferramenta, para que a gramática possa ser testada. A interface da ferramenta GALS é apresentada na Figura 12.

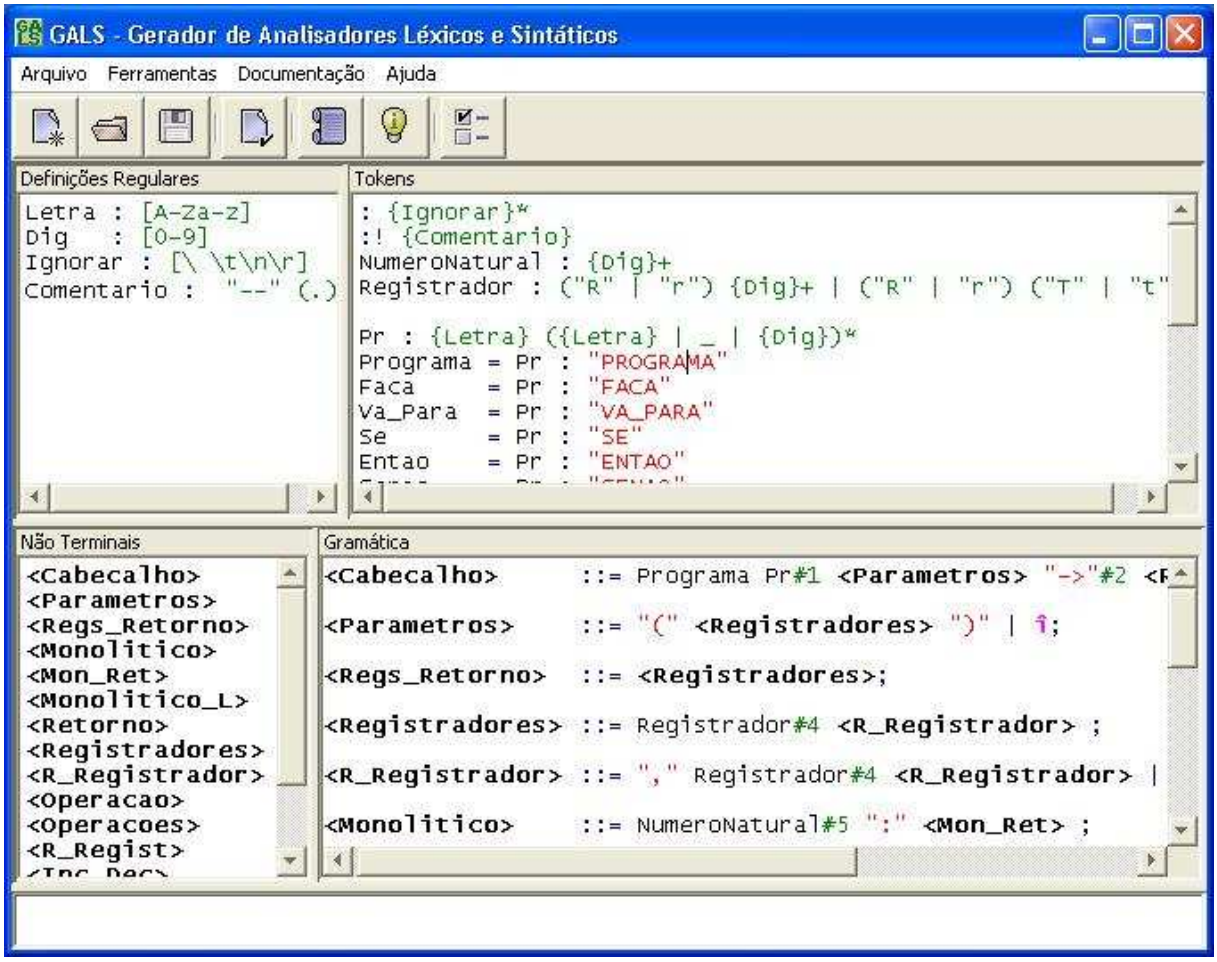


Figura 12 - Interface do Gerador de Analisadores Léxicos e Sintáticos (GALS)

3.2 ESPECIFICAÇÃO INFORMAL DA LINGUAGEM

A linguagem criada foi baseada na estrutura dos programas monolíticos na forma de instruções rotuladas, com a mesma estrutura de dados da máquina NORMA. Possui um número infinito de registradores como memória, e as seguintes operações sobre cada registrador:

a) atribuição de um valor;
b) atribuição do valor de um registrador para outro registrador;
c) adição do valor um;
d) subtração do valor um;
e) teste se o valor armazenado é zero.

Os registradores devem ser iniciados com a letra “r” ou “R”, seguido de um número natural ou da letra “t” ou “T”. Exemplos de registradores são: rt, r1, r2, r3,..., rn. A linguagem não é *case sensitive* ou seja, a definição de um registrador “r1” ou “R1” é a mesma.

A linguagem possui instruções de operação, teste e retorno. A primeira instrução do programa deve ser um cabeçalho, onde são definidos o nome do programa, a função de entrada e a função de saída. O cabeçalho de um programa tem a seguinte forma: “programa <nome do programa> (<função entrada>) -> <função saída>” onde:

a) <nome do programa>: é um conjunto de caracteres formados por letras, números e o caractere “_”, sendo que não pode haver um nome de programa com o formato de um registrador. Quando um programa for salvo, o nome do arquivo deve ser o mesmo do nome do programa, pois o nome do arquivo será utilizado para a chamada de macros, que serão detalhadas mais adiante;
b) <função entrada>: é um conjunto finito de registradores, separados uns dos outros por “,”. A função de entrada pode não existir, no caso de uma função constante;
c) <função saída>: é um conjunto finito de registradores, separados uns dos outros por “,”. A função de saída deve possuir pelo menos um registrador.

Um exemplo de cabeçalho é apresentado no Quadro 37.

```
programa Exemplo_Cabecalho (r1, r2) -> r3, r4
```

Quadro 37 - Cabeçalho de um programa

Logo após o cabeçalho pode-se então definir um conjunto de instruções rotuladas de operações e testes.

Uma instrução de operação tem a seguinte forma: “<rotulo>: faca <operação> va_para <rotulo desvio>” onde:

a) <rótulo>: é um número natural que identifica a instrução rotulada, sendo que um rótulo pode ser atribuído a somente uma instrução rotulada;
b) <operação>: uma operação sobre um registrador pode ser uma das seguintes formas:
   - inc(<registrador>): esta operação incrementa o valor do registrador passado como parâmetro (Quadro 38),
   - dec(<registrador>): esta operação decrementa o valor do registrador passado como parâmetro, se o seu valor for maior que zero (Quadro 39),
   - <registrador> = <registrador>: esta operação atribui o valor de um registrador a outro registrador (Quadro 40),
   - <registrador> = <valor natural>: esta operação atribui um valor natural a um registrador (Quadro 41),
   - <lista de registradores> = <macro>: esta operação permite a chamada de uma macro, que deve estar armazenada dentro de uma pasta chamada “Lib”, no mesmo diretório onde o arquivo executável do ambiente foi instalado. Quando ocorrer a chamada de uma macro, a função de saída da mesma será atribuída a lista de registradores (Quadro 42, Quadro 43);
c) <rótulo desvio>: este rótulo é um número natural, que aponta para uma instrução rotulada. Uma instrução rotulada de operação pode ter como rótulo de desvio, o seu próprio rótulo. Porém, quando o programa for convertido para instruções rotuladas compostas, esta instrução irá gerar uma instrução de ciclo infinito, e no processo de simplificação das instruções rotuladas compostas ela será automaticamente eliminada.

```
1: faca inc(r1) va_para 2
```

Quadro 38 - Instrução de adição do valor um

```
1: faca dec(r2) va_para 2
```

Quadro 39 - Instrução de subtração do valor um

```
1: faca r1 = r2 va_para 2
```

Quadro 40 - Instrução de atribuição do valor de um registrador para outro registrador

```
1: faca r1 = 0 va_para 2
```

Quadro 41 - Instrução de atribuição de um valor natural a um registrador

O Quadro 42 apresenta um exemplo de chamada de uma macro “igual”, onde são passados dois registradores como parâmetro. Esta macro retorna o número natural 0 (zero) se os valores dos dois parâmetros são iguais e o número natural 1 (um) caso contrário.

```
1: faca r1 = igual(r1, r2) va_para 2
```

Quadro 42 - Atribuição com chamada de uma macro com parâmetros

O Quadro 43 apresenta outro exemplo de chamada de macro, porém neste caso não são passados parâmetros para a macro. Esta macro retorna simplesmente a constante PI, onde o valor inteiro da constante é atribuída ao registrador “r1” e a parte decimal ao registrador “r2”.

```
1: faca r1, r2 = constante_PI va_para 2
```

Quadro 43 - Atribuição com chamada de uma macro constante

Para que uma instrução rotulada de teste possa ser feita, foi definido um registrador especial denominado “rt”. A instrução rotulada de teste sempre irá verificar se o valor do registrador “rt” é zero. Por este motivo, o registrador de teste “rt” deve receber um valor antes da primeira instrução de teste do programa, caso contrário acontecerá um erro na compilação do programa. Uma instrução rotulada de teste não altera o valor de nenhum registrador do programa, nem mesmo o valor de “rt”.

Uma instrução de teste possui a seguinte forma: “<rótulo> : se T entao va_para <rótulo desvio verdadeiro> senao va_para <rótulo desvio falso>” onde:

a) <rótulo>: é um número natural que identifica a instrução rotulada, sendo que um rótulo pode ser atribuído a somente uma instrução rotulada;
b) <rótulo desvio verdadeiro>: é um número natural, que aponta para a próxima instrução rotulada a ser executada, caso o valor de registrador de teste seja zero;
c) <rótulo desvio falso>: é um número natural, que aponta para a próxima instrução rotulada a ser executada, caso o valor de registrador de teste seja diferente de zero.

Os desvios de uma instrução de teste não podem ser para uma outra instrução de teste, a não ser para ela mesma. Como só existe um registrador de teste, se o valor do registrador de teste na primeira instrução for igual a zero, então na segunda também seria zero. Se o desvio de uma instrução for para ela mesma, quando o programa for transformado para instruções rotuladas compostas, esta instrução irá gerar uma instrução de ciclo infinito.

Para finalizar um programa, deve-se incluir uma instrução de retorno, que possui a seguinte forma: “<rótulo>: retorna”, onde o <rótulo> é um número natural que identifica a instrução rotulada. Só pode existir uma instrução de retorno em um programa, a qual deve ser a última. Se o programa possuir uma instrução de retorno, mas nenhuma outra instrução do programa chamá-la, quando o programa for transformado em instruções rotuladas compostas, o mesmo será simplificado a uma única instrução de ciclo infinito.

O Quadro 44 apresenta um programa que compara se dois números são iguais. O Quadro 45 apresenta outro programa, que compara se três números são iguais, auxiliado por uma macro, que é o programa do Quadro 44. Em ambos os casos, como a linguagem só utiliza registradores do tipo natural, o retorno dos programas será o número natural 0 (zero) caso a comparação seja verdadeira e o número natural 1 (um) caso contrário.

```
programa Comp_Dois_Num_Iguais(R1, R2)-> rT
1: faca rt= r1 va_para 2
2: se T entao va_para 3 senao va_para 5
3: faca rt = r2 va_para 4
4: se T entao va_para 10 senao va_para 7
5: faca rt= r2 va_para 6
6: se T entao va_para 7 senao va_para 8
7: faca rt=1 va_para 10
8: faca dec(r1) va_para 9
9: faca dec(r2) va_para 1
10: retorna
```

Quadro 44 - Programa que compara se dois números naturais são iguais

```
programa Comp_Tres_Num_Iguais(r1, r2, r3)-> rt
1: faca  rt = Comp_Dois_Num_Iguais(r1, r2) va_para 2
2: se T entao va_para 3 senao va_para 4
3: faca  rt = Comp_Dois_Num_Iguais(r1, r3) va_para 4
4: retorna
```

Quadro 45 - Programa que compara se três números naturais são iguais utilizando macros

Comentários de linha podem ser adicionados ao programa da seguinte forma: “--“ <comentário>, onde <comentário> é formado por uma seqüência finita de caracteres. Tudo o que estiver após os caracteres “--“ até a quebra de linha será ignorado na transformação do programa para instruções rotuladas compostas. O Quadro 46 apresenta um programa com comentários de linha.

```
-- este programa retorna o fatorial de um número natural...
-- r1 = função de entrada
-- r1 = função de saída
programa fatorial(r1) -> r1
1: faca rt = r1 va_para 2                          -- atribui o valor de r1 a rt
2: se T entao va_para 3 senao va_para 4            -- testa se rt = 0
3: faca r1 = 1 va_para 9                           -- atribui o valor 1 a rt
4: faca r2 = r1 va_para 5                          -- atribui o valor de r1 a r2
5: faca dec(r2) va_para 6                          -- decrementa o valor de r2
6: faca rt = r2 va_para 7                          -- atribui o valor de r2 a rt
7: se T entao va_para 9 senao va_para 8            -- testa se rt = 0
8: faca r1 = Mult_Int_SemSinal(r1, r2) va_para 5   -- r1 = r1 x r2
9: retorna                                         -- programa terminou
```

Quadro 46 - Programa que retorna o fatorial de um número com comentários de linha

3.3 ESPECIFICAÇÃO FORMAL DA LINGUAGEM

A especificação da linguagem foi feita utilizando a notação BNF. Para a especificação, utilizou-se a ferramenta GALS, que é uma ferramenta de *software* livre (GALS, 2003), a qual gerou todas as classes do analisador léxico, sintático e um "esqueleto" do analisador semântico.

No Quadro 47 são mostradas as definições regulares da linguagem, que são utilizadas na definição dos símbolos terminais ou constantes, onde:

a) Letra: é uma letra do alfabeto, podendo ser maiúscula ou minúscula;
b) Dig: é um número natural de zero a nove;
c) Ignorar: é a especificação para que possam ser ignorados os caracteres não imprimíveis, como o espaço em branco, quebra de linha, entre outros;
d) Comentário: é a especificação do formato de um comentário de linha, onde um comentário deve ser iniciado com "--" seguido por qualquer caractere da tabela ASC II até a quebra de linha.

```
Letra : [A-Za-z]
Dig   : [0-9]
Ignorar : [\ \t\n\r]
Comentario :  "--" (.)*
```

Quadro 47 - Definições regulares

No Quadro 48 é mostrada a lista de símbolos terminais (*tokens*) da linguagem, onde:

a) ": {Ignorar}*": significa que o ambiente deve ignorar todos os espaços em branco, tabulações e quebra de linha;

b) ":! {Comentario}": significa que o ambiente deve ignorar os comentários de linha definidos no campo "definições regulares";

c) "NumeroNatural": um ou mais dígitos (0 a 9);

d) "Registrador": deve iniciar com a letra "r" ou "r", seguido por:

- um ou mais dígitos (números de 0 a 9),
- "t" ou "T";

f) "Pr": é a definição de uma palavra reservada, que deve ser iniciada por uma letra, seguida por uma seqüência de letras, dígitos ou pelo caractere "_";

```
: {Ignorar}*
:! {Comentario}
NumeroNatural : {Dig}+
Registrador : ("R" | "r") {Dig}+ | ("R" | "r") ("T" | "t")

Pr : {Letra} ({Letra} | _ | {Dig})*
Programa    = Pr : "PROGRAMA"
Faca        = Pr : "FACA"
Va_Para     = Pr : "VA_PARA"
Se          = Pr : "SE"
Entao       = Pr : "ENTAO"
Senao       = Pr : "SENAO"
Retorna     = Pr : "RETORNA"
Inc         = Pr : "INC"
Dec         = Pr : "DEC"
T           = Pr : "T"

"="
":"
","
"("
")"
"->"
```

Quadro 48 - Lista de símbolos terminais (*Tokens*)

No Quadro 49 são apresentadas as produções da gramática definida para a linguagem criada com as ações semânticas, utilizando a notação BNF, onde: uma ação semântica na gramática é iniciada pelo caractere "#" seguida por um número natural. A ação semântica #1 por exemplo, é responsável por criar uma instrução de cabeçalho e inicializá-la com o nome do programa. No Apêndice A, são apresentadas todas as ações semânticas da linguagem.

```
<Cabecalho>          ::=  Programa <Nome_Programa> <Funcao_Entrada> "->"#2
                          <Funcao_Saida>#3 <Monolitico>;

<Nome_Programa>      ::=  Pr#1;

<Funcao_Entrada>     ::=  "(" <Registradores> ")" | ε;

<Funcao_Saida>       ::=  <Registradores>;

<Registradores>      ::=  Registrador#4 <R_Registradores>;

<R_Registradores>    ::=  "," Registrador#4 <R_Registradores> | ε;

<Monolitico>         ::=  <Monolitico_L> <Monolitico> | ε;

<Monolitico_L>       ::=  <Instrucao_SE> | <Instrucao_FACA> | <Instrucao_Retorna>;

<Instrucao_SE>       ::=  <Rotulo> ":" Se#6 T entao va_para <Rotulo_Desv_Verd>
                          senao va_para <Rotulo_Desv_Falso>;

<Rotulo>             ::=  NumeroNatural#5;

<Rotulo_Desv_Verd>   ::=  NumeroNatural#17;

<Rotulo_Desv_Falso>  ::=  NumeroNatural#18;

<Instrucao_FACA>     ::=  <Rotulo> ":" Faca#6 <Operacao> va_para <Rotulo_Desvio>;

<Rotulo_Desvio>      ::=  NumeroNatural#16;

<Operacao>           ::=  <INC>#7 | <DEC>#7 | <Atribuicao> ;

<INC>                ::=  Inc "(" Registrador#8 ")";

<DEC>                ::=  Dec "(" Registrador#8 ")";

<Atribuicao>         ::=  Registrador#9 <R_Regist> = <R_Atribuicao>;

<R_Regist>           ::=  "," Registrador#10 <R_Regist> | ε;

<R_Atribuicao>       ::=  <Registrador> | <Constante> | <Macro>;

<Registrador>        ::=  #11Registrador#13;

<Constante>          ::=  #12NumeroNatural#13;

<Macro>              ::=  Pr#14 <Funcao_Entrada>#15;

<Instrucao_Retorna>  ::=  <Rotulo> ":" Retorna#6;
```

Quadro 49 - Produções da gramática utilizando a notação BNF

3.4 ESPECIFICAÇÃO DO AMBIENTE

A especificação do ambiente foi feita utilizando a técnica de análise orientada a objetos, representando-a através dos diagramas de casos de uso, de classes e de seqüência da UML. Para a especificação dos diagramas utilizou-se a ferramenta Rational Rose, a qual é uma ferramenta *Computer Aided Software Engineering* (CASE), que possibilita a modelagem visual de aplicações de software utilizando o padrão UML.

3.4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O ambiente desenvolvido deverá obedecer os seguintes requisitos:

a) editor: a ferramenta deverá ter um editor para programas monolíticos (requisito funcional – RF);
b) compilador: a ferramenta deverá possibilitar que um programa monolítico descrito na forma de instruções rotuladas seja compilado, verificando e apresentando a existência de erros léxicos e sintáticos (RF);
c) conversão: a ferramenta deverá possibilitar a conversão de um programa monolítico descrito na forma de instruções rotuladas em um programa monolítico na forma de instruções rotuladas compostas (RF);
d) simplificação: a ferramenta deverá mostrar a simplificação do programa, mostrando este antes e após a mesma (RF);
e) propriedades: a ferramenta deverá disponibilizar um módulo para verificar a existência de instruções mortas no programa monolítico, e um módulo para verificar a equivalência entre dois programas monolíticos. A aplicação destas propriedades em cima da estrutura estática do programa deverá ser mostrada através de uma janela (RF);
f) interpretação: a ferramenta deverá mostrar em uma janela o resultado da execução do programa monolítico (RF);
g) interpretação passo-a-passo: a ferramenta deverá possibilitar a execução passo-a-passo de um programa monolítico (RF);
h) registradores: a ferramenta deverá possibilitar a visualização dos valores dos registradores durante a execução através de uma janela (RF);
i) terminar execução: a ferramenta deverá possibilitar que a execução de um programa seja interrompida (RF);
j) instruções rotuladas: a ferramenta deverá ter uma janela para mostrar o programa monolítico transformado em instruções rotuladas compostas (RF);
k) interface: a ferramenta deverá ter uma interface amigável, com menus padrão Windows (requisito não-funcional – RNF);
l) confiabilidade: a ferramenta deverá apresentar resultados equivalentes aos obtidos através do processo manual (RNF);
m) menus: o editor da ferramenta deverá possuir menus para abrir, salvar, fechar, imprimir, compilar, transformar, executar e verificar propriedades de um programa

monolítico na forma de instruções rotuladas. Também deverá ter menus rápidos (botões na interface) para copiar, colar e recortar partes do programa fonte, assim como para inserir modelos de instruções de operação e teste (RNF).

3.4.2 DIAGRAMAS DE CASOS DE USO

A Tabela 3 descreve os casos de uso apresentados na Figura 13.

Tabela 3 - Descrição dos casos de uso do ambiente

Caso de Uso	**Descrição**
Abrir programa	O programador abre um programa existente através do comando Abrir.
Novo programa	O programador cria um novo programa através do comando Novo.
Salvar programa	Através do comando Salvar o programador salva as alterações efetuadas no programa fonte.
Imprimir programa	O programador pode imprimir o programa através do comando Imprimir.
Inserir modelo de instrução de operação	O programador pode inserir um modelo de instrução de operação, na linha corrente do editor, através o comando “Incluir instrução Faça”.
Inserir modelo de instrução de teste	O programador pode inserir um modelo de instrução de teste, na linha corrente do editor, através do comando “Incluir instrução Se”
Compilar programa	O programador compila o programa fonte através do comando Compilar. O ambiente emite uma mensagem informando se a compilação terminou com sucesso ou se houveram erros.
Transformar programa	O programador pode transformar o programa monolítico na forma de instruções rotuladas em instruções rotuladas compostas, através do comando Transformar, no menu Compilar.
Executar programa	O programador pode executar o programa na forma de instruções rotuladas compostas, através do comando Executar no menu Compilar.
Executar programa passo-a-passo	O programador pode executar passo-a-passo o programa na forma de instruções rotuladas compostas, através do comando “Executar programa passo-a-passo” no menu Compilar.
Terminar execução do programa	O programador pode interromper a execução de um programa através do comando “Parar execução” no menu Compilar.
Verificar instruções mortas do programa	O programador pode verificar as instruções mortas no programa, através do comando “Localizar rótulos mortos” no menu Compilar.
Verificar equivalência com outro programa	O programador pode verificar a equivalência do programa corrente com outro programa, através do comando “Verificar Equivalência” no menu Compilar.

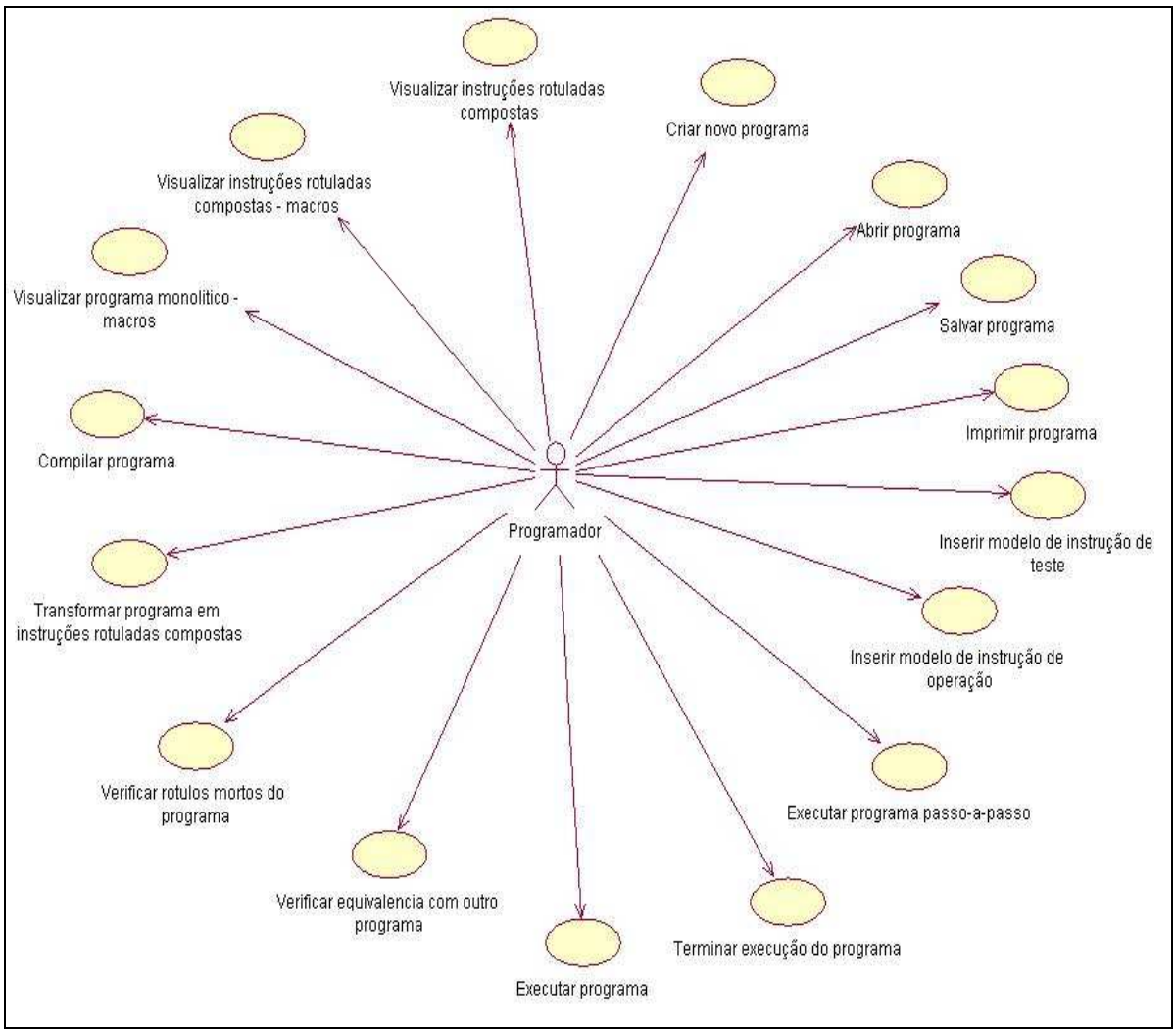


Figura 13 - Diagrama de casos de uso

3.4.3 DIAGRAMA DE CLASSES

A Figura 14 apresenta o diagrama de classes do ambiente.

As classes apresentadas na Figura 15 são responsáveis pela emissão de mensagens ao usuário quando ocorrer algum erro léxico, sintático ou semântico. Se ocorrer algum erro léxico, a classe ELexicalError será instanciada, se ocorrer um erro sintático, a classe ESyntaticError será instanciada e se ocorrer um erro semântico, a classe ESemanticError será instanciada. As classes ELexicalError, ESyntaticError e ESemanticError são especializações da classe EAnalysisError, a qual possui dois construtores de mensagens: um cria somente uma mensagem com o erro ocorrido e o outro cria uma mensagem com o erro ocorrido e mais a posição onde ocorreu o erro. Estas classes foram geradas automaticamente pelo GALS.

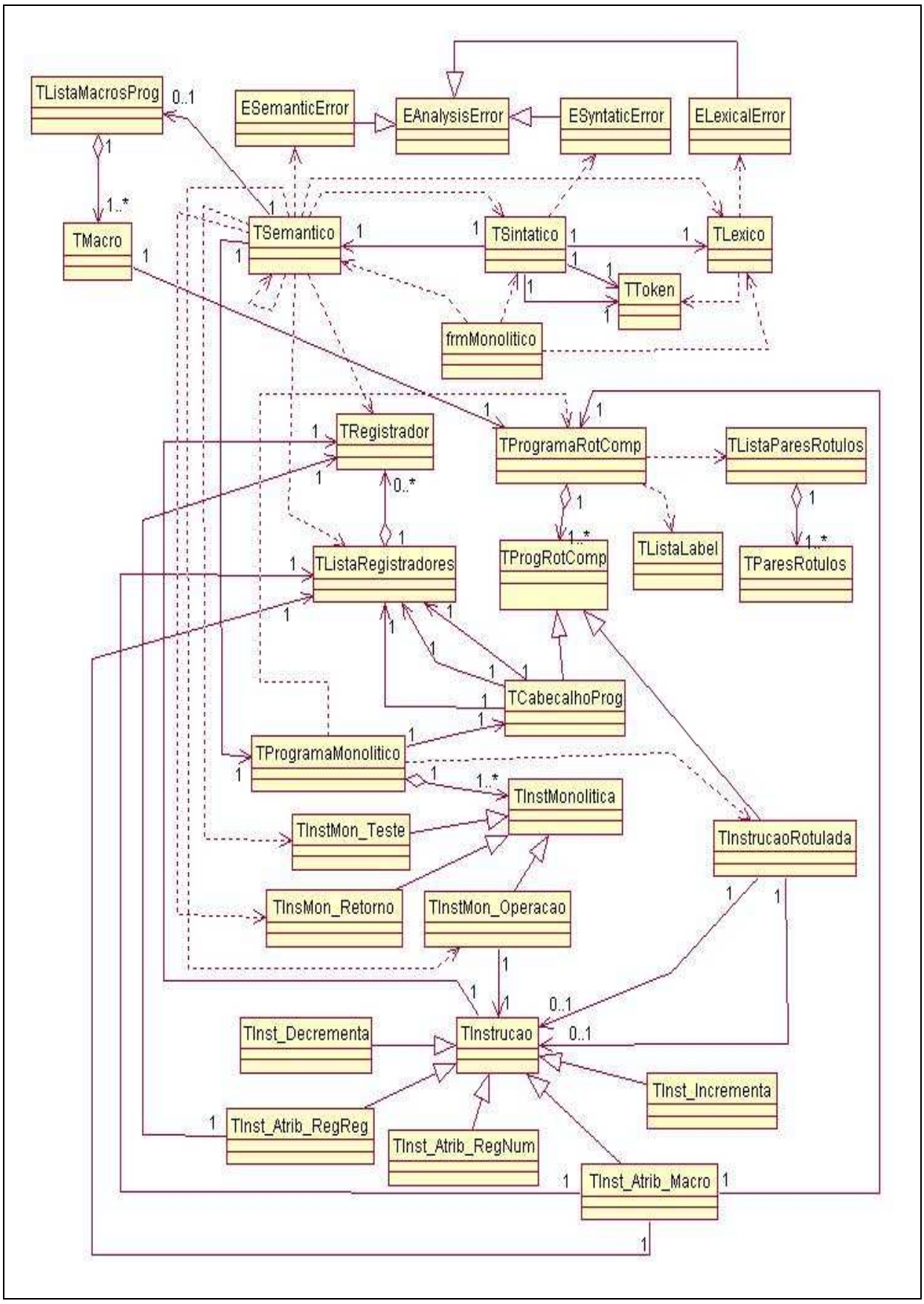


Figura 14 - Diagrama de classes

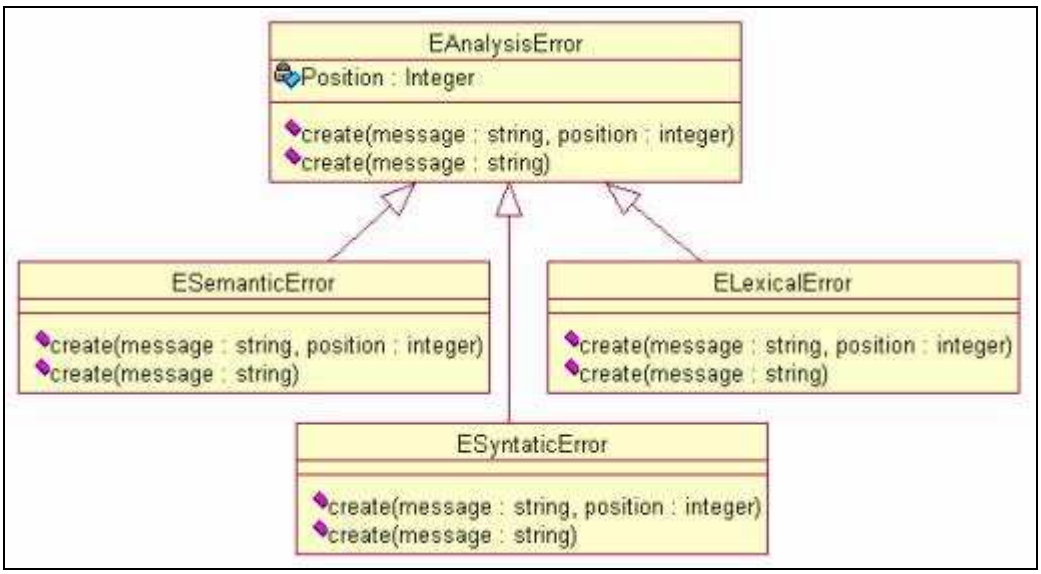


Figura 15 - Classe EAnalysisError e suas especializações

A Figura 16 apresenta as classes TLexico, TSintatico, TToken (geradas automaticamente pelo GALS), TSemantico, TListaMacroProg, TMacro e frmMonolitico. A classe TLexico é responsável pela análise léxica, a TSintatico pela análise sintática e a TSemantico pela análise semântica.

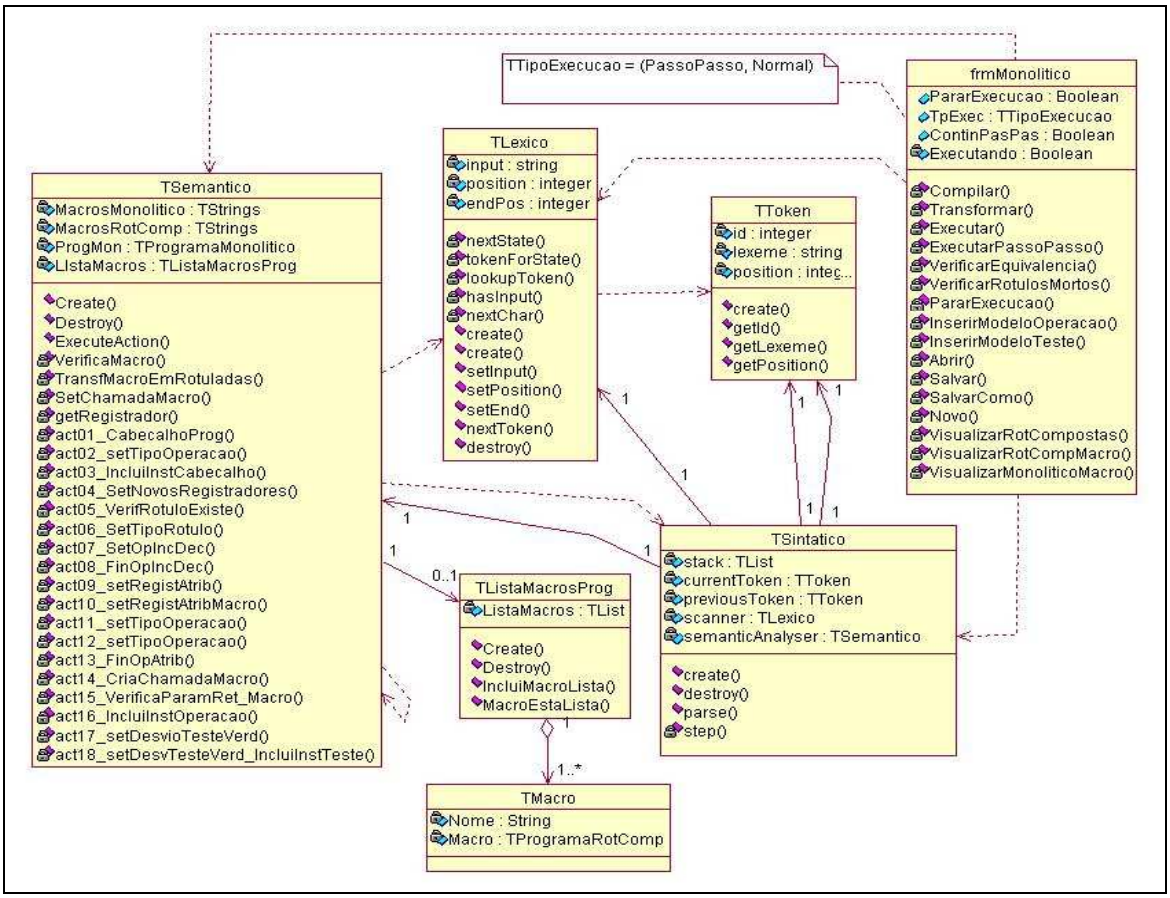


Figura 16 - Classes dos analisadores Léxico, Sintático e Semântico

A classe frmMonolitico é a classe da interface do ambiente, a qual o usuário interage. O método Compilar (Quadro 50) desta classe é responsável pela transformação de um programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas. Ele cria uma instância de cada analisador, e posteriormente chama o método *parse* da classe TSintatico, passando como parâmetro os analisadores léxico e semântico instanciados anteriormente. O método *parse* irá chamar a classe TLexico através do método *nextToken*, o qual pega o próximo *token* (símbolo terminal) a ser analisado. Se houver uma ação semântica associada a este *token*, o método *executeAction* da classe TSemantico é chamado, passando como parâmetro o número da ação semântica e o *token* associado. O método *executeAction* irá chamar outros métodos, que são as ações semânticas, as quais serão executadas conforme o número da ação semântica recebida como parâmetro. Este processo irá se repetir até o final da análise.

O atributo ListaMacros da classe TSemantico guarda as macros utilizadas no programa monolítico principal em uma estrutura do tipo TListaMacrosProg. Uma macro é criada uma única vez em um programa. Se um programa utilizar mais de uma vez a mesma macro, esta será buscada na lista de macros e será reutilizada.

A classe TSemantico é dependente dela mesma e das classes TLexico e TSintatico. Quando um programa possui uma chamada de macro, uma ação semântica criará novas instâncias de cada analisador, e fará a análise da macro. Após terminada a análise da macro, o programa monolítico principal continua sendo analisado.

Conforme o programa é analisado, ele é remontado em uma estrutura do tipo TProgramaMonolitico (Figura 18), que será utilizado para a transformação do programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas. Esta estrutura é guardada no atributo ProgMon da classe TSemantico.

A Figura 17 apresenta a classe TListaRegistradores, que pode guardar vários registradores do tipo TRegistrador em uma lista. Esta lista de registradores será utilizada por várias classes, conforme apresentado do diagrama de classes (Figura 14).

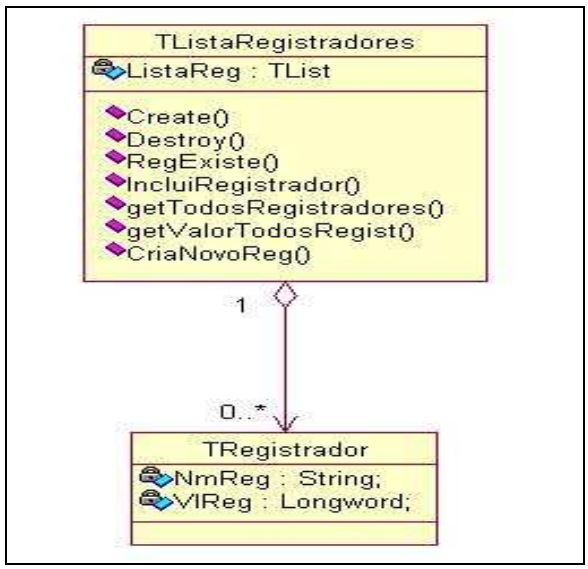


Figura 17 - Classe TListaRegistradores

A Figura 18 apresenta a classe TProgramaMonolitico. Esta classe irá guarda em uma lista, um cabeçalho do programa e várias instruções rotuladas. Sempre a primeira instrução da lista será um cabeçalho do tipo TCabecalhoProg e as próximas serão do tipo TInstMonolitica. A classe TProgramaMonolitico possui dois métodos principais os quais são VerifRotulosMortos e TransfMon_RotComp. O método VerifRotulosMortos verifica a existência de instruções mortas no programa monolítico na forma de instruções rotuladas e o método TransfMon_RotComp transforma o programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas, retornando uma estrutura do tipo TProgramaRotComp, que é apresentada na Figura 20.

A classe TInstMonolitica possui três classes mais especializadas, as quais são TInstMon_Teste, TInstMon_Operacao e TInsMon_Retorno. A classe especializada TInstMon_Operacao possui o atributo Instrucao, que será utilizada para a execução do programa.

A classe TCabecalhoProg tem por objetivo guardar informações como o nome do programa, os registradores utilizados no programa, os parâmetros do programa (função de entrada) e o retorno do programa (função de saída).

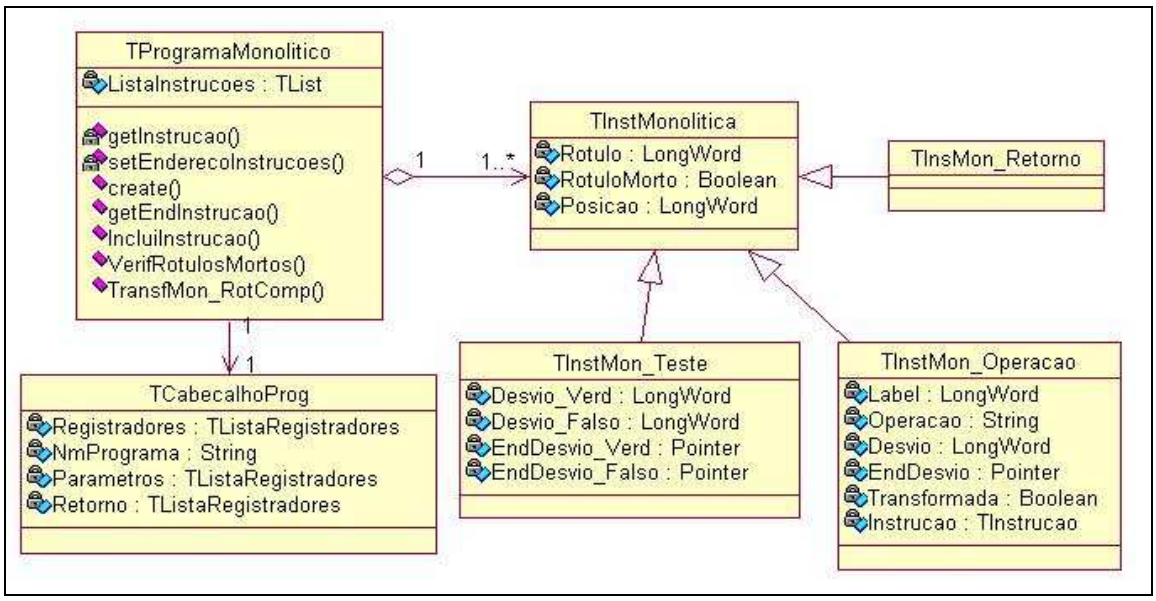


Figura 18 - Classe TProgramaMonolitico

A classe TInstrucao apresentada na Figura 19 possui cinco classes mais específicas. Estas classes são criadas durante a análise semântica. Quando uma instrução rotulada do tipo operação é encontrada, o analisador semântico cria uma instrução executável (TInstrucao) de acordo com a operação encontrada (incrementa, decrementa, atribuição de um número natural a um registrador, atribuição de um registrador a outro ou atribuição de uma macro a um ou mais registradores). As instruções executáveis (TInstrucao) criadas serão utilizadas quando um programa for executado. Elas serão chamadas através do método executaInstrucao, que é um método abstrato da classe TInstrucao e que é sobrescrito pelos métodos executaInstrucao das classes mais especializadas. O código do método executaInstrucao é apresentado no Quadro 53.

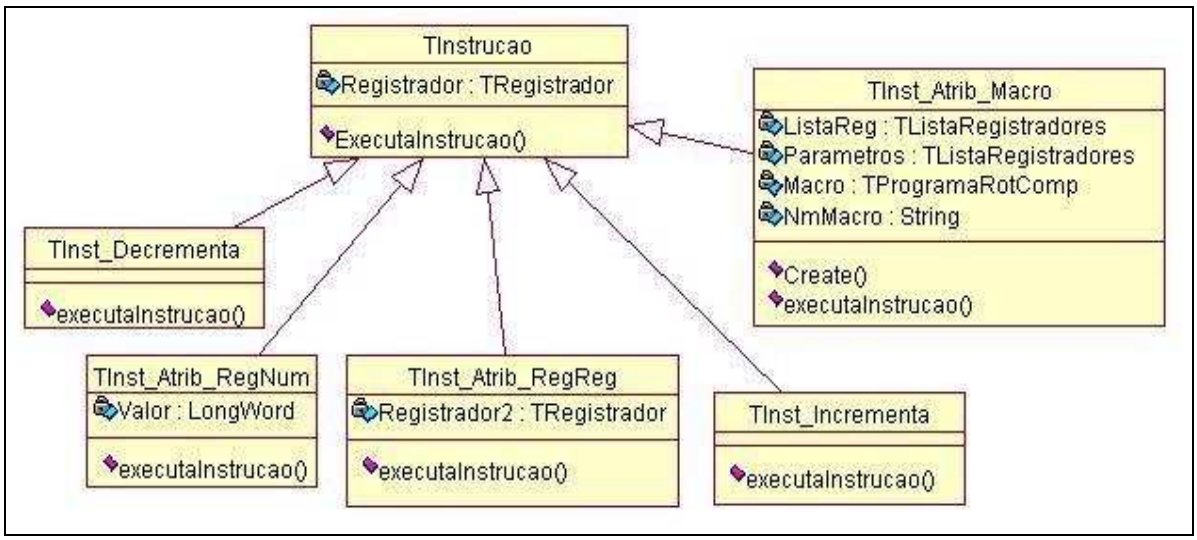


Figura 19 - Classe TInstrucao

A classe TProgramaRotComp apresentada na Figura 20 possui três métodos principais: SimpRotComp, que simplifica o programa na forma de instruções rotuladas compostas; VerifEquiv, que verifica a equivalência com outro programa e ExecutaPrograma, que executa o programa, retornando a função de saída (valor dos registradores de retorno). Esta classe guarda em uma lista, várias instâncias da classe TProgRotComp, que por sua vez possui duas classes mais especializadas as quais são: TCabecalhoProg e TInstrucaoRotulada. O cabeçalho do programa é somente copiado da classe TProgramaMonolitico quando o programa é transformado da forma de instruções rotuladas para instruções rotuladas compostas. A classe TInstrucaoRotulada é instanciada tantas vezes quantas necessárias, durante a transformação do programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas.

A classe TListaParesRotulos guarda em uma lista, vários pares de rótulos de dois programas monolíticos. Esta classe é utilizada para a verificação da equivalência entre dois programas monolíticos.

A classe TListaLabel guarda os rótulos de um programa em um lista. Esta classe é utilizada para a simplificação do programa na forma de instruções rotuladas compostas.

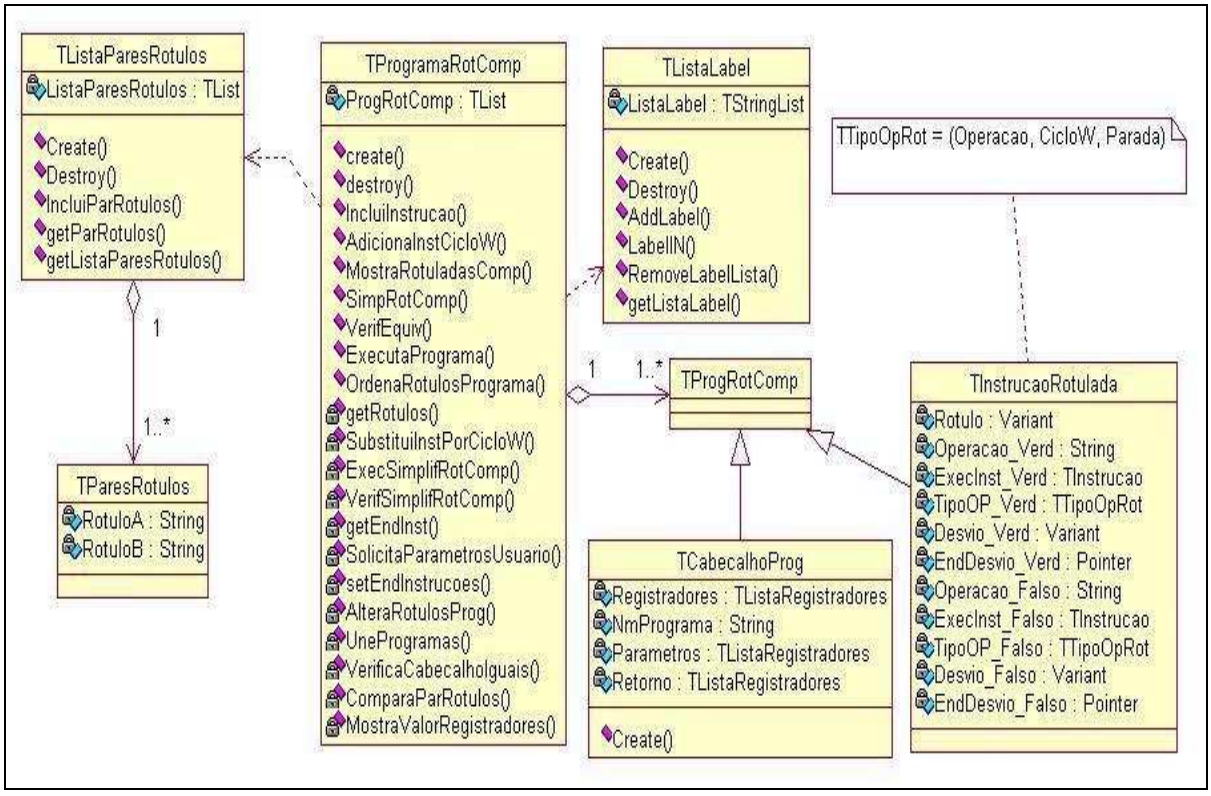


Figura 20 - Classe TProgramaRotComp

3.4.4 DIAGRAMAS DE SEQUÊNCIA

A Figura 21 apresenta o diagrama de seqüência do método “Compilar”, o qual compila o programa na forma de instruções rotuladas, verificando a existência de erros léxicos, sintáticos e semânticos.

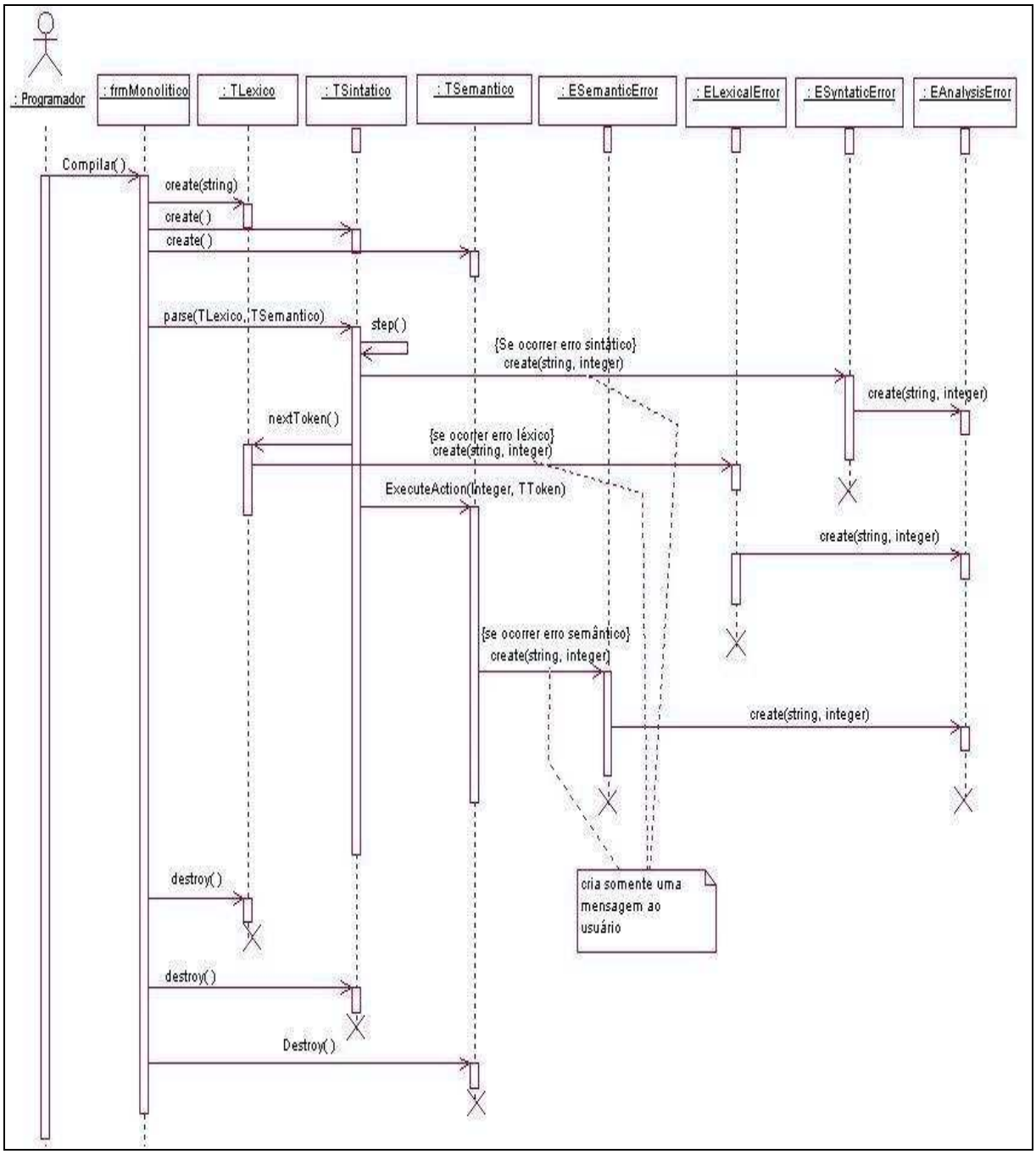


Figura 21 - Diagrama de seqüência do método "Compilar"

A Figura 22 apresenta o diagrama de seqüência do método “VerificarRotulosMortos”, o qual verifica a existência de instruções mortas no programa monolítico na forma de instruções rotuladas.

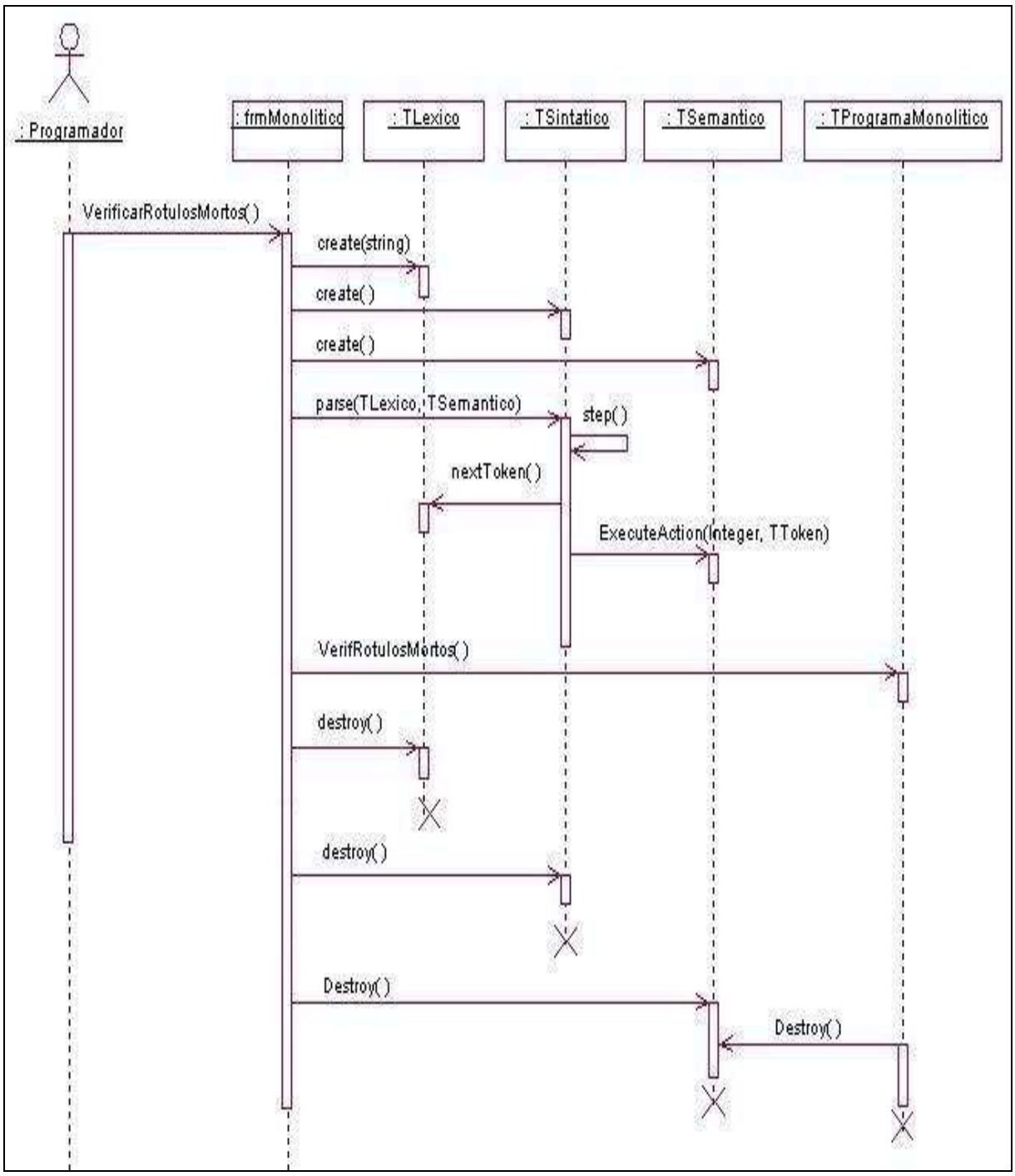


Figura 22 - Diagrama de seqüência do método “VerificarRotulosMortos”

A Figura 23 apresenta o diagrama de seqüência do método “Transformar”, o qual transforma um programa monolítico na forma de instruções rotuladas em instruções rotuladas compostas.

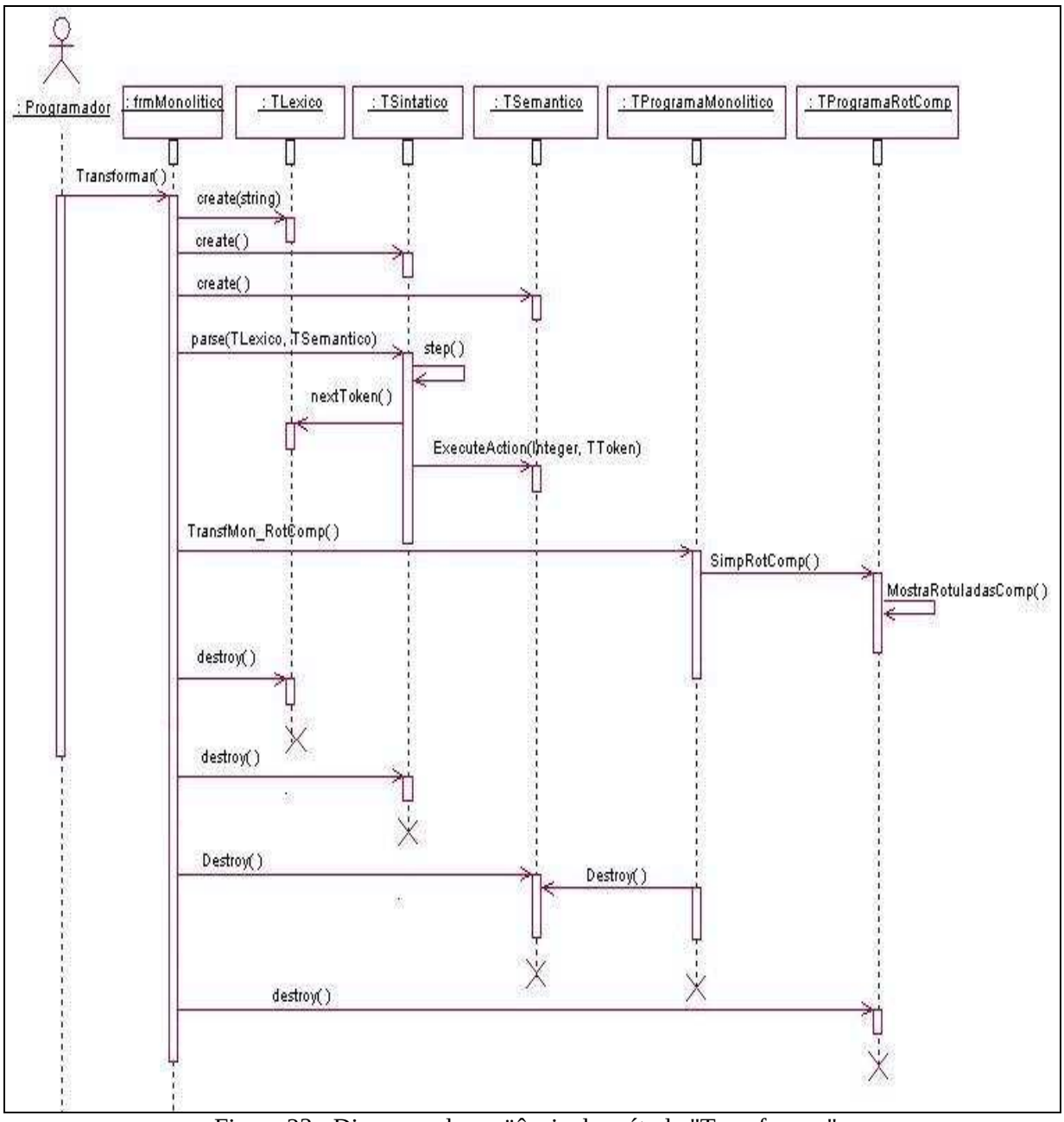


Figura 23 - Diagrama de seqüência do método "Transformar"

A Figura 24 apresenta o diagrama de seqüência do método “VerificarEquivalencia”, o qual verifica a equivalência entre dois programas monolíticos na forma de instruções rotuladas compostas.

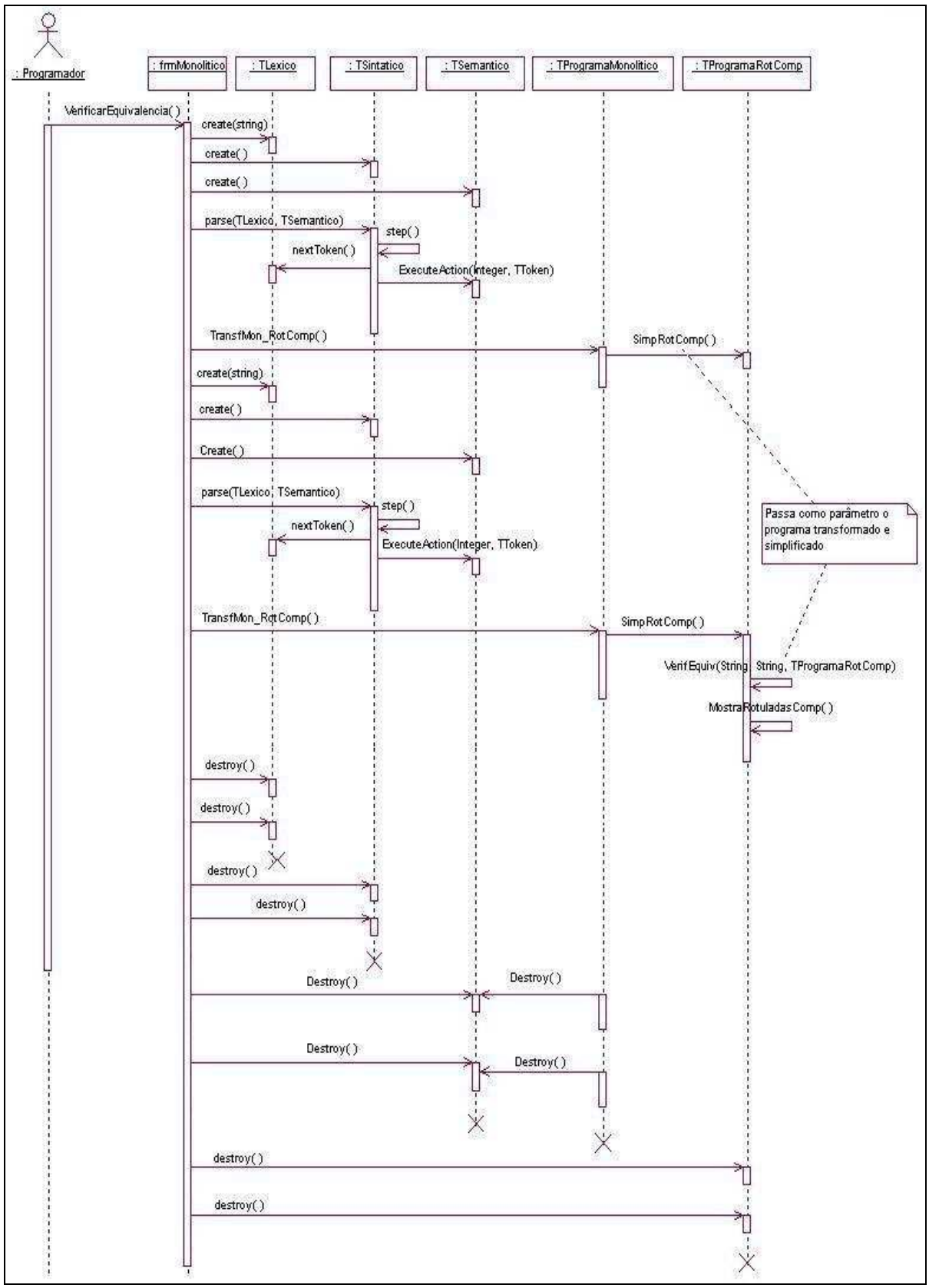


Figura 24 - Diagrama de seqüência do método "VerificarEquivalencia"

A Figura 25 apresenta o diagrama de seqüência do método “Executar”, o qual executa o programa monolítico já transformado em instruções rotuladas compostas.

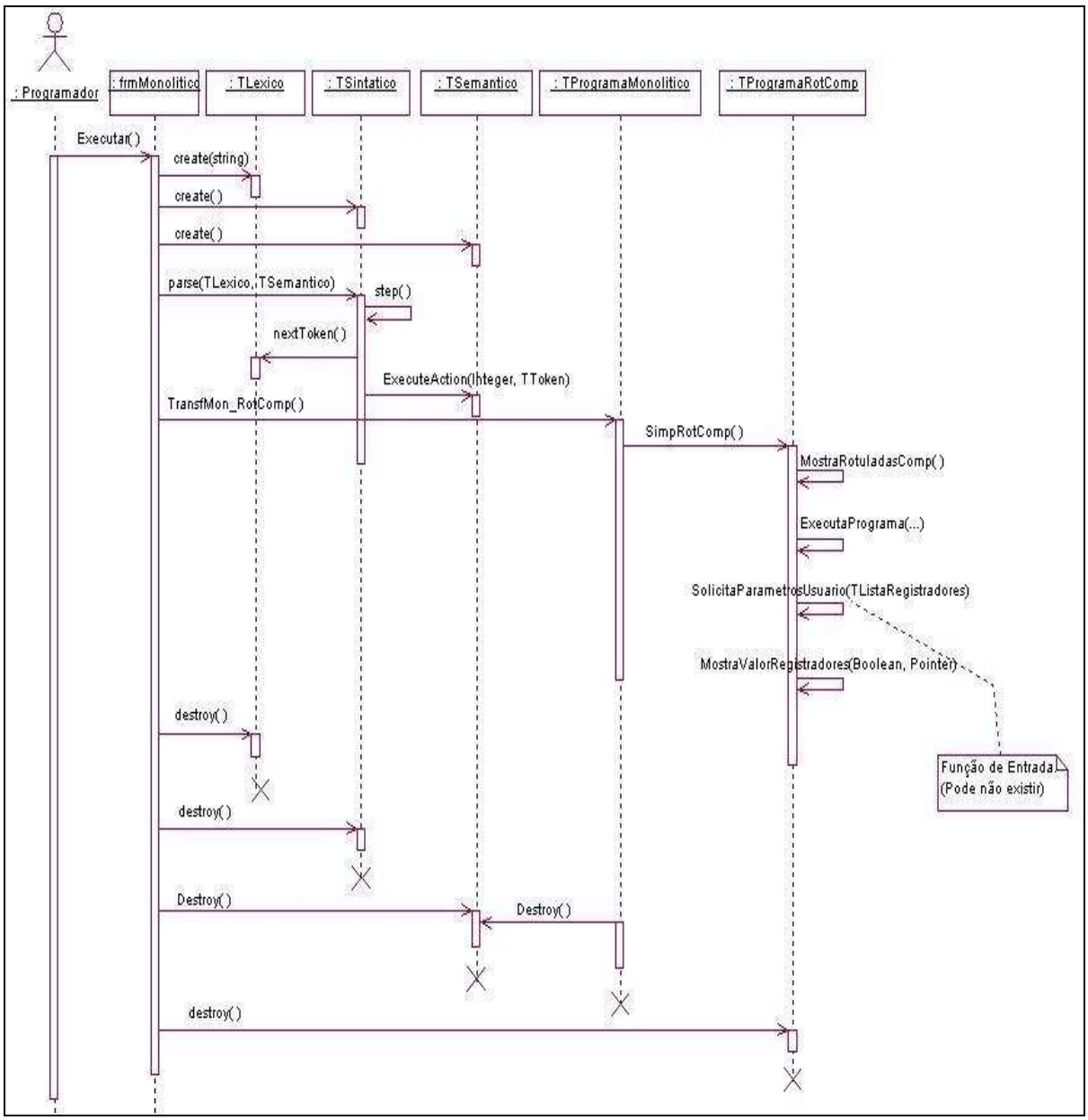


Figura 25 - Diagrama de seqüência do método "Executar"

Os diagramas de seqüência apresentados nas Figuras 22, 23, 24 e 25 possuem o mesmo tratamento de erros do diagrama da Figura 21.

3.5 IMPLEMENTAÇÃO

Após a geração das classes para análise léxica e sintática através do GALS, foi implementada a integração entre o código gerado pela ferramenta GALS em *Object Pascal*, com o código do ambiente. Após cada analisador ser instanciado, o método *parse* da classe sintático é chamado, verificando se existem erros e tratando-os. Após esta verificação, pode-

se transformar o programa monolítico na forma de instruções rotuladas em instruções rotuladas compostas. Esta interação é apresentada no Quadro 50.

```
procedure TfrmMonolitico.acTransformarExecute(Sender: TObject);
var
  vLexico : TLexico;
  vSintatico : TSintatico;
  vSemantico : TSemantico;
  vInstRotuladas : TProgramaRotComp;
  vRotuladasStr,
  vRotSimplificado : String;
begin
  // limpa area de mensagens
  StatusBar.Panels[2].Text:=EmptyStr;
  // limpa janelas das macros e das inst. rot. comp.
  LimpaJanelas;
  // cria uma instancia para cada analisador
  vLexico := TLexico.create(reEditor.Text); //o parametro eh o programa mon. fonte
  vSintatico := TSintatico.create;
  vSemantico := TSemantico.Create;
  try
    // executa a analise lexica / sintatica e semantica
    vSintatico.parse(vLexico, vSemantico);
    // executa a transformacao do prog. mon. para inst. rot. comp.
    vInstRotuladas := vSemantico.ProgMon.TransfMon_RotComp;
    vRotuladasStr := vInstRotuladas.MostraRotuladasComp;
    reRotuladasComp.Lines.Add(vRotuladasStr);
    // simplifica o programa
    vRotSimplificado := vInstRotuladas.SimpRotComp.MostraRotuladasComp;
    // verifica se o programa foi simplificado
    if vRotSimplificado <> vRotuladasStr then
      begin
        reRotuladasComp.Lines.Add(#13+'******SIMPLIFICADO********');
        reRotuladasComp.Lines.Add(vRotSimplificado);
      end
    else
      reRotuladasComp.Lines.Insert(0, '***PROGRAMA JÁ ESTÁ SIMPLIFICADO***');
    // se existirem macros no programa, mostra o seu cod. fonte
    if vSemantico.MacrosRotComp <> Nil then
      begin
        reMonolitico_Macro.Lines := vSemantico.MacrosMonolitico;
        rerotuladas_Macro.Lines:= vSemantico.MacrosRotComp;
      end;
    // atualiza o estado das janelas
    AtualizaJanelas;
    // se for um comando para executar o prog., executa-o
    if (Sender = acExecutar) or (Sender = acExecPassoPasso)then
      ExecutaRotuladasCompostas(Sender, vInstRotuladas.SimpRotComp);
  except // tratamento de excecoes
    on e : ELexicalError do // MOSTRA ERRO LEXICO
      StatusBar.Panels[2].Text:=Format(sErroLinha, [getLinha(e.getPosition), e.getMessage]);
    on e : ESyntaticError do // MOSTRA ERRO SINTATICO
      StatusBar.Panels[2].Text:=Format(sErroLinha, [getLinha(e.getPosition), e.getMessage]);
    on e : ESemanticError do // MOSTRA ERRO SEMANTICO
      StatusBar.Panels[2].Text:=Format(sErroLinha, [getLinha(e.getPosition), e.getMessage]);
  end;
  if not fExecutando then // se prog. nao estiver executando libera mem. alocada
    begin
      vInstRotuladas.Free;
      vLexico.Free;
      vSintatico.Free;
      vSemantico.Free;
    end;
end;
```

Quadro 50 - Interação entre código gerado pelo GALS e código do ambiente

O Quadro 51 apresenta o código do método que executa a transformação do programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas.

```
function TProgramaMonolitico.TransfMon_RotComp: TProgramaRotComp;
var
  vLabels, vLabelsExecutadas : TListaLabel ;
  vInstrucaoRot : TInstrucaoRotulada;
  vEnd_Aux : TInstMonolitica;
  vProgramaSaida : TProgramaRotComp;
  vInstAtual : Pointer;
  vCicloW, vDesvOpOp : Boolean;
begin
  setEnderecoInstrucoes; // DEFINE PARA CADA INSTRUCAO OS ENDERECOS DOS DESVIOS
  vCicloW := False;
  vDesvOpOp:= False;
  vProgramaSaida := TProgramaRotComp.Create;// CRIA O PROG. MON. NA FORMA DE INST. ROT. COMP.
  vProgramaSaida.IncluiInstrucao(fListaInstrucoes[0]); // INCLUI O CABECALHO DO PROGRAMA
  vLabels := TListaLabel.Create;           // LISTA DE ROTULOS QUE SERAO TRANSFORMADOS
  vLabelsExecutadas := TListaLabel.create; // LISTA DOS ROTULOS JA TRANSFORMADOS
  vLabels.AddLabel(1);                     // ROTULO INICIAL DO PROG. INST. ROT. COMP.
  vInstAtual := fListaInstrucoes[1];       // PRIMEIRA INSTRUCAO A SER CONVERTIDA
  while (vLabels.ListaLabels.Count > 0) do
    begin
      if TInstMonolitica(vInstAtual) is TInstMon_Operacao then
        begin
          if not TInstMon_Operacao(vInstAtual).Transf then
            begin // NOVA INSTRUCAO COM INST. VERDADEIRO = INST. FALSO
              NovaInstOperacao(vLabels.RemoveLabelLista, vInstrucaoRot, TInstMon_Operacao(vInstAtual));
              vLabelsExecutadas.AddLabel(vInstrucaoRot.Rotulo);
              vProgramaSaida.IncluiInstrucao(vInstrucaoRot);
              if not vLabelsExecutadas.LabelIN(TInstMon_Operacao(vInstAtual).Label_)
                vLabels.AddLabel(TInstMon_Operacao(vInstAtual).Label_);
              if (vDesvOpOp) and (vLabels.ListaLabels.Count > 0) then
                vInstAtual := getInstrucao(StrToInt(vLabels.ListaLabels[0]));
              vDesvOpOp := False;
            end
          else
            begin
              if TInstMonolitica(TInstMon_Operacao(vInstAtual).EndDesvio) is TInstMon_Operacao then
                begin
                  TInstMon_Operacao(TInstMon_Operacao(vInstAtual).EndDesvio).Transf := False;
                  vDesvOpOp := True;
                end;
              vInstAtual:= TInstMon_Operacao(vInstAtual).EndDesvio;
            end;
        end
      else if TInstMonolitica(vInstAtual) is TInstMon_Teste then
        begin
          vInstrucaoRot := TInstrucaoRotulada.Create;
          vInstrucaoRot.Rotulo := vLabels.RemoveLabelLista;
          vLabelsExecutadas.AddLabel(vInstrucaoRot.Rotulo);
          if TInstMon_Teste(vInstAtual).EndDesvio_Verd <> vInstAtual then
            begin // DEFINE A INSTRUCAO "VERDADEIRO"...
              vEnd_Aux := TInstMon_Teste(vInstAtual).EndDesvio_Verd ;
              setInstVerd_Op_Parada(vInstrucaoRot, vEnd_Aux, vLabels, vLabelsExecutadas);
            end
          else // ciclo W, desvio do teste eh para ele mesmo
            begin
              setInstVerd_CicloW(vInstrucaoRot, vInstAtual);
              vCicloW := TRUE;
            end;
          if TInstMon_Teste(vInstAtual).EndDesvio_Falso <> vInstAtual then
            begin // DEFINE A INSTRUCAO "FALSO"...
              vEnd_Aux := TInstMon_Teste(vInstAtual).EndDesvio_Falso ;
              setInstFalso_Op_Parada(vInstrucaoRot, vEnd_Aux, vLabels, vLabelsExecutadas);
            end
          else // ciclo W, desvio do teste eh para ele mesmo
            begin
              setInstFalso_CicloW(vInstrucaoRot, vInstAtual);
              vCicloW := TRUE;
            end;
          vProgramaSaida.IncluiInstrucao(vInstrucaoRot); // INCLUI INST. ROT. COMP.
          if vLabels.ListaLabels.Count > 0 then // SE HOUVER ROT. A TRANSF., PEGA O PRIMEIRO.
            vInstAtual := getInstrucao(StrToInt(vLabels.ListaLabels[0]));
        end
      else  // EH UMA INSTRUCAO DE PARADA (RETORNO)
        begin
          NovaInstParada(vLabels.RemoveLabelLista, vInstrucaoRot, vInstAtual);
          vLabelsExecutadas.AddLabel(vInstrucaoRot.Rotulo);
          vProgramaSaida.IncluiInstrucao(vInstrucaoRot); // INCLUI INST. ROT. COMP.
          if vLabels.ListaLabels.Count > 0 then // SE HOUVER ROT. A TRANSF., PEGA O PRIMEIRO.
            vInstAtual := getInstrucao(StrToInt(vLabels.ListaLabels[0]));
        end;
    end;
    if vCicloW then // VERIF. HOUVE INST. DE "CICLO W"...
      vProgramaSaida.AdicionaInstCicloW; // ADICINAO A INSTRUCAO "W:(CICLO, W), (CICLO,W)"
    vLabels.Free; // LIBERA MEMORIA ALOCADA
    vLabelsExecutadas.Free;
    Result := vProgramaSaida; // RETORNA O PROGRAMA MONOLITICO NA FORMA DE INST ROT COMPOSTAS
end;
```

Quadro 51 - Código fonte do método “TransfMon_RotComp”

O Quadro 52 apresenta o código do método “ExecutaPrograma”, o qual executa o programa na forma de instruções rotuladas compostas, até que seja encontrada uma instrução de “parada”, ou de “ciclo infinito”.

```
function TProgramaRotComp.ExecutaPrograma(var pNumRotTermino, pRotTermVerdFalso:String;pTipo:TTipoExec;
  var pContinua, pPararExecucao:Boolean;  var pTipoExec : TTipoExecucao): TTipoOpRot;
var
  vInstrucaoAtual : TInstrucaoRotulada;
  vRegTeste:TRegistrador;
  vParametros : TListaRegistradores;
  vContinua : Boolean;
begin
  setEndInstrucoes; // DEFINE PARA CADA INSTRUCAO OS ENDERECOS DOS DESVIOS
  // SE FOR A EXECUCAO DE UM PROGRAMA E SE EXISTIREM PARAMETROS, SOLICITA AO USUARIO
  if pTipo = exPrograma then
    begin
      vParametros := TListaRegistradores(TCabecalhoProg(fListaInstrucoes[0]).Parametros);
      if vParametros <> Nil then
        begin
          MostraValorRegistradores(False, fListaInstrucoes[0]); // MOSTRA O VALOR DE TODOS OS REGIST.
          // FICA EM LOOP AGUARDANDO ACAO DO USUARIO
          while Not(pContinua) and (pTipoExec = PassoPasso) and not(pPararExecucao) do
            Application.ProcessMessages;
          SolicitaParametrosUsuario(vParametros); // FUNCAO DE ENTRADA...
        end
    end;
  vRegTeste:=TRegistrador(TListaRegistradores(TCabecalhoProg(
             fListaInstrucoes[0]).Registradores.RegExiste('rt'))); // REGISTRADOR DE TESTE
  vInstrucaoAtual := fListaInstrucoes[1]; // PRIMEIRA INSTRUCAO QUE SERA EXECUTADA
  vContinua := vInstrucaoAtual <> Nil; // CONTINUA SE EXISTIR INSTRUCAO PARA EXECUTAR
  while vContinua do
    begin
      pContinua := False;
      MostraValorRegistradores((vRegTeste = Nil) or (vRegTeste.VlReg = 0),vInstrucaoAtual );
      // SE FOR EXECUÇÃO PASSO-A-PASSO FICA EM LOOP ESPERANDO ACAO DO USUARIO
      while Not(pContinua) and (pTipoExec = PassoPasso) and not(pPararExecucao) do
        Application.ProcessMessages;
      if (vRegTeste = Nil) or (vRegTeste.VlReg = 0) then //VERIF. QUAL INSTRUCAO IRA EXECUTAR,V/F
        begin
          if TInstrucaoRotulada(vInstrucaoAtual).TipoOperacao_Verd = Operacao then
            begin // EXECUTA A OPERACAO ASSOCIADA A INSTRUCAO "VERDADEIRO"
              TInstrucao(TInstrucaoRotulada(vInstrucaoAtual).ExecOP_Verd).ExecutaInstrucao;
              vInstrucaoAtual := TInstrucaoRotulada(vInstrucaoAtual).EndDesvio_Verd;
            end
          else // PROGRAMA TERMINOU EXECUCAO ...
            begin
              Result:= TInstrucaoRotulada(vInstrucaoAtual).TipoOperacao_Verd;
              pNumRotTermino := TInstrucaoRotulada(vInstrucaoAtual).Rotulo;
              pRotTermVerdFalso := 'Verdadeiro';
              vContinua := False;
            end
          else // REGISTRADOR DE TESTE NAO EXISTE OU SEU VALOR EH ZERO....
            begin
              if TInstrucaoRotulada(vInstrucaoAtual).TipoOperacao_Falso = Operacao then
                begin // EXECUTA A OPERACAO ASSOCIADA A INSTRUCAO "FALSO"
                  TInstrucao(TInstrucaoRotulada(vInstrucaoAtual).ExecOP_Falso).ExecutaInstrucao;
                  vInstrucaoAtual := TInstrucaoRotulada(vInstrucaoAtual).EndDesvio_Falso;
                end
              else
                begin // PROGRAMA TERMINOU EXECUCAO ...
                  Result:= TInstrucaoRotulada(vInstrucaoAtual).TipoOperacao_Falso;
                  pNumRotTermino := TInstrucaoRotulada(vInstrucaoAtual).Rotulo;
                  pRotTermVerdFalso := 'Falso';
                  vContinua := False;
                end;
            end;
          Application.ProcessMessages;
          if pPararExecucao then
            raise Exception.Create(sExecucaoInterrompida); // TERMINA A EXECUCAO
        end;
      MostraValorRegistradores(False,Nil); // MOSTRA VALOR DE TODOS OS REGISTRADORES UTILIZADOS
end;
```

Quadro 52 - Código fonte do método "ExecutaPrograma"

O método “ExecutaPrograma” (Quadro 52), quando encontrar uma instrução executável, ou seja, uma instrução diferente de “parada” ou de “ciclo infinito”, irá executar o método “executaInstrução” apresentado no Quadro 53. Este método irá executar a operação associada a instrução rotulada, a qual será somente uma das operações apresentadas como exemplos nos Quadros 38, 39, 40, 41, 42, 43.

```
procedure TInst_Incrementa.executaInstrucao;
begin
  inherited;
  fRegistrador.VlReg := fRegistrador.VlReg + 1;
end;
procedure TInst_Decrementa.executaInstrucao;
begin
  inherited;
  if fRegistrador.VlReg > 0 then
    fRegistrador.VlReg := fRegistrador.VlReg - 1;
end;
procedure TInst_Atrib_RegReg.executaInstrucao;
begin
  inherited;
  fRegistrador.VlReg := fRegistrador2.VlReg;
end;
procedure TInst_Atrib_RegNum.executaInstrucao;
begin
  inherited;
  fRegistrador.VlReg := fValor;
end;

procedure TInst_Atrib_Macro.executaInstrucao;
var
  vIndice : Integer;
  vCabecalhoMacro : TCabecalhoProg;
  vRotRet,
  vVerdFalso : String;
begin
  inherited;
  vCabecalhoMacro := fmacro.ProgRotComp[0];
  {INICIALIZA TODOS OS REGISTRADORES DA MACRO COM O VALOR "0"}
  for vIndice:= 0 to vCabecalhoMacro.Registradores.ListaReg.Count - 1 do
    TRegistrador(vCabecalhoMacro.Registradores.ListaReg[vIndice]).VlReg := 0;

  (*PASSAGEM DE PARAMETRO PARA A MACRO*)
  if vCabecalhoMacro.Parametros <> Nil then
    for vIndice := 0 to vCabecalhoMacro.Parametros.ListaReg.Count - 1 do
      TRegistrador(vCabecalhoMacro.Parametros.ListaReg[vIndice]).VlReg :=
        TRegistrador(fParametros.ListaReg[vIndice]).VlReg;

  {VERIFICA SE A EXECUCAO DA MACRO PAROU EM UMA INSTRUCAO DE CICLO W}
  if fMacro.ExecutaPrograma(vRotRet, vVerdFalso, exMacro,
    frmMonolitico.fContinPasPas, frmMonolitico.fPararExecucao,
    frmMonolitico.fTpExec) =  CicloW then
    Raise ESemanticError.create(Format(sTerminoCicloInfinitoMacro,
                                [vCabecalhoMacro.NmPrograma, vRotRet, vVerdFalso]));
  {RETORNO DA MACRO}
  for vIndice := 0 to fListaReg.ListaReg.Count - 1 do
    TRegistrador(fListaReg.ListaReg[vIndice]).VlReg :=
      TRegistrador(vCabecalhoMacro.Retorno.ListaReg[vIndice]).VlReg;
end;
```

Quadro 53 - Código fonte do método "ExecutaInstrução"

O Quadro 54 apresenta o código do método que verifica a equivalência entre dois programas monolíticos.

```
function TProgramaRotComp.VerifEquiv(var pRotTermA, pRotTermB : String;
  pPrograma:TProgramaRotComp) : TProgramaRotComp;
var
  vRotuloInicialA, vRotuloInicialB : String;
  vLstParRotulos, vLstparRotulosExec : TListaParesRotulos;
  vContinua : Boolean;
begin
  Self.UneProgramas(pPrograma, vRotuloInicialB);
  if not Self.VerificaCabecalhoIguais(pPrograma) then
    begin
      pRotTermA := '-1' ;
      pRotTermB := '-1' ;
    end
  else
    begin
      vRotuloInicialA := '1';
      vLstParRotulos := TListaParesRotulos.Create;
      vLstparRotulosExec := TListaParesRotulos.Create;
      vLstParRotulos.IncluiParRotulos(vRotuloInicialA, vRotuloInicialB);
      vContinua := True;

      while (vLstParRotulos.ListaParesRotulos.Count > 0) and vContinua do
        vContinua := ComparaParRotulos(vLstParRotulos,vLstparRotulosExec);

      if vContinua then
        begin
          pRotTermA := '0';
          pRotTermB := '0';
        end
      else
        begin
          pRotTermA := TParesRotulos(vLstParRotulos.ListaParesRotulos[0]).RotuloA;
          pRotTermB := TParesRotulos(vLstParRotulos.ListaParesRotulos[0]).RotuloB;
        end;
      vLstParRotulos.Free;
      vLstParRotulosExec.Free;
    end;
  Result := Self;
end;
```

Quadro 54 - Código fonte do método “VerifEquiv”

3.5.1 OPERACIONALIDADE DA IMPLEMENTAÇÃO

A Figura 26 apresenta a interface do ambiente com o usuário. Através desta interface, o usuário poderá criar novos programas monolíticos, abrir programas existentes, salvar, verificar a existência de instruções mortas, verificar equivalência com outro programa monolítico, compilar, transformar o programa em instruções rotuladas compostas, executar e executar passo-a-passo.

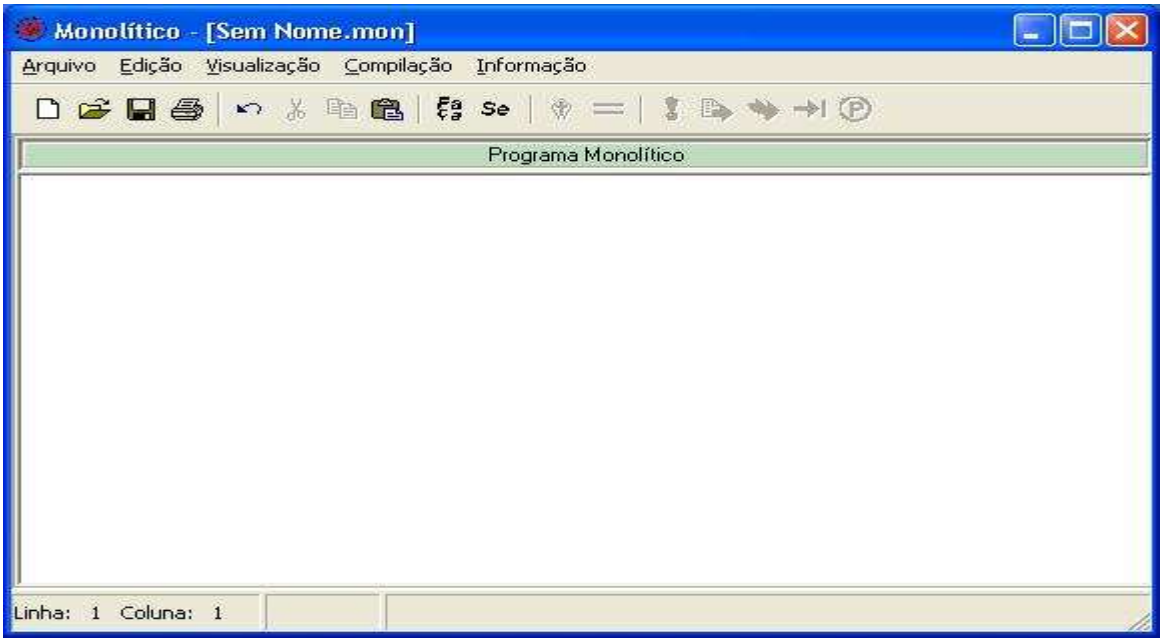


Figura 26 - Interface do ambiente

A Figura 27 apresenta um programa monolítico transformado, mostrando o programa monolítico na forma de instruções rotuladas e também na forma de instruções rotuladas compostas.

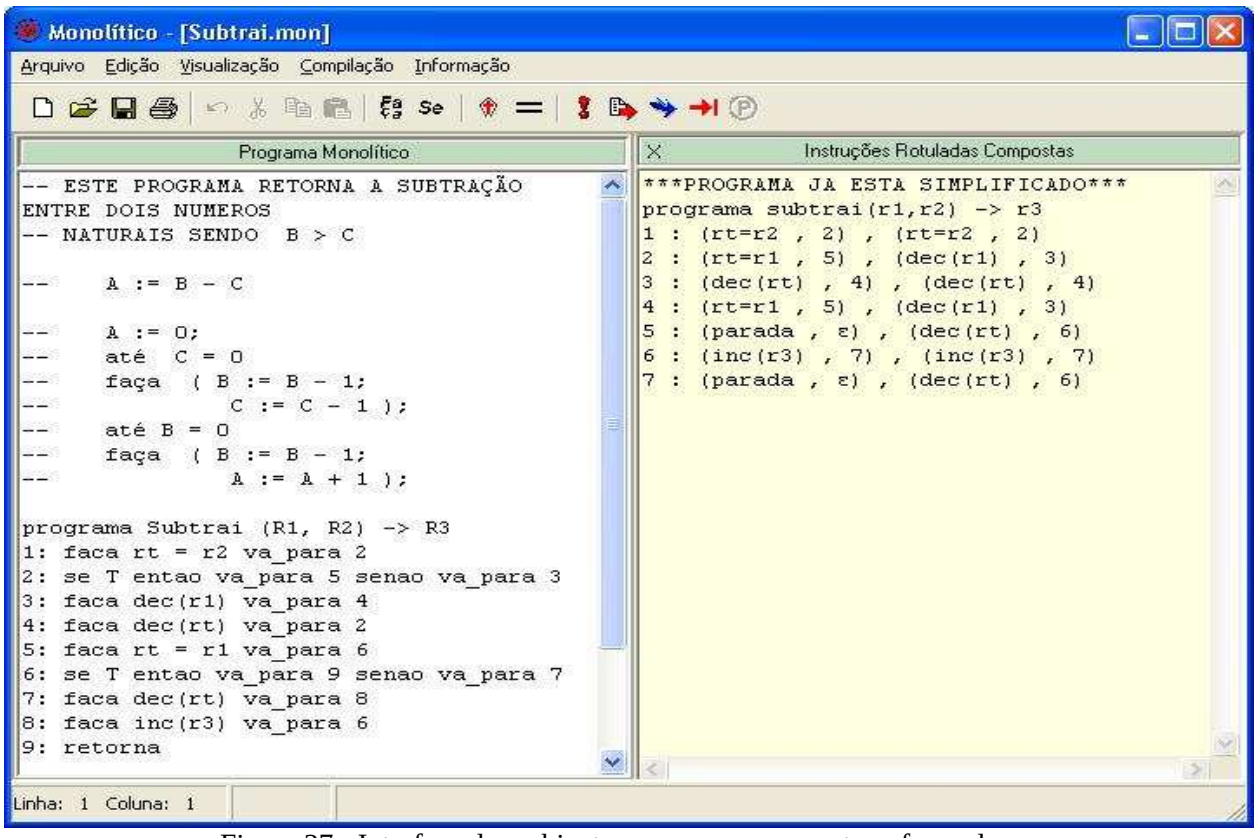


Figura 27 - Interface do ambiente com um programa transformado

A Figura 28 mostra o programa apresentado na Figura 32 sendo executado passo-a-passo, com a visualização dos registradores do programa.

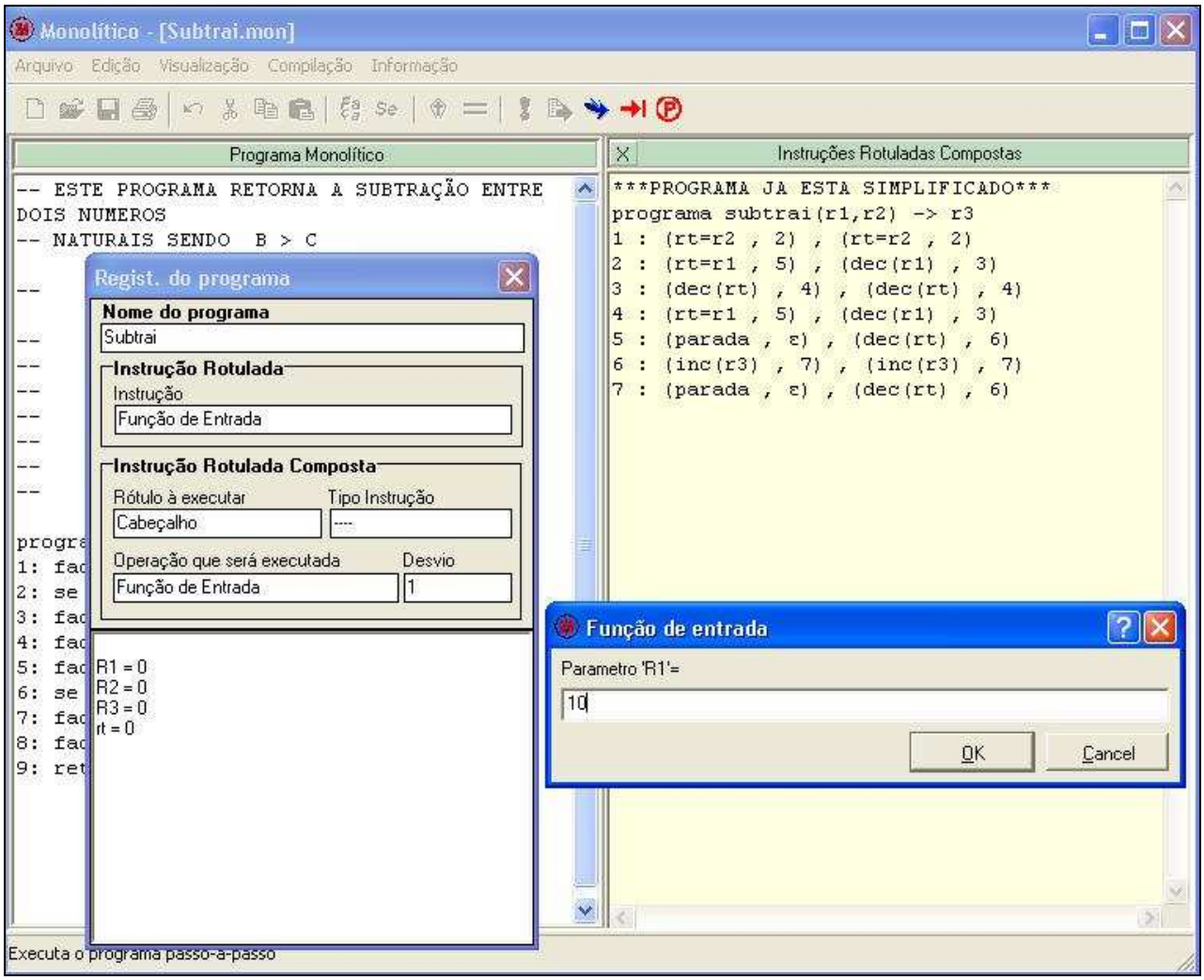


Figura 28 - Programa sendo executado passo-a-passo

A execução de um programa pode ser interrompida a qualquer momento, através do sub-menu “Parar execução” do menu principal “Compilação”.

A Figura 29 apresenta um programa que utiliza uma macro. Esta macro também é mostrada na interface do ambiente nas formas de instruções rotuladas e instruções rotuladas compostas. Se o usuário não desejar visualizar as macros, pode-se desabilitar a sua visualização através do menu principal “Visualização”.

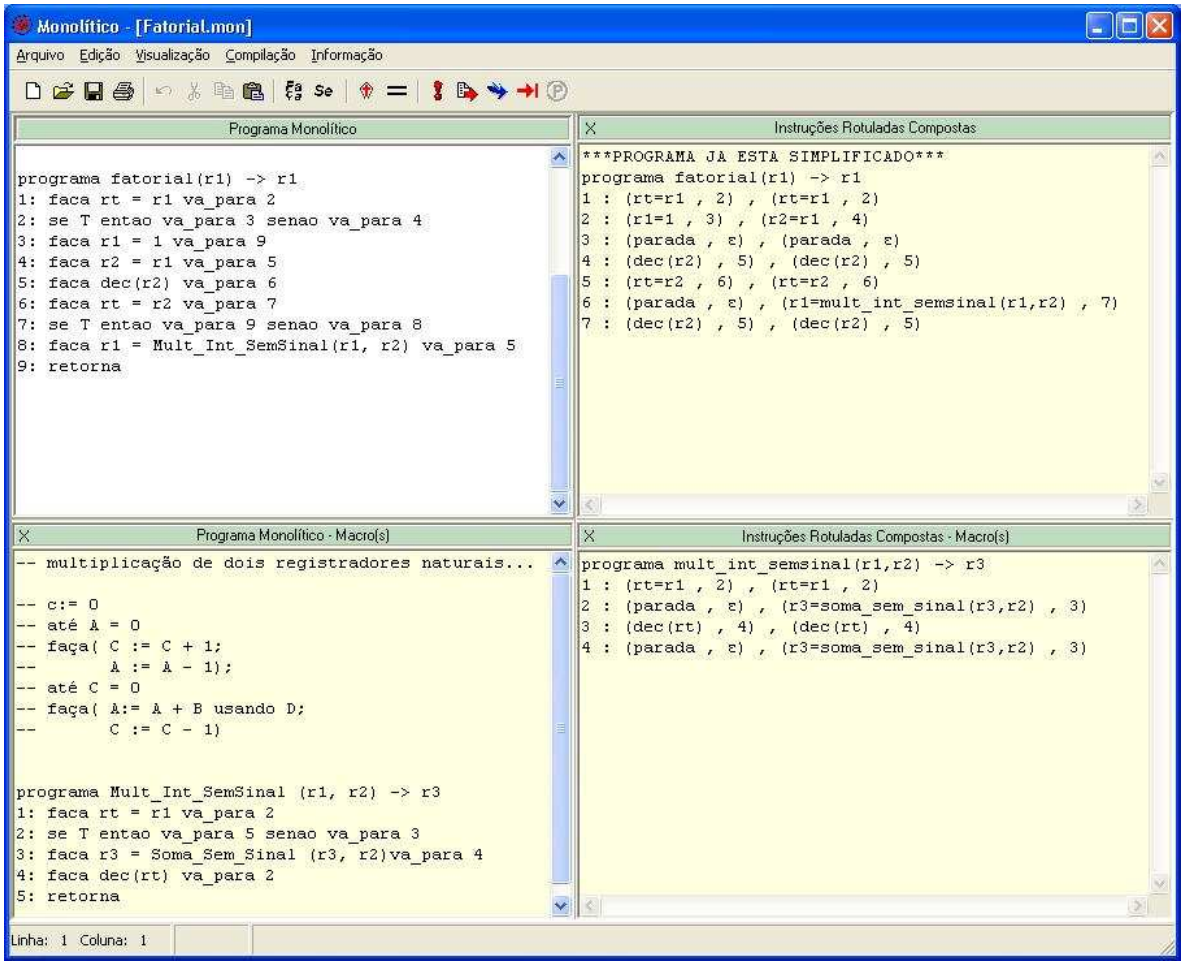


Figura 29 - Programa utilizando uma macro

A Figura 30 apresenta a verificação da existência de instruções mortas no programa, onde os rótulos mortos encontrados são mostrados através de uma mensagem.

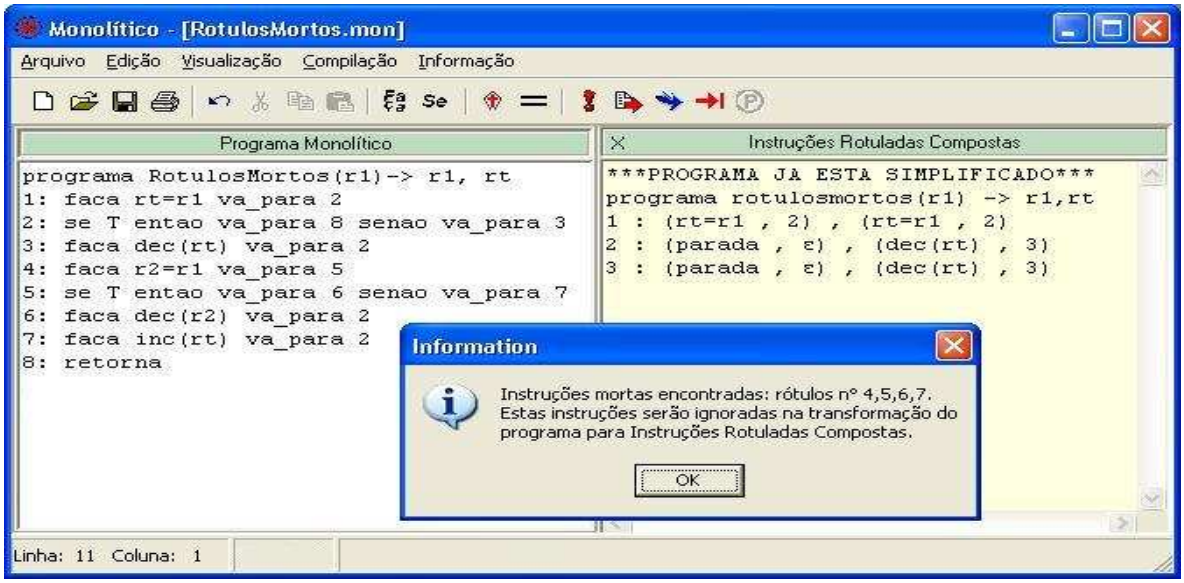


Figura 30 - Verificação da existência de instruções mortas no programa

A Figura 31 apresenta a verificação de equivalência entre dois programas monolíticos, com uma mensagem, informando que os programas não são equivalentes, onde esta mensagem informa quais são os rótulos que possuem instruções que não são equivalentes.

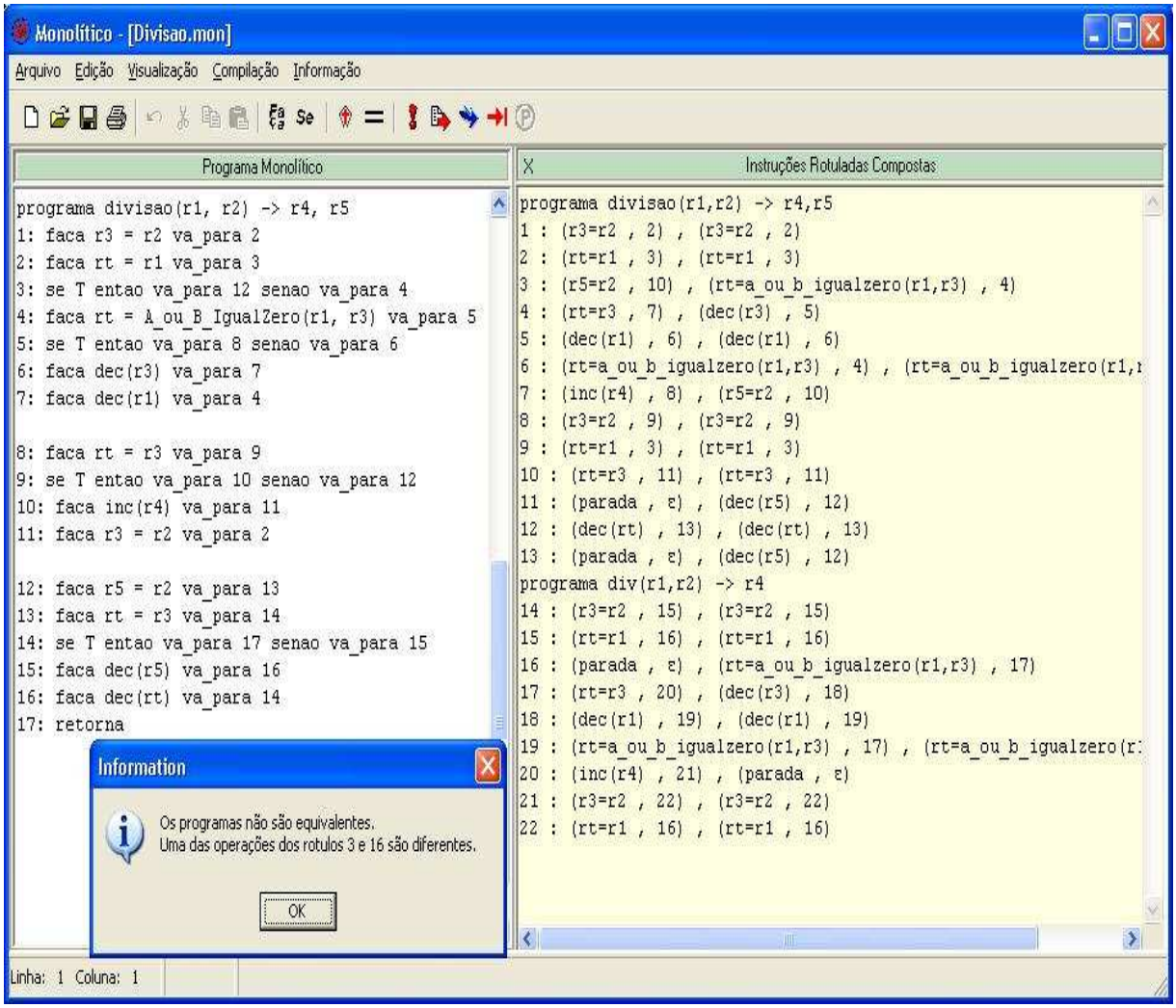


Figura 31 - Verificação de equivalência entre dois programas monolíticos

A Figura 32 apresenta um programa monolítico que foi transformado e simplificado. Durante a transformação o ambiente verifica automaticamente se o programa na forma de instruções rotuladas compostas pode ser simplificado. Se o programa foi simplificado, o ambiente mostra o programa antes e depois da simplificação. Antes da execução de qualquer programa, o ambiente verifica se o mesmo pode ser simplificado, caso seja possível, primeiramente a simplificação será feita, e somente depois o programa será executado.

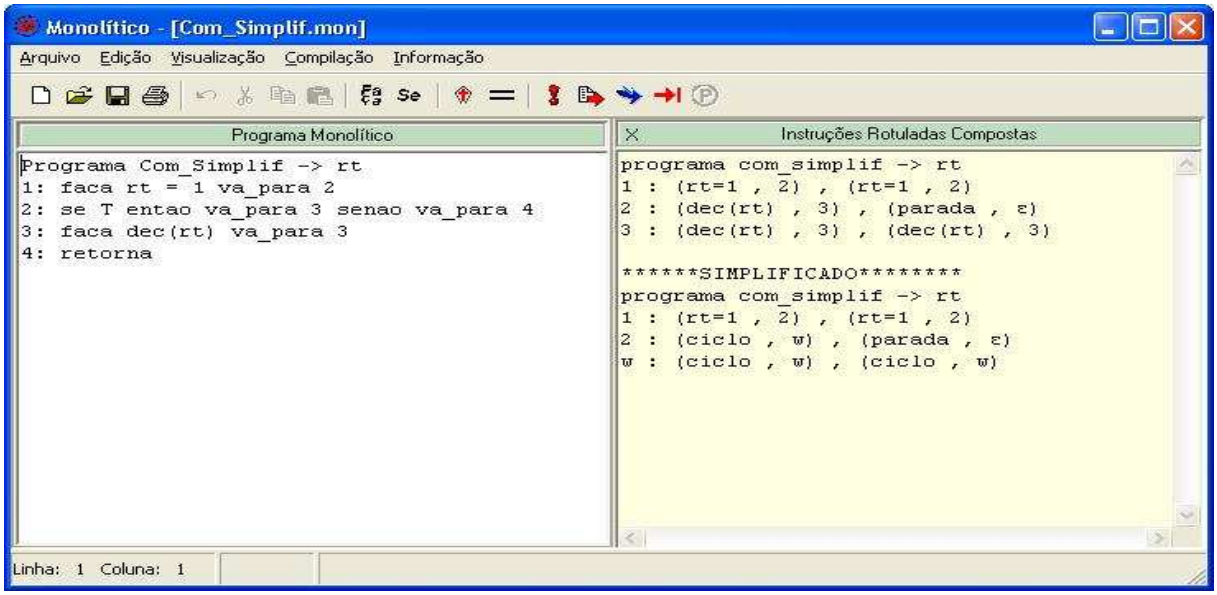


Figura 32 - Programa na forma de instruções rotuladas compostas simplificado

3.6 RESULTADOS E DISCUSSÃO

Os resultados obtidos após o término do ambiente foram satisfatórios. Alguns módulos que não estavam como requisito para o ambiente foram implementados. Entre eles estão as chamadas de macros no programa, a execução passo-a-passo do programa na forma de instruções rotuladas compostas e a possibilidade de visualização durante a execução de um programa, dos valores dos registradores utilizados pelo mesmo.

O ambiente possui algumas limitações. Entre elas está a de não poder ser feito o endereçamento indireto, utilizando registradores. Outra limitação é que para a verificação da equivalência entre dois programas, os programas devem utilizar os mesmos registradores.

A maior limitação do ambiente está na identificação de ciclos infinitos em tempo de execução. O ambiente consegue detectar os ciclos infinitos na estrutura estática do programa porém, o usuário pode definir uma instrução que, quando executada, entre em um ciclo infinito. Quando isto ocorrer, o ambiente irá executar infinitamente o programa. Para amenizar este problema, foram implementados dois módulos: um para executar o programa “passo-a-passo” e outro para interromper a execução. No módulo de execução passo-a-passo, o usuário poderá visualizar o valor dos registradores do programa, assim como a instrução que será executada a cada passo. No módulo para interromper a execução, o usuário poderá interromper a qualquer momento a execução normal ou passo-a-passo de um programa.

4 CONCLUSÕES

O objetivo principal deste trabalho, construir um ambiente para auxiliar o desenvolvimento de programas monolíticos foi alcançado. A linguagem foi especificada utilizando a notação BNF.

A utilização da ferramenta GALS para a construção dos analisadores léxico e sintático foi importante para o desenvolvimento do trabalho, pois ela gerou todas as classes para os analisadores léxico e sintático, acelerando e facilitando o desenvolvimento do ambiente. A ferramenta Rational Rose utilizada para especificação do ambiente e o ambiente de programação Delphi 7.0, utilizado para implementação do ambiente também auxiliaram consideravelmente o desenvolvimento do trabalho.

Para a implementação do ambiente foi necessário um estudo detalhado sobre programas monolíticos e suas propriedades, assim como sobre a máquina NORMA, pois a linguagem monolítica criada para o ambiente utiliza a mesma estrutura de dados (somente números naturais).

A principal contribuição deste trabalho é a aplicação do algoritmo proposto em Silva (2004), que possibilita a transformação de programas monolíticos na forma de instruções rotuladas para instruções rotuladas compostas. A grande vantagem deste novo algoritmo é a possibilidade de se transformar diretamente um programa monolítico na forma de instruções rotuladas para instruções rotuladas compostas, sem a necessidade de se converter um programa monolítico na forma de instruções rotuladas em um fluxograma, para somente depois convertê-lo em instruções rotuladas compostas.

A possibilidade de se interpretar os programas criados na linguagem desenvolvida é de grande valia para este trabalho, pois possibilita ao usuário verificar a validade dos programas e também comprova a validade da aplicação do algoritmo proposto em Silva (2004). Alguns exercícios propostos em Diverio e Menezes (2003) foram resolvidos no ambiente e são apresentados no Apêndice B.

Este trabalho poderá ser de grande valia na disciplina de Teoria da Computação, pois permitirá ao professor apresentar a funcionalidade dos programas monolíticos e também comprovar a aplicação do novo algoritmo proposto em Silva (2004).

4.1 EXTENSÕES

Como possíveis extensões para o trabalho, destacam-se:

a) implementação do endereçamento indireto para programas monolíticos;
b) implementação de um módulo para conversão de programas iterativos para programas monolíticos na forma de instruções rotuladas.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores:** princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: Guanabara Koogan, 1995.

BIRD, Richard. **Programs and machines**: an introduction to the theory of computation. London: J. Wiley, 1976.

DIVERIO, Tiarajú A.; MENEZES, Paulo F. B. **Teoria da computação**: máquinas universais e computabilidade. 2. ed. Porto Alegre: Sagra Luzzatto, 2000.

FERNANDES, Cláudia Santos et al. **Programas recursivos e conversão de programas monolíticos.** Rolândia, [2004?]. Disponível em: <www2.unoeste.br/~chico/artigo_programas_recursivos.pdf>. Acesso em: 25 out. 2004.

GALS: gerador de analisadores léxicos e sintáticos. [Florianópolis], [2003?]. Disponível em: <http://gals.sourceforge.net>. Acesso em: 11 nov. 2004.

GESSER, Carlos Eduardo. **GALS**: gerador de analisadores léxicos e sintáticos. 2003. 150 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

NASSI, I.; SCHNEIDERMAN, B. Flowchart techniques for structured programming. **SIGPLAN Notices**, [s.l.], v. 8, n. 8, p. 12-26, Aug. 1973.

SILVA, José Roque Voltolini da. **Proposta de um novo algoritmo para transformação de um programa monolítico em um programa com instruções rotuladas compostas**. Artigo não publicado. Blumenau, 2004.

APÊNDICE A – Ações semânticas da linguagem criada

Neste apêndice é apresentado o código fonte da classe TSemantico, a qual é responsável pela análise semântica dos programas monolíticos na forma de instruções rotuladas.

```
unit USemantico;

interface

uses UToken, Classes, SysUtils, uRegistrador, uProgSaida, uMonolitico,
UListaRegistradores,  uExecInstrucao, uMacrosPrograma ;

type
  TTipoOperacao = (opParametroProg, opRetorno, opAtrRegNum, opAtrRegReg,
opChamadaMacro);
  TSemantico = class
  private
    fTipoOPAtual : TTipoOperacao;
    fLabel : LongWord;
    fRotuloAtual : Longword;
    fInstrucao :Pointer;
    fExecInst : TInstrucao;
    fRegistradores : TListaRegistradores;
    fRegist_Retorno : TListaRegistradores;
    fRegistMacro : TListaRegistradores;
    fRegistOperacao : TRegistrador;
    fProgramaMonolitico :TProgramaMonolitico;
    fSemanticoMacro : TSemantico;
    fMacros_RotComp : TStrings;
    fMacros_Monolit: TStrings;
    fListaMacros: TListaMacrosProg;

    procedure setMacrosMonolitico(const Value: TStrings);
    procedure setMacrosRotComp(const Value: TStrings);
    function getMacrosMonolitico: TStrings;
    function getMacrosRotComp: TStrings;

    procedure VerificaMacro;
    procedure TransfMacroEmRotuladas(pNmMacro, pProgMon:String);
    procedure SetChamadaMacro;
    function getRegistrador(pRegistrador:String):TRegistrador;

    procedure act01_CabecalhoProg(pToken:TToken);
    procedure act02_setTipoOperacao;
    procedure act03_IncluiInstCabecalho;
    procedure act04_SetNovosRegistradores(pToken:TToken);
    procedure act05_VerifRotuloExiste(ptoken: TToken);
    procedure act06_SetTipoRotulo(pToken : TToken);
    procedure act07_SetOpIncDec(pToken : TToken);
    procedure act08_FinOpIncDec(pToken: TToken);
    procedure act09_setRegistAtrib(pToken:TToken);
    procedure act10_setRegistAtribMacro(pToken:TToken);
    procedure act11_setTipoOperacao;
    procedure act12_setTipoOperacao;
    procedure act13_FinOpAtrib(pToken : TToken);
    procedure act14_CriaChamadaMacro(pToken : TToken);
```

```
    procedure act15_VerificaParamRet_Macro(pToken : TToken);
    procedure act16_IncluiInstOperacao(pToken : TToken);
    procedure act17_setDesvioTesteVerd(pToken :TToken);
    procedure act18_setDesvTesteVerd_IncluiInstTeste(pToken : TToken);
  public
    constructor Create;

    destructor Destroy; override;
    property MacrosMonolitico: TStrings read getMacrosMonolitico
            write setMacrosMonolitico;
    property MacrosRotComp : TStrings read getMacrosRotComp
            write setMacrosRotComp;
    property ProgMon : TProgramaMonolitico
            read fProgramaMonolitico write fProgramaMonolitico;
    procedure ExecuteAction(action : integer; const token : TToken);
  end;

implementation

uses uCabecalhoPrograma, ULexico, USintatico, USemanticError, uPrincipal,
uMensagens ;

constructor TSemantico.create;
begin
  fRegistradores := TListaRegistradores.create;
  fRegist_Retorno := TListaRegistradores.create;
  fProgramaMonolitico := TProgramaMonolitico.Create;
  fSemanticoMacro := NIL;
  fRegistMacro := Nil;
  fListaMacros := Nil;
  fMacros_RotComp := Nil;
  fMacros_Monolit := Nil;
  fLabel := 2;
end;

destructor TSemantico.destroy;
begin
  (* LIBERA MEMORIA ALOCADA*)
  fRegistradores.Free;
  fRegist_Retorno.Free;
  fProgramaMonolitico.Free;
  if fSemanticomacro <> NIl then
    fSemanticomacro.Destroy;
  if fListaMacros <> Nil then
    fListaMacros.Destroy;
  inherited;
end;

procedure TSemantico.setMacrosMonolitico(const Value: TStrings);
begin
  fMacros_Monolit := Value;
end;

procedure TSemantico.setMacrosRotComp(const Value: TStrings);
begin
  fMacros_RotComp:= Value;
end;
function TSemantico.getMacrosMonolitico: TStrings;
begin
  Result := fMacros_Monolit;
```

```
end;

function TSemantico.getMacrosRotComp: TStrings;
begin
  Result := fMacros_RotComp;
end;

procedure TSemantico.executeAction(action : integer; const token : TToken);
begin
  case action of
    1: act01_CabecalhoProg(Token);
    2: act02_setTipoOperacao;
    3: act03_IncluiInstCabecalho;
    4: act04_SetNovosRegistradores(Token);
    5: act05_VerifRotuloExiste(Token);
    6: act06_setTipoRotulo(Token);
    7: act07_SetOpIncDec(Token);
    8: act08_FinOpIncDec(Token);
    9: act09_setRegistAtrib(Token);
    10: act10_setRegistAtribMacro(Token);
    11: act11_setTipoOperacao;
    12: act12_setTipoOperacao;
    13: act13_FinOpAtrib(Token);
    14: act14_CriaChamadaMacro(Token);
    15: act15_VerificaParamRet_Macro(Token);
    16: act16_IncluiInstOperacao(Token);
    17: act17_setDesvioTesteVerd(Token);
    18: act18_setDesvTesteVerd_IncluiInstTeste(Token);
  end;
end;


procedure TSemantico.VerificaMacro;
var
  vNmMacro,
  vArquivo:String;
  vProgMon : TStringList;
  vNmProg:String;
  vMacro : TProgramaRotComp;
begin
  vNmProg := TCabecalhoProg(TProgramaMonolitico(
            fProgramaMonolitico).ProgMon[0]).NmPrograma;
  vNmMacro := TInst_Atrib_Macro(fExecInst).NmMacro;
  if LowerCase(vNmMacro) = LowerCase(vNmProg) then
    Raise ESemanticError.create(Format(
            sRecursivo, [vNmMacro]), TInstMon_Operacao(
            fInstrucao).Posicao);
  vMacro := fListaMacros.MacroEstaLista(vNmMacro);
  if vMacro <> Nil then
    begin
      TInst_Atrib_Macro(fExecInst).Macro := vMacro;
      exit;
    end;
  vArquivo := frmMonolitico.OpenDialog.InitialDir + '\Lib\' +
            vNmMacro + '.mon';
  try
    vProgMon := TStringList.Create;
    vProgMon.LoadFromFile(vArquivo);
  except
    Raise ESemanticError.create(Format(sMacroNaoEncontrada,
```

```
            [vNmMacro]),TInstMon_Operacao(fInstrucao).Posicao)
  end;
  TransfMacroEmRotuladas(vNmMacro, vProgMon.Text);
  vProgMon.Free;
end;


procedure TSemantico.TransfMacroEmRotuladas(pNmMacro, pProgMon:String);
var
  vLexico : TLexico;
  vSintatico : TSintatico;
begin
  vLexico := TLexico.create(pProgMon);
  vSintatico := TSintatico.create;
  fSemanticoMacro := TSemantico.Create;
  try
    vSintatico.parse(vLexico, fSemanticoMacro);
    TInst_Atrib_Macro(fExecInst).Macro :=
      fSemanticoMacro.ProgMon.TransfMon_RotComp.SimpRotComp;
    if fMacros_RotComp = Nil then
      begin
        fMacros_Monolit := TStringList.Create;
        fMacros_RotComp := TStringList.Create;
      end;
    fMacros_Monolit.Add(Trim(pProgMon));
    fMacros_RotComp.Add(TInst_Atrib_Macro(
            fExecInst).Macro.MostraRotuladasComp);
    fListaMacros.IncluiMacro(
            pNmMacro, TInst_Atrib_Macro(fExecInst).Macro);
  except
    Raise ESemanticError.Create(format(sErroTransfMacro,
            [TInst_Atrib_Macro(fExecInst).NmMacro]),
            TInstMon_Operacao(fInstrucao).Posicao);
  end;
  vLexico.Destroy;
  vSintatico.Destroy;
end;

procedure TSemantico.SetChamadaMacro;
begin
  if fListaMacros = Nil then
    fListaMacros := TListaMacrosProg.Create;
  fTipoOPAtual := opChamadaMacro;
  fRegistMacro := TListaRegistradores.create;
  fRegistMacro.IncluiRegistrador(fRegistOperacao);
end;

function TSemantico.getRegistrador(pRegistrador:String):TRegistrador;
begin
  Result := fRegistradores.RegExiste(pRegistrador);
  if   Result = NIL then
    begin
      Result := fRegistradores.CriaNovoReg(pRegistrador, 0);
      fRegistradores.IncluiRegistrador(Result);
    end;
end;

procedure TSemantico.act01_CabecalhoProg(pToken: TToken);
begin
  fTipoOPAtual := opParametroProg;
```

```
  fInstrucao := TCabecalhoProg.Create;
  TCabecalhoProg(fInstrucao).NmPrograma := ptoken.getLexeme;
  TCabecalhoProg(fInstrucao).Registradores := fRegistradores;
end;



procedure TSemantico.act02_setTipoOperacao;
begin
  fTipoOPAtual := opRetorno;
end;

procedure TSemantico.act03_IncluiInstCabecalho;
begin
  fProgramaMonolitico.IncluiInstrucao(fInstrucao);
end;

procedure TSemantico.act04_SetNovosRegistradores(pToken: TToken);
var
  vNmReg : String;
  vReg : TRegistrador;
begin
  vNmReg := pToken.getLexeme;
  vReg:= getRegistrador(pToken.getLexeme);
  if fTipoOPAtual = opParametroProg then
    begin
      if TCabecalhoProg(fInstrucao).Parametros = Nil then
         TCabecalhoProg(fInstrucao).Parametros :=
                  TListaRegistradores.Create;
       TCabecalhoProg(fInstrucao).Parametros.IncluiRegistrador(vReg);
    end
  else if fTipoOPAtual =  opRetorno then
    TCabecalhoProg(fInstrucao).Retorno.IncluiRegistrador(vReg)
  else if fTipoOPAtual = opChamadaMacro then
    begin
      if TInst_Atrib_Macro(fExecInst).Paramentos = Nil then
        begin
          TInst_Atrib_Macro(fExecInst).Paramentos :=
                  TListaRegistradores.Create;
          TInstMon_Operacao(fInstrucao).Operacao:=
                  TInstMon_Operacao(fInstrucao).Operacao + '(' ;
        end;
      TInst_Atrib_Macro(fExecInst).Paramentos.IncluiRegistrador(vReg);
      if TInst_Atrib_Macro(fExecInst).Paramentos.ListaReg.Count <= 1 then
        TInstMon_Operacao(fInstrucao).Operacao :=
                  TInstMon_Operacao(fInstrucao).Operacao + vNmReg
      else
        TInstMon_Operacao(fInstrucao).Operacao :=
                  TInstMon_Operacao(fInstrucao).Operacao + ',' + vNmReg;
    end;
end;

procedure TSemantico.act05_VerifRotuloExiste(pToken: TToken);
var
  vRotulo : Integer;
begin
  vRotulo := StrToInt(pToken.getLexeme);
  if fProgramaMonolitico.getEndInstrucao(vRotulo) = NIL then
    fRotuloAtual := vRotulo
  else
```

```
    raise ESemanticError.create(Format(sRotuloRepetido ,[vRotulo]),
                pToken.getPosition);
end;



procedure TSemantico.act06_SetTipoRotulo(pToken: TToken);
begin
  if UpperCase(pToken.getLexeme) = 'SE' then
    begin
      if fRegistradores.RegExiste('rt') = NIL then
        raise ESemanticError.create(sRegTesteInexistente,
                ptoken.getPosition);
      fInstrucao := TInstMon_Teste.Create
    end
  else if UpperCase(pToken.getLexeme) = 'FACA' then
    begin
      fInstrucao := TInstMon_Operacao.Create;
      TInstMon_Operacao(fInstrucao).Label_ := fLabel;
      TInstMon_Operacao(fInstrucao).Transf := False;
      Inc(fLabel);
    end
  else
    begin
      fInstrucao := TInsMon_Retorno.Create;
      fProgramaMonolitico.IncluiInstrucao(fInstrucao);
    end;
  TInstMonolitica(fInstrucao).Rotulo := fRotuloAtual;
  TInstMonolitica(fInstrucao).Posicao := pToken.getPosition;
end;

procedure TSemantico.act07_SetOpIncDec(pToken: TToken);
var
  vReg : String;
begin
  vReg := LowerCase(pToken.getLexeme) ;
  if  (vReg = 'inc') then
    fExecInst := TInst_Incrementa.Create
  else if (vReg = 'dec') then
    fExecInst := TInst_Decrementa.Create;
  TInstMon_Operacao(fInstrucao).Operacao:= vReg + '(';
end;

procedure TSemantico.act08_FinOpIncDec(pToken: TToken);
begin
  TInstrucao(fExecInst).Registrador := getRegistrador(pToken.getLexeme);
  TInstMon_Operacao(fInstrucao).Operacao :=
    TInstMon_Operacao(fInstrucao).Operacao +
    TInstrucao(fExecInst).Registrador.NmReg + ')';
end;

procedure TSemantico.act09_setRegistAtrib(pToken: TToken);
begin
  fRegistOperacao := getRegistrador(pToken.getLexeme);
  TInstMon_Operacao(fInstrucao).Operacao := fRegistOperacao.NmReg ;
end;

procedure TSemantico.act10_setRegistAtribMacro(pToken: TToken);
var
```

```
  vReg: TRegistrador;
begin
  if fRegistMacro = Nil then
    SetChamadaMacro;
  vReg:= getRegistrador(pToken.getLexeme);
  fRegistMacro.IncluiRegistrador(vReg);
  TInstMon_Operacao(fInstrucao).Operacao :=
    TInstMon_Operacao(fInstrucao).Operacao + ',' + vReg.NmReg;
end;

procedure TSemantico.act11_setTipoOperacao;
begin
  fTipoOPAtual := opAtrRegReg;
end;

procedure TSemantico.act12_setTipoOperacao;
begin
  fTipoOPAtual := opAtrRegNum;
end;

procedure TSemantico.act13_FinOpAtrib(pToken: TToken);
var
  vReg:TRegistrador;
begin
  if fTipoOPAtual = opChamadaMacro then
    raise ESemanticError.create(format(sErroAtrib,[pToken.getLexeme]),
pToken.getPosition)
  else
    begin
      if fTipoOPAtual = opAtrRegReg then
        begin
          vReg := getRegistrador(pToken.getLexeme);
          fExecInst := TInst_Atrib_RegReg.Create;
          TInst_Atrib_RegReg(fExecInst).Registrador2 := vReg;
        end
      else
        begin
          fExecInst := TInst_Atrib_RegNum.Create;
          TInst_Atrib_RegNum(fExecInst).Valor :=
                  StrToInt(pToken.getLexeme);
        end;
      TInstrucao(fExecInst).Registrador := fRegistOperacao;
      TInstMon_Operacao(fInstrucao).Operacao :=
         TInstMon_Operacao(fInstrucao).Operacao + '=' + pToken.getLexeme;
    end;
end;

procedure TSemantico.act14_CriaChamadaMacro(ptoken:TToken);
begin
  if fRegistMacro = Nil then
    SetChamadaMacro;
  TInstMon_Operacao(fInstrucao).Operacao :=
      TInstMon_Operacao(fInstrucao).Operacao + '=' + pToken.getLexeme;
  fExecInst := TInst_Atrib_Macro.Create;
  TInst_Atrib_Macro(fExecInst).ListaReg:= fRegistMacro;
  TInst_Atrib_Macro(fExecInst).NmMacro := ptoken.getLexeme;
  TInst_Atrib_Macro(fExecInst).Paramentos := Nil;
end;

procedure TSemantico.act15_VerificaParamRet_Macro(pToken:TToken);
```

```
var
  vAux_A,
  vAux_B  : TListaRegistradores;
begin
  VerificaMacro;


  vAux_A :=
            TCabecalhoProg(TInst_Atrib_Macro(
            fExecInst).Macro.ProgRotComp[0]).Retorno;
  vAux_B := TInst_Atrib_Macro(fExecInst).ListaReg;

  if vAux_A.ListaReg.Count <> vAux_B.ListaReg.Count then
    raise ESemanticError.create(sErroAtribMacro, ptoken.getPosition);

  vAux_A := TInst_Atrib_Macro(fExecInst).Paramentos;
  vAux_B := TCabecalhoProg(TInst_Atrib_Macro(
      fExecInst).Macro.ProgRotComp[0]).Parametros;

  if ((vAux_A <> Nil) and (vAux_B <> Nil)) then
    begin
     if (vAux_A.ListaReg.Count <> vAux_B.ListaReg.Count) then
       raise ESemanticError.create(sErroPassagemParametros,
                  ptoken.getPosition);
    end
  else if (vAux_A <> vAux_B) then
    raise ESemanticError.create(
            sErroPassagemParametros, ptoken.getPosition);

  fRegistMacro:= Nil;
  TInstMon_Operacao(fInstrucao).Operacao:=
            TInstMon_Operacao(fInstrucao).Operacao + ')';
end;

procedure TSemantico.act16_IncluiInstOperacao(pToken: TToken);
begin
  TInstMon_Operacao(fInstrucao).Instrucao := fExecInst;
  TInstMon_Operacao(fInstrucao).Desvio := StrToInt(pToken.getLexeme);
  fProgramaMonolitico.IncluiInstrucao(fInstrucao);
end;

procedure TSemantico.act17_setDesvioTesteVerd(pToken: TToken);
begin
  TInstMon_Teste(fInstrucao).Desvio_Verd := StrToInt(ptoken.getLexeme);
end;

procedure TSemantico.act18_setDesvTesteVerd_IncluiInstTeste(
  pToken: TToken);
begin
  TInstMon_Teste(fInstrucao).Desvio_Falso := StrToInt(pToken.getLexeme);
  fProgramaMonolitico.IncluiInstrucao(fInstrucao);
end;

end.
```

APÊNDICE B – Programas exemplos resolvidos

Neste apêndice são apresentados alguns programas monolíticos na forma de instruções rotuladas, que foram testados através do ambiente.

```
-- este programa testa se um dos dois registradores
-- passados como parametro é zero...

-- ele poderá ser utilizado em uma construção do tipo:
--   até (A = 0 ou B = 0
--         faça (...)

-- o retorno do programa será:
--    0 - caso um dos dois registradores posssui o valor zero
--    1 - caso contrário

programa A_ou_B_IgualZero (rt, r2) -> rt
1: se T entao va_para 5 senao va_para 2
2: faca rt = r2 va_para 3
3: se T entao va_para 5 senao va_para 4
4: faca rt = 1 va_para 5
5: retorna
```

Quadro 55 - Programa que testa se (A = 0 ou B = 0)

```
--    Este programa retorna a parte inteira da divisão entre
--    dois números naturais...

--    A := B div C
--    OBS: Ao final da execução, o quociente estará armazenado em A

--    A := 0;
--    D := C usando E;
--    até B = 0
--    faça ((até B = 0 ou D = 0 )
--          faça ( B := B - 1;
--                 D := D - 1 ) )
--          se D = 0
--          então  ( A := A + 1;
--                   D := C usando E ) )

--    Se r2 = 0 a divisão é impossível (PROG. FICA EM LOOP)

programa Div (r1, r2) -> r4
1: faca r3 = r2 va_para 2
2: faca rt = r1 va_para 3
3: se T entao va_para 12 senao va_para 4
4: faca rt = A_ou_B_IgualZero(r1, r3) va_para 5
5: se T entao va_para 8 senao va_para 6
6: faca dec(r3) va_para 7
7: faca dec(r1) va_para 4
8: faca rt = r3 va_para 9
9: se T entao va_para 10 senao va_para 12
10: faca inc(r4) va_para 11
11: faca r3 = r2 va_para 2
12: retorna
```

Quadro 56 - Programa que retorna a parte inteira da divisão entre dois números naturais

```
--    Este programa retorna o resultado da divisão entre dois
--    números inteiros...
--    A := B / C
--    OBS: Ao final da execução, o quociente estará
--    armazenado em A e o resto em F

--    A := 0;
--    D := C usando E;
--    até B = 0
--    faça ((até B = 0 ou D = 0 )
--          faça ( B := B - 1;
--                 D := D - 1 ) )
--          se D = 0
--          então  ( A := A + 1;
--                   D := C usando E ) )
--    F : = C usando E;
--    até D = 0
--    faça ( F := F - 1;
--           D := D - 1 ) )

--    r4 = parte inteira
--    r5 = resto da divisão
--    se r2 = 0 a divisão é impossível (PROG. FICA EM LOOP)

programa divisao(r1, r2) -> r4, r5
1: faca r3 = r2 va_para 2
2: faca rt = r1 va_para 3
3: se T entao va_para 12 senao va_para 4
4: faca rt = A_ou_B_IgualZero(r1, r3) va_para 5
5: se T entao va_para 8 senao va_para 6
6: faca dec(r3) va_para 7
7: faca dec(r1) va_para 4

8: faca rt = r3 va_para 9
9: se T entao va_para 10 senao va_para 12
10: faca inc(r4) va_para 11
11: faca r3 = r2 va_para 2

12: faca r5 = r2 va_para 13
13: faca rt = r3 va_para 14
14: se T entao va_para 17 senao va_para 15
15: faca dec(r5) va_para 16
16: faca dec(rt) va_para 14
17: retorna
```

Quadro 57 - Programa que retorna a divisão entre dois números naturais

```
-- este programa retorna em r1 a soma de
-- dois registradores naturais

programa Soma_Sem_Sinal(r1, r2) -> r1
1: faca rt = r2 va_para 2
2: se T entao va_para 5 senao va_para 3
3: faca inc(r1) va_para 4
4: faca dec(rt) va_para 2
5: retorna
```

Quadro 58 - Programa que retorna a soma de dois números naturais

```
-- multiplicação de dois registradores naturais...

-- c:= 0
-- até A = 0
-- faça( C := C + 1;
--       A := A - 1);
-- até C = 0
-- faça( A:= A + B usando D;
--       C := C - 1)


programa Mult_Int_SemSinal (r1, r2) -> r3
1: faca rt = r1 va_para 2
2: se T entao va_para 5 senao va_para 3
3: faca r3 = Soma_Sem_Sinal (r3, r2)va_para 4
4: faca dec(rt) va_para 2
5: retorna
```

Quadro 59 - Programa que retorna a multiplicação entre dois registradores naturais

```
-- este programa retorna o fatorial de um número
-- natural.  (n!)

-- A := A!

-- se A = 0
-- entao A := 1
-- senao(B := A
--       B := B - 1
--       ate B := 0
--       faca(A := A * B
--            B := B - 1))

programa fatorial(r1) -> r1
1: faca rt = r1 va_para 2
2: se T entao va_para 3 senao va_para 4
3: faca r1 = 1 va_para 9
4: faca r2 = r1 va_para 5
5: faca dec(r2) va_para 6
6: faca rt = r2 va_para 7
7: se T entao va_para 9 senao va_para 8
8: faca r1 = Mult_Int_SemSinal(r1, r2) va_para 5
9: retorna
```

Quadro 60 - Programa que retorna o fatorial de um número natural

```
-- ESTE PROGRAMA RETORNA A SUBTRAÇÃO ENTRE DOIS NUMEROS
-- NATURAIS, SENDO  B > C

--    A := B - C

--    A := 0;
--    até  C = 0
--    faça  ( B := B - 1;
--            C := C - 1 );
--    até B = 0
--    faça  ( B := B - 1;
--            A := A + 1 );

programa Subtrai (R1, R2) -> R3
1: faca rt = r2 va_para 2
2: se T entao va_para 5 senao va_para 3
3: faca dec(r1) va_para 4
4: faca dec(rt) va_para 2
5: faca rt = r1 va_para 6
6: se T entao va_para 9 senao va_para 7
7: faca dec(rt) va_para 8
8: faca inc(r3) va_para 6
9: retorna
```

Quadro 61 - Programa que retorna o resultado da subtração de dois números naturais

```
-- este programa retorna 0 se A < B e 1 caso
-- contrario (utilizando numeros naturais)

-- A < B

-- ate (A = 0 ou B = 0)
-- faça ( A := A - 1;
--        B := B - 1)
-- se B = 0
-- então falso
-- senão verdadeiro

programa A_Menor_B (r1, r2) -> rt
1: faca rt = A_ou_B_IgualZero(r1, r2) va_para 2
2: se T entao va_para 5 senao va_para 3
3: faca dec(r1) va_para 4
4: faca dec(r2) va_para 1
5: faca rt = r2 va_para 6
6: se T entao va_para 7 senao va_para 8
7: faca rt = 1 va_para 9
8: faca rt = 0 va_para 9
9: retorna
```

Quadro 62 - Programa que verifica se A < B (utilizando números naturais)

```
-- este programa retorna 0 se A <= B e 1 caso
-- contrario (utilizando numeros naturais)

-- A < B

-- ate (A = 0 ou B = 0)
-- faça ( A := A - 1;
--        B := B - 1)
-- se B = 0
-- então verdadeiro
-- senão falso

programa A_MenorIgual_B (r1, r2) -> rt
1: faca rt = A_ou_B_IgualZero(r1, r2) va_para 2
2: se T entao va_para 5 senao va_para 3
3: faca dec(r1) va_para 4
4: faca dec(r2) va_para 1
5: faca rt = r1 va_para 6
6: se T entao va_para 7 senao va_para 8
7: faca rt = 0 va_para 9
8: faca rt = 1 va_para 9
9: retorna
```

Quadro 63 - Programa que verifica se A <= B (utilizando números naturais)

```
-- Este programa retorna a soma de dois números inteiros...

-- A := B + C
-- A1, B1, C1 = SINAL
-- A2, B2, C2 = MAGNITUDE

-- se B1 = C1
-- então (A1 := B1 usando D;
--        A2 := B2 + C2 usando D)
-- senão (A2 := B2 usando D;
--        até A2 = 0 ou C2 = 0
--        faça (A2 := A2 - 1;
--              C2 := C2 - 1)
--        se C2 = 0
--        então se A2 = 0
--              então A1 = 0
--              senão A1 := B1 usando D
--        senão (até C2 =
--               faça ( A2 := A2 + 1;
--                      C2 := C2 - 1)
--               A1 := C1 usando D)

--    r1 = sinal do número r2 (0 = positivo, 1 = negativo)
--    r2 = número
--    r3 = sinal do número r4 (0 = positivo, 1 = negativo)
--    r4 = número
programa SomaInteiros_A_B (r1, r2, r3, r4) -> r5, r6
1: faca rt = Comp_Dois_Num_Iguais(r1, r3) va_para 2
2: se T entao va_para 3 senao va_para 6
3: faca r5 = r1 va_para 4
4: faca r6 = Soma_Sem_Sinal(R2, R4) va_para 22

6: faca r6 = r2 va_para 7
7: faca rt = A_ou_B_IgualZero(r6, r4) va_para 8
8: se T entao va_para 11 senao va_para 9
9: faca dec(r6) va_para 10
10: faca dec(r4) va_para 7

11: faca rt = r4 va_para 12
12: se T entao va_para 13 senao va_para 17
13: faca rt = r6 va_para 14
14: se T entao va_para 15 senao va_para 16
15: faca r5 = 0 va_para 22
16: faca r5 = r1 va_para 22

17: faca rt = r4 va_para 18
18: se T entao va_para 21 senao va_para 19
19: faca inc(r6) va_para 20
20: faca dec(rt) va_para 18
21: faca r5 = r3 va_para 22
22: retorna
```

Quadro 64 - Programa que retorna a soma de dois números inteiros

```
-- este programa retorna a multiplicação entre dois
-- números inteiros...
-- A:= B * C
-- A1, B1, C1 = SINAL
-- A1, B2, C2 = MAGNITUDE (Número)

-- se B2 = 0 ou C2 = 0
-- entao (A1 = 0;
--        A2 = 0)
-- senão(se B1 = 0
--       então  se C1 = 0
--              então A1 := 0
--              senão A1 := 1
--       senão  se C1 = 0
--              então A1 := 1
--              senão A1 := 0;
--       A2 := 0;
--       até C2 = 0
--       faça (A2 := A2 + B2 usando D;
--             C2 := C2 - 1))

--    r1 = sinal do número r2 (0 = positivo, 1 = negativo)
--    r2 = número
--    r3 = sinal do número r4 (0 = positivo, 1 = negativo)
--    r4 = número
programa MultiplicaInteiros_A_B (r1, r2, r3, r4) -> r5, r6
16: faca rt = A_ou_B_IgualZero(r2, r4) va_para 17
17: se T entao va_para 18 senao va_para 1
18: faca r5 = 0 va_para 19
19: faca r6 = 0 va_para 15
1: faca rt = r1 va_para 2
2: se T entao va_para 3 senao va_para 7
3: faca rt = r3 va_para 4
4: se T entao va_para 5 senao va_para 6
5: faca r5 = 0 va_para 11
6: faca r5 = 1 va_para 11
7: faca rt = r3 va_para 8
8: se T entao va_para 9 senao va_para 10
9: faca r5 = 1 va_para 11
10: faca r5 = 0 va_para 11
11: faca rt = r4 va_para 12
12: se T entao va_para 15 senao va_para 13
13: faca r6 = Soma_Sem_Sinal(r6, r2) va_para 14
14: faca dec(rt) va_para 12
15: retorna
```

Quadro 65 - Programa que retorna a multiplicação entre dois números inteiros

```
-- Este programa retorna a subtração de dois números inteiros
-- A = B - C
-- A1, B1, C1 = SINAL
-- A2, B2, C2 = MAGNITUDE (Número)
-- se C1 = 1
-- entao(se B1 = 0
--       entao(A1 = 0; A2 = B2 + C2)
--       senao(se C2 < B2
--             então (A1 = 1; A2 = B2 - C2)
--             senão (A1 = 0; A2 = C2 - B2)))
-- senao(se B1 = 1
--       entao(A1 = 1; A2 = B2 + C2)
--       senao(se B2 < C2
--             então (A1 = 1; A2 = C2 - B2)
--             senão (A1 = 0; A2 = B2 - C2)))

--    r1, r3, r5 = sinal (0 = positivo, 1 = negativo)
--    r2, r4, r6 = número

programa SubtraiInteiros_A_B (r1, r2, r3, r4) -> r5, r6
1: faca rt = r3 va_para 2
2: se T entao va_para 13 senao va_para 3
3: faca rt = r1 va_para 4
4: se T entao va_para 5 senao va_para 7
5: faca r5 = 0 va_para 6
6: faca r6 = soma_sem_sinal(r2, r4) va_para 23
7: faca rt = A_Menor_B(r4, r2) va_para 8
8: se T entao va_para 9 senao va_para 11
9: faca r5 = 1 va para 10
10: faca r6 = subtrai(r2, r4) va_para 23
11: faca r5 = 0 va_para 12
12: faca r6 = subtrai(r4, r2) va_para 23

13: faca rt = r1 va_para 14
14: se T entao va_para 17 senao va_para 15
15: faca r5 = 1 va_para 16
16: faca r6 = soma_sem_sinal(r2,r4) va_para 23
17: faca rt = A_Menor_B(r2, r4) va_para 18
18: se T entao va_para 19 senao va_para 21
19: faca r5 = 1 va_para 20
20: faca r6 = subtrai(r4, r2) va_para 23
21: faca r5 = 0 va_para 22
22: faca r6 = subtrai(r2, r4) va_para 23
23: retorna
```

Quadro 66 - Programa que retorna a subtração entre dois números inteiros

```
-- este progrma retorna o resultado da divisão entre
-- dois números inteiros

-- A := B / C

-- A1, B1, C1 = sinal
-- A2, B2, C2 = MAGNITUDE (Número)

-- se B1 = 1
-- então( se C1 = 1
--         então A1 := 0
--         senão A1 := 1)
-- senão(se C1 = 1
--        então A1 = 1
--        senão A1 = 0)
-- A2 = B2 / C2

-- r1, r3, r5 = Sinal (0 = positivo, 1 = negativo)
-- r2, r4, r6 = Magnitude (Número)
-- r7 = resto da divisão

-- Se r4 = 0 a divisão é impossível (PROG. FICA EM LOOP)

programa DivisaoInteiros_A_B (r1, r2, r3, r4) -> r5, r6, r7
1: faca rt = r1 va_para 2
2: se T entao va_para 7 senao va_para 3
3: faca rt = r3 va_para 4
4: se T entao va_para 6 senao va_para 5
5: faca r5 = 0 va_para 11
6: faca r5 = 1 va_para 11
7: faca rt = r3 va_para 8
8: se T entao va_para 10 senao va_para 9
9: faca r5 = 1 va_para 11
10: faca r5 = 0 va_para 11
11: faca r6, r7 = divisao(r2, r4) va_para 12
12: retorna
```

Quadro 67 - Programa que retorna a divisão entre dois números inteiros

```
-- Este programa retorna a soma entre
-- dois números racionais positivos.
-- sendo que um número rácional r é representado
-- pelo par ordenado (a,b) com b > 0
-- e r = a / b

-- Número racional 1 = (r1, r2) = r1 / r2
-- Numero racional 2 = (r3, r4) = r3 / r4

-- r5, r6 = ((r1 * r4) + (r2 * r3)) , (r2 * r4)

-- O resultado da soma será r5 / r6

programa Soma_Rac_Pos_A_B(r1, r2, r3, r4) -> r5, r6
1: faca r5 = Mult_Int_SemSinal(r1, r4) va_para 2
2: faca r6 = Mult_Int_SemSinal(r2, r3) va_para 3
3: faca r5 = Soma_Sem_Sinal(r5, r6) va_para 4
4: faca r6 = Mult_Int_SemSinal(r2, r4) va_para 5
5: retorna
```

Quadro 68 - Programa que retorna a soma de dois números racionais positivos

```
-- Este programa retorna a subtração entre
-- dois números racionais positivos.
-- sendo que um número rácional r é representado
-- pelo par ordenado (a,b) com b > 0
-- e r = a / b

-- Número racional 1 = (r1, r2) = r1 / r2
-- Numero racional 2 = (r3, r4) = r3 / r4

-- r5, r6 = ((r1 * r4) - (r2 * r3)) , (r2 * r4)

-- O resultado da subtração será r5 / r6

programa Subtracao_Rac_Pos_A_B(r1, r2, r3, r4) -> r5, r6
1: faca r5 = Mult_Int_SemSinal(r1, r4) va_para 2
2: faca r6 = Mult_Int_SemSinal(r2, r3) va_para 3
3: faca r5 = Subtrai(r5, r6) va_para 4
4: faca r6 = Mult_Int_SemSinal(r2, r4) va_para 5
5: retorna
```

Quadro 69 - Programa que retorna a subtração de dois números racionais positivos

```
-- Este programa retorna a multiplicação entre
-- dois números racionais positivos.
-- sendo que um número rácional r é representado
-- pelo par ordenado (a,b) com b > 0
-- e r = a / b

-- Número racional 1 = (r1, r2) = r1 / r2
-- Numero racional 2 = (r3, r4) = r3 / r4

-- r5, r6 =  (r1 * r3), (r2 * r4)

-- O resultado da multiplicação será r5 / r6

programa Multiplicacao_Rac_Pos_A_B(r1, r2, r3, r4) -> r5, r6
1: faca r5 = Mult_Int_SemSinal(r1, r3) va_para 2
2: faca r6 = Mult_Int_SemSinal(r2, r4) va_para 3
3: retorna
```

Quadro 70 - Programa que retorna a multiplicação entre dois números racionais positivos

```
-- Este programa retorna a divisão entre
-- dois números racionais positivos.
-- sendo que um número rácional r é representado
-- pelo par ordenado (a,b) com b > 0
-- e r = a / b

-- Número racional 1 = (r1, r2) = r1 / r2
-- Numero racional 2 = (r3, r4) = r3 / r4

-- r5, r6 =  (r1 * r4), (r2 * r3) com r3 <> 0

-- O resultado da divisão será r5 / r6

programa Divisao_Rac_Pos_A_B(r1, r2, r3, r4) -> r5, r6
1: faca r5 = Mult_Int_SemSinal(r1, r4) va_para 2
2: faca r6 = Mult_Int_SemSinal(r2, r3) va_para 3
3: retorna
```

Quadro 71 - Programa que retorna a divisão de um número racional positivo por outro

```
-- Este programa verifica se dois números
-- racionais positivos são iguais

-- sendo que um número rácional r é representado
-- pelo par ordenado (a,b) com b > 0
-- e r = a / b

-- Número racional 1 = (r1, r2) = r1 / r2
-- Numero racional 2 = (r3, r4) = r3 / r4

-- (r1, r2) = (r3, r4) se e somente se (r1 * r4) = (r2 * r3)

-- O resultado será 0 (zero) caso os números sejam iguais
-- e 1 (um) caso contrário.

programa Igual_Rac_Pos_A_B(r1, r2, r3, r4) -> r5
1: faca r5 = Mult_Int_SemSinal(r1, r4) va_para 2
2: faca r6 = Mult_Int_SemSinal(r2, r3) va_para 3
3: faca r5 = Comp_Dois_Num_Iguais(r5, r6) va_para 4
4: retorna
```

Quadro 72 - Programa que verifica se dois números racionais positivos são iguais