UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

GERAÇÃO DE CÓDIGO PARA A MÁQUINA VIRTUAL DE RÓTULOS

DOUGLAS RUSKOWSKI HAASE

BLUMENAU
2013

2013/1-13

DOUGLAS RUSKOWSKI HAASE

GERAÇÃO DE CÓDIGO PARA A MÁQUINA VIRTUAL DE RÓTULOS

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
2013

2013/1-13

GERAÇÃO DE CÓDIGO PARA A MÁQUINA VIRTUAL DE RÓTULOS

Por

DOUGLAS RUSKOWSKI HAASE

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:	Profa. Joyce Martins, Mestre – FURB
Membro:	Prof. Miguel Alexandre Wisintainer, Mestre – FURB

Blumenau, 09 de julho de 2013.

AGRADECIMENTOS

Agradeço meus pais, que procuraram me educar da melhor forma possível. Pela iniciativa de garantir que eu tivesse acesso ao estudo e que obtivesse um diploma naquilo que me identificasse. Por este apoio e pelas cobranças, também.

Agradeço minha namorada, Julianna, por estar sempre do meu lado em todos os momentos, por me apoiar, acreditar e incentivar na conclusão deste trabalho, mesmo quando eu já não encontrava ânimo.

Ao meu orientador, José R. V. da Silva, por ter conduzido comigo esta ideia até sua conclusão e por todo o auxílio e dedicação prestados durante a elaboração deste trabalho.

Respect your efforts, respect yourself. Self-respect leads to self-discipline. When you have both firmly under your belt, that's real power.

Clint Eastwood.

RESUMO

Este trabalho consiste na criação de um ambiente de programação para uma Linguagem de Alto Nível (LAN) e uma Linguagem Iterativa (LIT) capaz de serem executadas pela Máquina Virtual de Rótulos (MVR). O trabalho demonstra a criação e os recursos da LAN e da LIT utilizando o Gerador de Analisadores Léxicos e Sintáticos (GALS), que gera uma estrutura inicial de analisador léxico, sintático e semântico para implementação. A LAN e a LIT são convertidas para a Linguagem Intermediária Rotulada (LIR). A LIR e a MVR são propostas elaboradas por Silva (2003, 2012). A especificação das duas linguagens criadas faz uso da notação *Backus-Naur Form* (BNF) para expressar a gramática livre de contexto das linguagens e ambas passam pelo processo de compilação e conversão para a representação monolítica LIR. O resultado convertido é salvo em arquivo compatível para utilização com o ambiente desenvolvido por Silva (2004), que utiliza a LIR. A especificação da ferramenta segue o padrão da *Unified Modeling Language* (UML).

Palavras-chave: Compiladores. Programas iterativos. Programas monolíticos. MVR. LIR. LAN. LIT.

ABSTRACT

This work consists of the creation of a programming environment for a High Level Language (LAN) and an Iterative Language (LIT) capable of running with the Virtual Labels Machine (MVR). The work demonstrates the creation and the resources of the LAN and LIT using the Lexical and Syntactic Analyzers Generator (GALS), which generates an initial structure of lexical, syntactical and semantic analyzers. Both LAN and LIT are converted to Labeled Intermediate Language (LIR). LIR and MVR are proposals developed by Silva (2003, 2012). The specification of the two created languages use the Backus-Naur-Form (BNF) to express the context-free grammar of the two languages and both go through the process of compilation and conversion to the LIR monolithic representation. The converted result is saved in a compatible file for use with the environment developed by Silva (2004), which uses the LIR. The specification of the tool follows the pattern of the Unified Modeling Language (UML).

Key-words: Compilers. Iterative programs. Monolithic programs. MVR. LIR. LAN. LIT.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

BNF - *Backus-Naur-Form*

GALS – Gerador de Analisadores Léxicos e Sintáticos

LAN – Linguagem de Alto Nível

LIR – Linguagem Intermediária Rotulada

LIT – Linguagem ITerativa

MVC – *Model-View-Controller*

MVR – Máquina Virtual de Rótulos

NORMA - *Number theOretic Register MAchine*

RF – Requisito Funcional

RNF – Requisito Não Funcional

UC – *Use Case*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A MVR é uma proposta apresentada em Silva (2003). A proposta desta máquina objetiva a identificação de ciclos infinitos e nós e caminhos mortos, de forma tratável (num tempo admissível), sendo baseada na máquina *Number theOretic Register MAchine* (NORMA) (DIVERIO; MENEZES, 2008, p. 33-37, 70). A máquina NORMA possui uma estrutura de controle iterativa. Entende-se por ciclos infinitos os comandos que nunca levam ao final do programa. Já os nós mortos são comandos que nunca são executados e os caminhos mortos são aqueles não alcançados (percorridos). A memória da referida máquina é composta por um conjunto infinito (máquina teórica) de registradores naturais. A proposta da estrutura da MVR é monolítica, diferindo da abordagem iterativa da máquina NORMA, mas que também utiliza como memória um conjunto de registradores naturais.

A linguagem objeto para a MVR definida é chamada de LIR (SILVA, 2003). A LIR é uma linguagem orientada por instruções. Uma primeira implementação da MVR pode ser vista em Silva (2004). No referido trabalho foi implementada a MVR e um ambiente para desenvolvimento de programas escritos em LIR, onde o montador já detecta ciclos infinitos e nós e caminhos mortos. Ainda neste ambiente, existe a opção de execução normal ou passo a passo.

Mediante o descrito acima, este trabalho propõe-se à definição de duas linguagens, a LAN e a LIT (independentes de plataforma – hardware e sistema operacional) e a construção de um compilador para as mesmas, sendo o código objeto gerado em LIR para a MVR. Tanto a programação de origem (alto nível ou iterativa) quanto a conversão para a representação em LIR serão feitas no ambiente desenvolvido. Este resultado será fornecido como entrada para o trabalho de Silva (2004), em que se construiu um ambiente para auxiliar o desenvolvimento de programas monolíticos. No ambiente de Silva (2004) espera-se como entrada instruções rotuladas, as quais são convertidas na forma de instruções rotuladas compostas.

O trabalho proposto também objetiva um melhoramento da didática utilizada para aplicação de exercícios de teoria da computação, mostrando a programação para uma máquina teórica na prática, já que será possível praticar a programação e conversão de linguagens de alto nível para instruções rotuladas.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é o desenvolvimento de uma ferramenta capaz de converter duas linguagens pré-definidas, a LAN e a LIT, em uma representação de baixo nível (LIR) interpretada pela MVR.

Os objetivos específicos do trabalho são:

a) disponibilizar uma linguagem de programação simples de alto nível;
b) disponibilizar uma linguagem de programação simples iterativa;
c) disponibilizar um compilador para as referidas linguagens, convertendo-as para a linguagem objeto LIR.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está organizado em quatro capítulos. No presente capítulo é apresentada uma introdução sobre o trabalho bem como seu objetivo.

O segundo capítulo aborda conceitos de compiladores, máquinas (NORMA e MVR), programas e apresenta a ferramenta GALS. Ainda, são descritos dois trabalhos correlatos.

O terceiro capítulo mostra o desenvolvimento da aplicação, contendo os requisitos do trabalho, especificações das linguagens criadas através da BNF (LAN e LIT), especificações do ambiente, a implementação da ferramenta e os resultados e discussões obtidos ao final do trabalho.

Ao final, o quarto capítulo apresenta as conclusões do presente trabalho, sugerindo ainda extensões para melhoria da ferramenta desenvolvida.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 são abordados conceitos sobre compiladores e suas fases de implementação. A seção 2.2 descreve as máquinas computacionais e sua estrutura para a computação de programas, descrevendo a máquina NORMA e MVR, bem como a LIR para utilização com a MVR. Na seção 2.3 são descritos os tipos de programas e suas estruturas, abrangendo programas iterativos, monolíticos e recursivos, bem como a transformação de programas iterativos em LIR. A seção 2.4 apresenta a ferramenta GALS e seus recursos. Na seção 2.5 são apresentados dois trabalhos correlatos.

2.1 COMPILADORES

Segundo Aho, Setti e Ullman (1995, p. 1), um compilador é um programa que lê uma linguagem fonte e a traduz num programa equivalente em outra linguagem. Apesar da aparente complexidade nesta área devido às múltiplas linguagens, tanto na linguagem fonte quanto na linguagem alvo convertida, as tarefas básicas que um compilador precisa realizar são essencialmente as mesmas. Ainda, como parte importante do processo de compilação é a apresentação dos erros ao usuário, caso ocorrerem, a fim de se iniciarem os ajustes de programação do programa fonte submetido.

As tarefas essenciais de qualquer compilador são (AHO; SETTI; ULLMAN, 1995, p. 1-8):

a) análise léxica: é a primeira fase de um compilador. Consiste na leitura dos caracteres de um programa fonte e agrupamento num fluxo de *tokens* que representam as estruturas léxicas da linguagem, como as palavras chaves, operadores e identificadores. Também realiza tarefas secundárias do nível da interface com o usuário, removendo espaços excedentes, tabulações e caracteres de controles;
b) análise sintática: esta tarefa obtém as cadeias de *tokens* oriundas do analisador léxico e verifica se elas podem ser geradas pela gramática da linguagem fonte. Como parte essencial deste processo, espera-se que o analisador sintático apresente todos os erros de sintaxe encontrados de forma compreensível para correção do

usuário;

c) análise semântica: são realizadas verificações para assegurar que os componentes do programa escrito combinam-se de forma significativa. Assim, esta fase verifica os erros semânticos no programa fonte como os identificadores e variáveis não declarados e incompatibilidade de tipos;
d) gerador de código intermediário: alguns compiladores, após as fases anteriores, geram uma representação intermediária do programa fonte. Pode-se imaginar como um programa para uma máquina abstrata. Gera-se uma sequência de código que pode ter propriedades e formas diferentes de interpretação em cada compilador, como por exemplo, o código de três endereços. Este código consiste numa sequência de instruções, cada uma delas possuindo no máximo três operandos. Esta fase também trata dos problemas da passagem de código fonte para código objeto;
e) otimizador de código: aprimora o código intermediário gerado na fase anterior, tornando-o mais rápido em tempo de execução. Existem vários tipos de otimizações e algoritmos diferentes para esta finalidade, sendo gasta uma porção significativa de processamento nesta fase;
f) gerador de código: nesta tarefa o compilador gera o código alvo. São selecionadas as localizações de memória para cada uma das variáveis utilizadas pelo programa e traduzidas as instruções intermediárias em instruções de máquina que realizam a mesma tarefa especificada na linguagem do programa fonte.

2.2 MÁQUINAS

A máquina computacional deve suprir todas as informações necessárias para que a computação de um programa possa ser descrita (DIVERIO; MENEZES, 2008, p. 20-21). A máquina é responsável pela atribuição de significado aos identificadores das operações, o que caracteriza uma transformação na estrutura da memória da máquina. Cada identificador de teste interpretado pela máquina deve ser associado a uma função verdade. A máquina também deve descrever o armazenamento ou recuperação de informações em sua estrutura de memória.

Segundo Diverio e Menezes (2008, p. 21), uma máquina é uma 7-upla representada por $M = (V, X, Y, \pi_X, \pi_Y, \pi_F, \pi_T)$, 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 π_X: X → V;
e) π_Y : função de saída tal que π_Y: V → Y;
f) π_F : conjunto de interpretações de operações onde, para cada identificador de operação F interpretado por M, existe uma única função: π_F: V → 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: π_T: V → {verdadeiro, falso} em π_T.

2.2.1 Máquina NORMA

Para trabalhar com programas iterativos, pode-se fazer uso da máquina NORMA. Proposta por Bird (1976, p. 50), esta é uma máquina de registradores. Possui um conjunto infinito de registradores naturais como memória, dois tipos de operações e um teste.

O teste executado pela máquina NORMA verifica se o valor armazenado em registrador natural é igual a zero. As duas operações que a máquina NORMA pode realizar são:

a) adição do valor um;
b) subtração do valor um.

Segundo Diverio e Menezes (2008, p. 72), a máquina NORMA é universal e extremamente simples, com poder computacional equivalente a qualquer computador moderno.

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

a) atribuição do valor zero a um registrador: a atribuição [A:=0] pode ser obtida através do programa iterativo descrito no Quadro 1. Esta atribuição poderá ser usada como uma macro, facilitando a definição de atribuição do valor zero a um registrador;

Quadro 1 - Macro A:= 0

```
Até A = 0
Faça (A:= A-1)
```

Fonte: Diverio e Menezes (2008, p. 73).

b) adição de dois registradores: a macro [A:= A + B] pode ser obtida através do programa iterativo descrito no Quadro 2. 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 3;

Quadro 2 - Macro A := A + B

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

Fonte: Diverio e Menezes (2008, p. 74).

Quadro 3 - Macro A := A + B usando C

```
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 (2008, p. 74).

c) multiplicação de dois registradores: a macro [A:= A x B usando C, D] descrita no Quadro 4, executa multiplicação de A por B, usando dois registradores de trabalho C e D;

Quadro 4 - Macro A:= A x B usando C, D

```
C:= 0
Até A = 0
Faça (C:= C + 1; A:= A - 1);
Até C = 0
Faça (A:= A+B usando D; C:= C -1)
```

Fonte: Diverio e Menezes (2008, p. 75).

testa se o valor de um registrador é um número primo: a macro [teste_primo(A) usando C] descrita no Quadro 5 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, caso contrário, retorna o valor falso.

Quadro 5 - Macro teste_primo(A) usando C

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

Fonte: Diverio e Menezes (2008, p. 75).

2.2.2 MVR

A MVR foi criada para dar suporte a programas monolíticos na forma de instruções rotuladas. Os programas monolíticos para serem executados na MVR devem estar representados na LIR. A memória da MVR é composta por registradores naturais. A MVR possui comandos do tipo instrução e do tipo controle (SILVA, 2004, p. 49-50).

Os comandos do tipo instrução são:

a) atribuição de um valor para um registrador;
b) atribuição do valor de um registrador para outro registrador;
c) adição do valor 1 para um registrador;
d) subtração do valor 1 para um registrador;
e) teste: verifica se o valor de um registrador é igual a 0.

Os comandos do tipo controle são os seguintes:

a) chamada de macros (sub-rotinas);
b) desvios condicionais;
c) instrução de retorno, a qual finaliza o programa em LIR. Um exemplo de uso da instrução de retorno pode ser visto na instrução de número dez do Quadro 7.

A MVR executa os programas desenvolvidos em LIR. No início da execução é possível fornecer os valores naturais (parâmetros) como entrada para os registradores (SILVA, 2004, p. 50).

A LIR é a linguagem objeto executada pela MVR. Os registradores da LIR devem ser iniciados com a letra “r” ou “R”, seguido de um número natural ou da letra “t” ou “T”. Exemplos de nomes dos registradores são `rt`, `r1`, `r2`, `r3`,..., `rn`. A linguagem não é *case sensitive*. 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> (<parâmetros>) -> <saída>)` onde (SILVA, 2004, p. 50):

a) `<nome do programa>`: é um conjunto de caracteres iniciado por uma letra e após esta, é seguido por letras, números ou por caracteres do tipo sublinhado (“_”), sendo que não pode haver um nome de programa com o formato de um registrador;
b) `<parâmetros>`: é um conjunto finito de registradores, separados uns dos outros por vírgula;

c) <saída>: é um conjunto finito de registradores, separados uns dos outros por vírgula. A função de saída deve possuir pelo menos um registrador.

Com estas definições pode-se criar o seguinte cabeçalho de programa em LIR: programa exemplo_programa (r1, r2) -> r4. Após o cabeçalho definido pode-se então definir um conjunto de instruções rotuladas de operações e testes.

Um comando em LIR tem a seguinte forma: <rotulo>: faca <comando> va_para <rotulo desvio>, onde:

a) <rotulo>: é um número natural que identifica a instrução rotulada, sendo que um rótulo pode ser atribuído a somente uma instrução rotulada;

b) <comando>: um comando que utiliza um registrador pode ser uma das seguintes formas:

- inc(<registrador>): esta operação incrementa de 1 o valor do registrador passado como parâmetro. Um exemplo pode ser visto no Quadro 6, linha 1,
- dec(<registrador>): esta operação decrementa de 1 o valor do registrador passado como parâmetro, se o seu valor for maior que zero. Um exemplo pode ser visto no Quadro 6, linha 2,
- <registrador> = <registrador>: esta operação atribui o valor de um registrador a outro registrador. Um exemplo pode ser visto no Quadro 6, linha 3,
- <registrador> = <valor natural>: esta operação atribui um valor natural a um registrador. Um exemplo pode ser visto no Quadro 6, linha 4,
- <lista de registradores> = <macro>: esta operação permite a chamada de uma macro. Quando ocorrer a chamada de uma macro, a função de saída da mesma será atribuída a lista de registradores. Um exemplo pode ser visto no Quadro 6, linha 5;

Quadro 6 - Exemplos de instrução do tipo operação em LIR

```
1: Faca inc(R2) va_para 2
2: Faca dec(R1) va_para 3
3: Faca R1 = R3 va_para 4
4: Faca R2 = 10 va_para 5
5: Faca R1 = Soma_Sem_Sinal(R1, R2) va_para 1
```

c) <rótulo desvio>: este rótulo é um número natural, que aponta para uma instrução rotulada.

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.

Para finalizar um programa ou uma macro 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. Os registradores que foram especificados na função de <saída> terão seu valor exibido ao final da execução do programa.

O Quadro 7 apresenta um programa em LIR para execução pela MVR que compara se dois números naturais são iguais, onde é possível visualizar desde o cabeçalho do programa até a utilização de testes e da instrução de retorno.

Quadro 7 - Programa que compara se dois números são iguais

```
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
```

2.3 PROGRAMAS

Os programas são parte essencial para a utilização das máquinas. O conceito de programa mais básico consiste num conjunto de operações e testes escritos de acordo com uma estrutura de controle (DIVERIO; MENEZES, 2008, p. 10). Este conjunto permite a uma máquina aplicar sucessivamente as instruções definidas até que os dados iniciais fornecidos resultem na saída esperada.

O que define a classificação de programas e consequentemente das máquinas é o tipo de estrutura de controle associada, as quais são (DIVERIO; MENEZES, 2008, p. 10-11):

a) monolítica: é estruturada usando desvios condicionais e incondicionais, não fazendo uso explícito de mecanismos auxiliares de programação. A lógica é distribuída por todo o bloco (monólito) que constitui o programa;
b) iterativa: são utilizados métodos de controle de iterações em trechos dos programas, não permitindo o uso de desvios incondicionais;
c) recursiva: a estrutura recursiva é uma técnica de indução de programas. Antes do final da execução de uma função, a mesma é invocada novamente de forma direta ou indireta. Nas linguagens de programação o conceito de programa recursivo aparece associado ao de sub-rotinas recursivas. Assim como a iterativa, a estrutura recursiva não permite desvios incondicionais.

2.3.1 Programas monolíticos

Programas monolíticos podem ser representados de duas formas: através de instruções rotuladas ou através de fluxogramas. As instruções rotuladas constituem uma notação formal para a representação de programas monolíticos. A definição formal de programas monolíticos é melhor descrita usando a notação de instruções rotuladas do que fluxogramas (DIVERIO; MENEZES, 2008, p. 14). Uma instrução rotulada é uma sequência de símbolos que pode representar:

a) operação: indica a operação que será executada seguida de um desvio incondicional para a próxima instrução;
b) teste: determina um desvio condicional na execução. É feita a avaliação especificada pelo teste e executada a instrução seguinte de acordo com seu resultado;
c) parada: determina o fim do bloco monolítico. Especifica-se um desvio incondicional para um rótulo sem instrução associada.

No Quadro 8 é apresentado um exemplo de programa monolítico representado através de instruções rotuladas.

Quadro 8 - Conjunto de instruções rotuladas

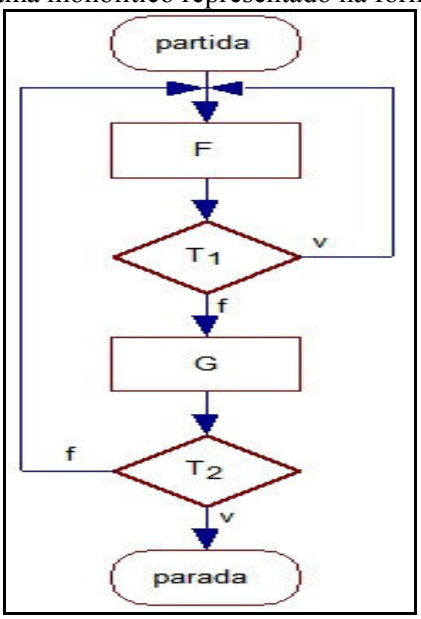
```
1: faca F va_para 2
2: se T1 entao va_para 1 senao va_para 3
3: faca G va_para 4
4: se T2 entao va_para 5 senao va_para 1
```

Fonte: Diverio e Menezes (2008, p. 49).

Na instrução rotulada número 4, do Quadro 8, existe um desvio para o rótulo 5. Entretanto, 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, 2008, p. 13). A representação de programas na forma de instruções rotuladas é útil para manipulações diversas dos programas monolíticos, como a verificação de ciclos infinitos, nós mortos, simplificação de código e a equivalência entre programas.

Programas monolíticos também podem ser representados por fluxogramas. Como um programa monolítico é definido usando a noção de fluxograma, ambos os termos são identificados e usados indistintamente. Um exemplo de representação em fluxograma de um programa monolítico é apresentado na Figura 1. Este fluxograma representa o programa apresentado no Quadro 8.

Figura 1 – Programa monolítico representado na forma de fluxograma



Fonte: Diverio e Menezes (2008, p. 13).

2.3.2 Programas Iterativos

Programas iterativos são baseados em três mecanismos de composição (sequenciais) de programas (DIVERIO; MENEZES, 2008, p. 16). A composição sequencial de dois programas resulta em um terceiro, cujo efeito é a execução do primeiro e, após, a execução do

segundo bloco componente. Os mecanismos de composições sequenciais são:

a) sequencial: a execução da operação ou teste subsequente somente pode ser realizada após o término da execução da operação ou teste anterior;
b) condicional: há dois blocos componentes em que somente um será executado, dependendo do resultado de um teste. O resultado desta execução é um terceiro bloco componente;
c) enquanto: composição de um programa resultando em um segundo cujo efeito é a execução, repetidamente, do bloco componente enquanto o resultado de um teste for verdadeiro.

De forma análoga à composição "enquanto", pode ser conveniente a definição da composição "até". Assim o término da iteração é causado pelo retorno do valor verdadeiro para o teste que será executado. O mecanismo de composição "até" é definido da seguinte maneira: enquanto o resultado de um teste for falso, o bloco componente é executado e quando o resultado do teste for verdadeiro, é finalizada a execução do bloco componente.

Segundo Diverio e Menezes (2008, 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. O objetivo é substituir os desvios incondicionais por estruturas de controle de repetição, o que resulta em uma melhor estruturação dos desvios e consequente legibilidade facilitada dos programas. No Quadro 9 é mostrado um programa iterativo, onde T_1, T_2 e T_3 são identificadores de teste e F e G são operações.

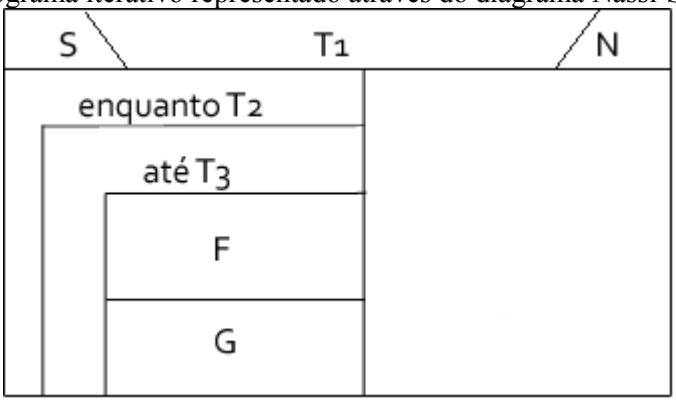
Quadro 9 – Programa iterativo

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

Fonte: Diverio e Menezes (2008, p. 18).

Um programa iterativo também pode ser representado através de diagramas de Nassi-Schneidermann (NASSI; SCHNEIDERMANN, 1973). A Figura 2 apresenta o programa iterativo representado no Quadro 9, através do diagrama de Nassi-Schneiderman.

Figura 2 - Programa iterativo representado através do diagrama Nassi-Schneiderman



A definição proposta por Silva (2012) cria uma representação intermediária para os programas iterativos desenvolvidos em LIT. Esta representação é armazenada em forma de tabela com valores inteiros, sendo posteriormente usada para conversão para LIR. Para demonstrar a criação desta tabela é definida inicialmente a especificação de programas iterativos (LIT), que pode ser vista no Quadro 10.

Quadro 10 - Definição de programa iterativo

```
Prog-Itr:= NOME_PROGRAMA (@IDS) -> (@IDS) CMD COMANDOS.
COMANDOS:= CMD COMANDOS | ε.
CMD:= "(" TESTE ")" COND |
      "ATE" COND BLOCO-REP |
      "EQTO" COND BLOCO-REP |
      @ID ("+" | "-").
BLOCO-REP:= "FACA" "(" COMANDOS ")" ENCHIMENTO.
TESTE:= "SE" COND "ENTAO" COMANDOS ENCHIMENTO "SENAO" COMANDOS.
COND:= @IDS "=" "0".
```

Fonte: Silva (2012).

Ações semânticas são acrescidas na especificação do Quadro 10, resultando na definição do Quadro 11.

Quadro 11 - Definição para representação intermediária iterativa

```
Prog-ltr:=          {inc}                          // incrementa 1 no ender que começa de zero
NOME_PROGRAMA PARAMETROS "->" RETORNOS CMD COMANDOS.
COMANDOS:= CMD COMANDOS | ε.
CMD:= "(" TESTE ")"
                    {C4(Pop1(), ender)}            // atualiza o próximo do último comando do então
                    {C5(Pop1(), Pop2())}           //atualiza "senão" do teste
          |
          "ATE" COND
                    {C0(ender)}                    // armazena endereço atual
                    {C1(1)}                        // instrução do tipo teste (1)
                    {C5(ender, succ)}              // inicio do bloco do enquanto
                    {Push1(ender)}                 //empilha até
                    {inc}                          // incrementa 1 no ender (rótulo)
                    BLOCO-REP
                    {C4(Front1(), ender)}          // atualiza saída do até
                    {C4(pred, Pop1())}             // atualiza a próxima instrução do último comando do bloco enquanto
                    {inc}                          // incrementa 1 no ender (rótulo)
          |
          "EQTO" COND
                    {C0(ender)}                    // armazena endereço atual
                    {C1(1)}                        // instrução do tipo teste (1)
                    {C4(ender, succ)}              // inicio do bloco do enquanto
                    {Push1(ender)}                 //empilha enquanto
                    {inc}                          // incrementa 1 no ender (rótulo)
                    BLOCO-REP
                    {C5(Front1(), ender)}          // atualiza saída do enquanto
                    {C4(pred, Pop1())}             // atualiza a próxima instrução do último comando do bloco enquanto
                    {inc}                          //alteração em
          |
          @ID       {C0(ender)}                    // cria operação
                    {C2(nr)}                       // receptor (registrador que será incrementado ou decrementado 1)
                    {C4(ender, succ)}              // aponta para o próximo comando
          ("+"      {C1(ender, 3)}                 // instrução do tipo operação de incremento (3)
          | "-"     {C1( 4)}                       // instrução do tipo operação de decremento (4)
          )         {inc}                          // incrementa 1 no ender (rótulo)
          .
BLOCO-REP:=         "FACA" "(" COMANDOS ")"
                    ENCHIMENTO.

ENCHIMENTO:=        {C0(ender)}
                    {C1(0)}                        // instrução nula
                    {C4(ender, succ)}              //próxima
TESTE:= "SE" COND
                    {C0(ender)}                    // armazena endereço atual
                    {C1(1)}                        // instrução do tipo teste (1)
                    {C4(ender, succ)}              // inicio do "então"
                    {Push1(ender)}                 //empilha teste
                    {inc}                          // incrementa 1 no ender (rótulo)
          "ENTAO" COMANDOS ENCHIMENTO
                    {Push1(ender)}                 //empilha endereço do enchimento
                    {inc}
          "SENAO"
                    {Push2(ender)}                 //empilha endereço do início do senão na pilha 2
                    COMANDOS.
COND:= @IDS
                    {C0(ender)}                    // cria operação atribuição para registrador de teste (R0)
                    {C1(2)}                        // instrução do tipo operação atribuição de registrador (2)
                    {C2(0)}                        // receptor (0-> registrador de teste)
                    {C3(nr)}                       //  enviador (registrador que será copiado o valor)
                    {C4(ender, succ)}              // "então"
                    {inc}.
```

Fonte: Silva (2012).

No Quadro 11 é definida a LIT através da BNF, com ações semânticas para transformação da referida linguagem em uma tabela, onde a partir da mesma será construído o programa em LIR.

As ações semânticas descritas na definição do Quadro 11 são:

a) ID: identificador de registradores de números naturais;
b) ender: endereço da instrução, iniciado em 0;
c) nr: número do registrador retirado do último ID;
d) C0(p): função que atribui o endereço da instrução atual em um índice da tabela;
e) C1(p): função que atribui o parâmetro p, que indica o tipo de comando, em um índice da tabela. O valor 0 indica uma instrução de desvio chamada de enchimento, o valor 1 indica teste, o valor 2 indica atribuição, o valor 3 indica incremento de 1 e o 4 indica decremento de 1;
f) C2(p): função que armazena em um índice da tabela o valor do registrador utilizado em operações de incremento, decremento e teste;
g) C3(p): função que indica o número do registrador utilizado na operação de atribuição em um índice da tabela;
h) C4(p1, p2): função que indica o número do endereço sucessor (parâmetro p2) da instrução atual em um índice da tabela (parâmetro p1);
i) C5(p1, p2): função que indica o número do endereço (parâmetro p2) em um índice da tabela (parâmetro p1) que corresponde à instrução `senao`, caso o resultado do teste for falso;
j) inc(): incrementa em 1 o valor do endereço atual;
k) push1(p): empilha o endereço atual (parâmetro p) na pilha número 1;
l) push2(p): empilha o endereço atual (parâmetro p) na pilha número 2;
m) pop1(): desempilha um endereço da pilha número 1;
n) pop2(): desempilha um endereço da pilha número 2;
o) front(): lê o topo da pilha e armazena o valor em um registrador temporário;
p) succ: função que indica o endereço sucessor à instrução atual.

Com as ações definidas anteriormente é possível representar na tabela de valores inteiros (Quadro 12) os valores obtidos com a execução das ações semânticas. A partir destas ações é criada a tabela com 6 colunas (C0, C1, C2 e C3, C4 e C5) e *x* linhas (*x* corresponde ao valor do número da instrução, armazenado no atributo `ender`). Para cada *token* identificado na especificação do Quadro 10, é realizada uma das ações definidas no Quadro 11.

Assim que a execução terminar, a tabela é convertida para programa monolítico (LIR). A coluna C1 indica o tipo de comando a ser gerado em LIR. Se o valor da coluna C1 for 0 é criada uma operação que zera o valor do registrador de teste e aponta para a próxima instrução. Se o valor da coluna C1 for 1 é criada uma operação de teste. O valor 2 indica uma

operação de atribuição de registrador. Já os valores 3 e 4 correspondem, respectivamente, a operação de incremento e decremento.

O Quadro 12 apresenta um programa iterativo exemplo, a partir do qual é construída a tabela. Na primeira linha é representada a operação `R1+`. O valor da coluna C0 é 1, pois corresponde a primeira instrução. O valor da coluna C1 é 3, pois trata-se de uma operação de incremento. O valor da coluna C2 é 1, pois é o número do registrador utilizado na operação de incremento. A coluna C3 não é utilizada. Esta coluna só é utilizada para armazenar os números dos registradores utilizados em instruções de teste. A coluna C4 possui o valor 2, pois é o desvio para a próxima instrução do programa conforme implementação da ação semântica `ID`. O valor da coluna C5 não é utilizado, pois não há desvio condicional falso. Na linha seguinte é feita a atribuição do valor do registrador `R1` para um registrador de teste. O valor da coluna C0 é 2, pois é a próxima instrução a partir da número 1. A coluna C1 possui valor 2 que indica uma instrução de atribuição de registrador. A coluna C2 possui valor 0 que indica que o valor do registrador `R1` será armazenado em um registrador de teste. A coluna C3 possui valor 1 que indica o número do registrador que envia o valor para o registrador de teste. A coluna C4 com o valor 3 indica o desvio para a próxima instrução. A coluna C5 não é utilizada, pois ainda não foi feita a instrução de teste, apenas a atribuição do registrador que será testado. A linha de número 3 corresponde à instrução de teste após a atribuição do registrador a partir de `R1`. O valor da coluna C0 é 3, pois é a terceira instrução. O valor da coluna C1 é 1, que indica uma instrução de teste. Não são utilizadas as colunas C2 e C3, pois não há cópia de valores entre registradores. O valor 4 na coluna C4 indica a instrução de desvio para verdadeiro, como resultado do teste. O valor 16 na coluna C5 indica a instrução de desvio falso. A coluna C5 só será atualizada quando for identificado o final da instrução de teste pelas ações semânticas.

Quadro 12 - Programa iterativo transformado em representação intermediária

```
PROGRAMA_TESTE(R1)->R2
R1+;
( SE R1
  ENTÃO  EQTO R1
         FAÇA ( R2-;
                EQTO R2
                FAÇA  ( R2-;
                        R3+
                      );
                R4-
               );
         R5+
  SENÃOR1-
);
ATÉ R2
FAÇA  ( R1+
      )
```

Ender: ~~1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 29 20~~ 21

	ender C0	Comando C1	Receptor C2	Sender C3	Então C4	Senão C5	
1	1	3	1		2		R1+
2	2	2	0	1	3		R0=R1
3	3	1			4	16	se
4	4	2	0	1	5		R0=R1
5	5	1			6	13	Eqto
6	6	4	2		7		R2-
7	7	2	0	2	8		R0=R2
8	8	1			9	11	Eqto
9	9	4	2		10		R2-
10	10	3	3		8		R3+
11	11	0			12		enchimento
12	12	4	4		5		R4-
13	13	0			14		enchimento
14	14	3	5		15		R5+
15	15	0			17		enchimento
16	16	4	1		17		R1-
17	17	2	0	2	18		R0=R2
18	18	1			20	19	Até
19	19	3	1		18		
20	0				21		
21							

Pilha1

18
15
12
8
5
3

Pilha2

16

Fonte: Silva (2012).

A partir da tabela representada no Quadro 12 é possível construir o programa monolítico usando a LIR. Os valores de cada linha e coluna são transformados em instruções monolíticas. Analisando a primeira linha, por exemplo, nota-se que o valor da coluna C1 é 3. Portanto, é criada uma operação de incremento em LIR com a seguinte forma: `{C0 : faca inc(rC2) va_para C4}`, o que resulta na instrução em LIR `1: faca inc(r1) va_para 2`. A próxima linha corresponde a uma operação de atribuição de registrador, já que o valor da coluna C1 é 2. A operação de atribuição segue a forma: `{C0: faca rT = rC3 va_para C4}`, resultando na instrução em LIR `2: faca rT = r1 va_para 3`. A terceira linha da tabela do

Quadro 12 corresponde à operação de teste, que segue a forma: `{C0 : se T entao va_para C4 senao va_para C5}`, o que resulta na instrução em LIR `3: se T entao va_para 4 senao va_para 16`. Assim sucessivamente todas as linhas da tabela são analisadas e convertidas para LIR. A operação de decremento segue a mesma forma da operação de incremento, com a diferença de usar-se o termo `dec` em vez de `inc`.

O Quadro 13 apresenta o programa do Quadro 12 totalmente convertido para LIR.

Quadro 13 - Programa iterativo transformado em LIR

```
programa PROGRAMA_TESTE(R1) -> R2
1: faca inc(r1) va_para 2
2: faca rt = r1 va_para 3
3: se T entao va_para 4 senao va_para 16
4: faca rt = r1 va_para 5
5: se T entao va_para 6 senao va_para 13
6: faca dec(r2) va_para 7
7: faca rt = r2 va_para 8
8: se T entao va_para 9 senao va_para 11
9: faca dec(r2) va_para 10
10: faca inc(r3) va_para 8
11: faca rt = 0 va_para 12
12: faca dec(r4) va_para 5
13: faca rt = 0 va_para 14
14: faca inc(r5) va_para 15
15: faca rt = 0 va_para 17
16: faca dec(r1) va_para 17
17: faca rt = r2 va_para 18
18: se T entao va_para 20 senao va_para 19
19: faca inc(r1) va_para 18
20: faca rt = 0 va_para 21
21: retorna
```

2.3.3 Programas recursivos

A recursão é uma forma indutiva de definir programas. Possui mecanismos de estruturação em sub-rotinas recursivas e não possibilita o uso de desvios incondicionais (DIVERIO; MENEZES, 2008, p. 18).

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. O Quadro 14 exemplifica um programa recursivo, onde R e S são sub-rotinas.

Quadro 14 - Programa recursivo

```
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 (2008, p. 20).

A computação de um programa recursivo consiste na avaliação da expressão inicial onde cada identificador de sub-rotina referenciado é substituído pela correspondente expressão que o define, e assim sucessivamente, até que seja substituído pela operação vazia (√), determinando o fim da recursão.

No Quadro 15 é possível visualizar a definição da máquina de um registrador (`um_reg`).

Quadro 15 - Máquina de um registrador

$$um\_reg = (\mathbb{N}, \mathbb{N}, \mathbb{N}, id_{\mathbb{N}}, id_{\mathbb{N}}, \{ad, sub\}, \{zero\})$$

onde:

$\mathbb{N}$ corresponde, simultaneamente, aos conjuntos de valores de memória, entrada e saída

$id_{\mathbb{N}}: \mathbb{N} \rightarrow \mathbb{N}$ é a função de entrada e de saída

$ad: \mathbb{N} \rightarrow \mathbb{N}$ é interpretação tal que, $\forall n \in \mathbb{N}$, $ad(n) = n+1$

$sub: \mathbb{N} \rightarrow \mathbb{N}$ é interpretação tal que, $\forall n \in \mathbb{N}$:

$sub(n) = n-1$, se $n \neq 0$; $sub(n) = 0$, se $n = 0$

$zero: \mathbb{N} \rightarrow \{verdadeiro, falso\}$ é interpretação tal que, $\forall n \in \mathbb{N}$:

$zero(n) = verdadeiro$, se $n = 0$; $zero(n) = falso$, caso contrário

Fonte: Diverio e Menezes (2008, p. 27).

No Quadro 16 é apresentado o programa recursivo `duplica` e a computação do mesmo na referida máquina `um_reg`. Para um valor inicial de memória igual a três, nota-se que o valor final da memória é o valor inicial duplicado.

Quadro 16 - Computação finita de programa recursivo

```
                    Programa Recursivo duplica
          duplica é R onde
                R def (se zero então ✔ senão sub;R;ad;ad)
(R;✔, 3)
((se zero então ✔ senão (sub;R;ad;ad));✔, 3)
((sub;R;ad;ad);✔, 3)
(sub;(R;ad;ad);✔, 3)
((R;ad;ad);✔, 2)
(R;(ad;ad);✔, 2)
((se zero então ✔ senão (sub;R;ad;ad));(ad;ad);✔, 2)
((sub;R;ad;ad);(ad;ad);✔, 2)
(sub;(R;ad;ad);(ad;ad);✔, 2)
((R;ad;ad);(ad;ad);✔, 1)
(R;(ad;ad);(ad;ad);✔, 1)
((se zero então ✔ senão(sub;R;ad;ad));(ad;ad);(ad;ad);✔,
((sub;R;ad;ad));(ad;ad);(ad;ad);✔, 1)
(sub;(R;ad;ad);(ad;ad);(ad;ad);✔, 1)
((R;ad;ad);(ad;ad);(ad;ad);✔, 0)
(R;(ad;ad);(ad;ad);(ad;ad);✔, 0)
((se zero então ✔
   senão (sub;R;ad;ad));(ad;ad);(ad;ad);(ad;ad);✔, 0)
(✔;(ad;ad);(ad;ad);(ad;ad);✔, 0)
((ad;ad);(ad;ad);(ad;ad);✔, 0)
(ad;(ad;(ad;ad);(ad;ad);✔), 0)
((ad;(ad;ad);(ad;ad);✔), 1)
(ad;((ad;ad);(ad;ad);✔), 1)
(((ad;ad);(ad;ad);✔), 2)
(ad;(ad;(ad;ad);✔), 2)
((ad;(ad;ad);✔), 3)
(ad;((ad;ad);✔), 3)
(((ad;ad);✔), 4)
(ad;((ad);✔), 4)
(((ad);✔), 5)
(ad;(✔), 5)
((✔), 6)
(✔, 6)
```

Fonte: Diverio e Menezes (2008, p. 28).

2.4 TRABALHOS CORRELATOS

Nesta seção são apresentados dois trabalhos correlatos. O primeiro serve como base para a construção do ambiente gerador de código para a MVR e o segundo demonstra a conversão de programas monolíticos para programas recursivos.

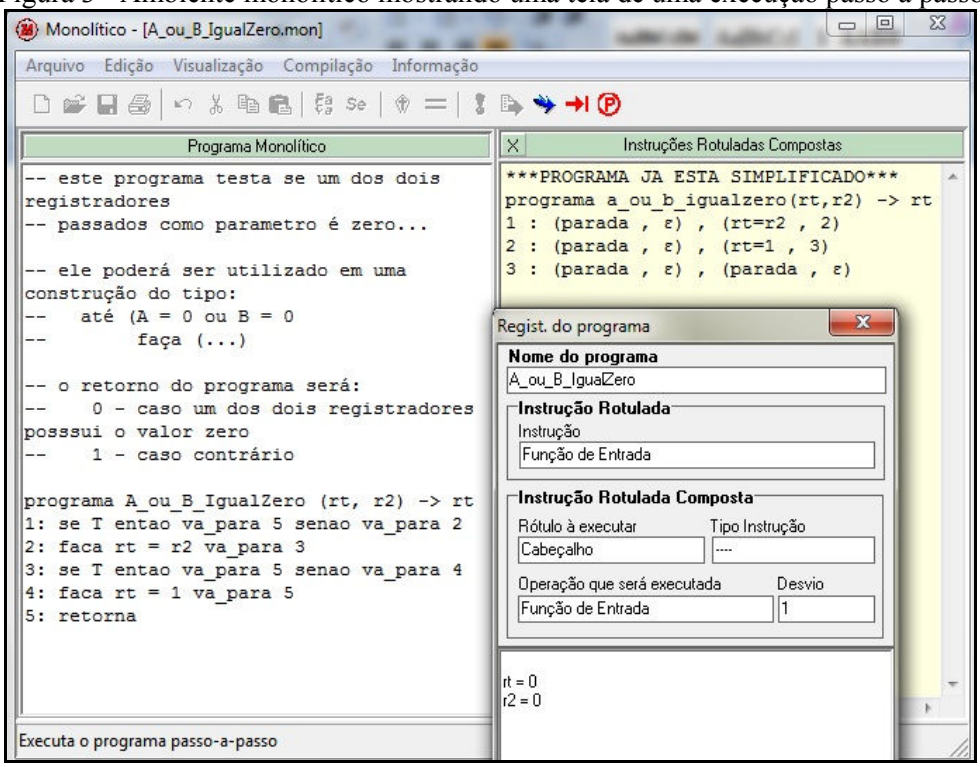
2.4.1 Ambiente para auxiliar o desenvolvimento de programas monolíticos

Desenvolvido por Silva (2004), no ambiente Delphi 7.0 e com a linguagem Object Pascal, o trabalho consiste em um ambiente para desenvolvimento de programas monolíticos

utilizando como memória somente registradores que comportam números naturais. Foi aplicado um algoritmo proposto por Silva (2003) para a transformação de um programa monolítico descrito na forma de instruções rotuladas para instruções rotuladas compostas.

A ferramenta permite a escrita e a compilação de programas monolíticos na forma de instruções rotuladas, sendo verificada a existência de erros léxicos e sintáticos. Após esta etapa, o programa é convertido na forma de instruções rotuladas compostas. Dispõe ainda de recursos para verificação de ciclos infinitos e instruções mortas apresentados em janelas modais. É possível executar o programa passo a passo, observando os rótulos, tipo de instrução (operação ou teste) que será executada e os resultados obtidos nos registradores. Ao final, exibe o programa simplificado na forma de instruções rotuladas compostas. Na Figura 3 é apresentado um exemplo da interface do ambiente em uso, com a tela que exibe a execução passo a passo do programa monolítico.

Figura 3 - Ambiente monolítico mostrando uma tela de uma execução passo a passo



Fonte: Silva (2004).

A principal contribuição deste trabalho é a aplicação do algoritmo proposto em Silva (2003), 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, conforme

mostrado em Diverio e Menezes (2008, p. 47-49), para somente depois convertê-lo em instruções rotuladas compostas.

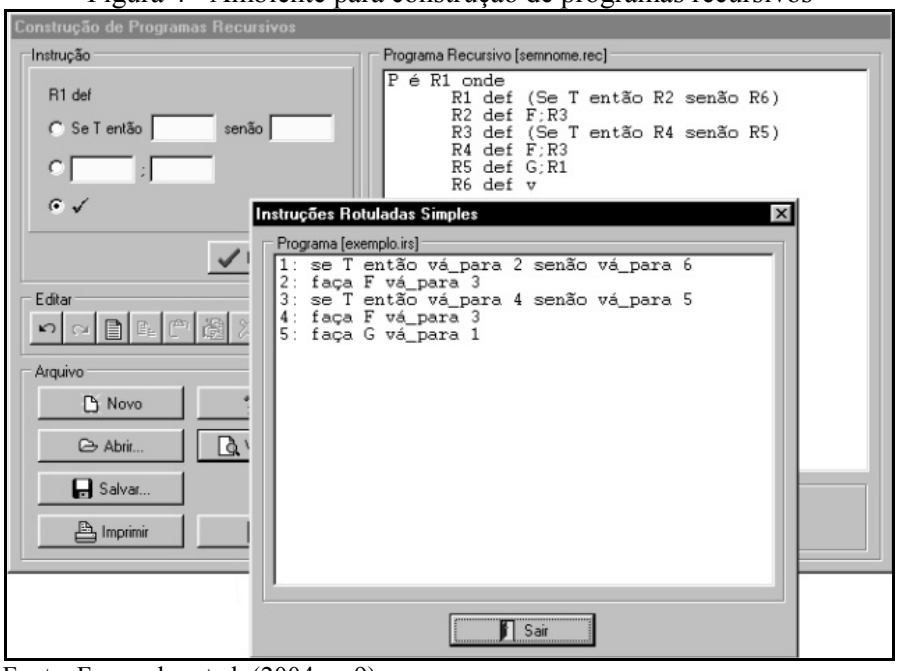
2.4.2 Construção de programas recursivos

O trabalho desenvolvido por Fernandes et al. (2004) foi implementado utilizando a linguagem Object Pascal. A ferramenta converte programas monolíticos na forma de instruções rotuladas em programas recursivos. É apresentado ainda um breve estudo descrevendo a classificação de programas.

O programa recursivo resultante pode ser editado, permitindo a adição e a remoção de novas instruções, refazer ou desfazer operações e imprimir o código recursivo. É possível visualizar a transformação passo a passo do programa de entrada fornecido. Conta ainda com recurso de equivalência entre dois programas na forma de instruções rotuladas. O ambiente disponibiliza ainda uma documentação básica de ajuda para utilização do sistema.

Na Figura 4 é apresentado um exemplo da interface do ambiente em execução.

Figura 4 - Ambiente para construção de programas recursivos



Fonte: Fernandes et al. (2004, p. 9).

Ainda, na Figura 4 observa-se a visualização do programa monolítico em instruções rotuladas simples fornecidas como entrada para conversão em programa recursivo.

2.4.3 Características dos trabalhos correlatos

No Quadro 17 é apresentado um comparativo das características existentes no ambiente para construção de programas recursivos (FERNANDES et al., 2004) e no ambiente de desenvolvimento de programas monolíticos (SILVA, 2004).

Quadro 17- Comparativo entre os dois ambientes de desenvolvimento

Descrição do recurso	**Ambiente recursivo**	**Ambiente monolítico**
Conversão de programas monolíticos para recursivos	X	
Conversão de programas monolíticos para Instruções Rotuladas Compostas		X
Simplificação de programas recursivos	X	
Simplificação de programas monolíticos		X
Demonstração passo a passo da simplificação de programas	X	X
Verificação de equivalência entre programas monolíticos	X	X
Verificação de nós e caminhos mortos em programas monolíticos		X
Suporte a utilização de macros		X
Edição do programa simplificado (resultante)	X	X
Inserção de instruções com recursos da interface do programa	X	X
Visualização e impressão dos programas convertidos	X	X

3 DESENVOLVIMENTO DO AMBIENTE

Este capítulo trata das etapas de desenvolvimento do ambiente gerador de código para a MVR. São abordados os requisitos, especificação da LIT, especificação da LAN, especificação do ambiente, implementação e, por fim, resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos para o ambiente gerador de código para a MVR são:

a) possuir um editor para escrita de programas em LAN e LIT (Requisito Funcional - RF);
b) analisar e compilar os programas escritos, apresentando erros léxicos, sintáticos e semânticos (RF);
c) converter os programas escritos em LAN e LIT para um programa monolítico descrito na forma de LIR (RF);
d) exibir em tela o resultado do programa na linguagem objeto LIR (RF);
e) salvar o código objeto LIR em arquivo de formato compatível com o ambiente desenvolvido por Silva (2004) (RF);
f) ser implementado utilizando a linguagem de programação Java (Requisito Não Funcional - RNF).
g) ser implementado utilizando o ambiente Eclipse (RNF).

3.2 ESPECIFICAÇÃO DA IMPLEMENTAÇÃO DA LIT

A LIT foi especificada por Silva (2012), conforme apresentado no Quadro 10 e no Quadro 11. Esta especificação foi adaptada para a ferramenta GALS, que gerou todas as classes do analisador léxico, sintático e uma estrutura inicial contendo `switch` e `case` para controlar as ações semânticas numeradas, sendo necessário implementá-las.

A linguagem é *case sensitive* e possui um número infinito de registradores de memória

e as operações de incremento e decremento sobre cada um desses. A LIT possui comandos de teste e de repetição.

No Quadro 18 são mostradas as definições regulares da LIT que são utilizadas na definição dos símbolos terminais ou constantes, onde:

a) `letra`: é uma letra do alfabeto, sendo apenas maiúscula;
b) `d`: é um número natural compreendido entre os valores 0 e 9;
c) `plus`: símbolo que identifica o incremento de registrador;
d) `minus`: símbolo que identifica o decremento de registrador.

Quadro 18 - Definições regulares da LIT

```
letra: [A-Z]

d: [0-9]

plus: "+"
minus: "-"
```

No Quadro 19 são mostrados os *tokens* (símbolos terminais) da linguagem, onde:

a) `id`: representa um registrador, formado pela letra inicial "R" e um ou mais dígitos;
b) `operacao`: representa a operação de incremento ou decremento de registrador;
c) `palavra_reservada`: define uma palavra reservada que é constituída por letras maiúsculas;
d) `nome_programa`: define a formação do nome do programa, iniciado por uma letra, seguido por letras, *underline* ou dígitos;
e) `[\s \t \n]`: notação para que o compilador ignore os caracteres de tabulação;
f) `R`, `FACA`, `ATE`, `EQTO`, `SE`, `ENTAO`, `SENAO`: *tokens* que são palavras reservadas da LIT;
g) `(`, `)`, `->`, `.`, `,`: símbolos especiais da LIT.

Quadro 19 - Símbolos terminais da linguagem iterativa

```
id: R {d} {d}*

operacao: {plus} | {minus}

palavra_reservada: {letra}+

nome_programa: ({letra} ({letra} | _ | {d})*)

//tabulacao
:[\s \n \t]

// palavras reservadas

R = palavra_reservada: "R"
FACA = palavra_reservada: "FACA"
ATE = palavra_reservada: "ATE"
EQTO = palavra_reservada: "EQTO"
SE = palavra_reservada: "SE"
ENTAO = palavra_reservada: "ENTAO"
SENAO = palavra_reservada: "SENAO"

// simbolos especiais
"("
")"
"->"
"."
";"
","
```

No Quadro 20 são apresentadas as produções da gramática definida para a LIT criada, juntamente com as ações semânticas, utilizando a notação BNF, onde: uma ação semântica na gramática é iniciada pelo caractere “#” e seguida por um número natural. A ação semântica #1, por exemplo, é responsável por iniciar o controle do endereçamento das instruções que serão criadas, incrementando um valor natural ao registrador que inicia com valor 0. Este valor representa o endereço de cada uma das instruções, sendo incrementado em 1 a cada nova instrução.

Quadro 20 - Gramática da linguagem iterativa

```
<programa_iterativo> ::= #1 nome_programa <parametros> "->" <retornos> <cmd> <comandos> "." ;

<parametros> ::= ł | "(" <ids_parametros> ")" ;
<ids_parametros> ::= id #12 <ids_parametros_aux> ;
<ids_parametros_aux> ::=  ł | "," <ids_parametros> ;

<retornos> ::= id #13 <ids_retornos> ;
<ids_retornos> ::= ł | "," <retornos> ;

<comandos> ::= <cmd> <comandos> | ł ;

<cmd> ::= "(" <teste> ")" #2 | ATE <cond> #3 <bloco_rep> #4 | EQTO <cond> #5 <bloco_rep> #6 | id #7 operacao #8 "." ;

<bloco_rep> ::= FACA "(" <comandos> ")" ;

<teste> ::= SE <cond> #9 ENTAO <comandos> SENAO #10 <comandos> ;

<cond> ::= id #11 ;
```

Fonte: adaptado de Silva (2012).

As ações semânticas apresentadas no Quadro 20 são correspondentes às ações semânticas apresentadas no Quadro 11.

No Quadro 21 é apresentado um programa desenvolvido em LIT.

Quadro 21 - Programa em LIT

```
TESTE_PROGRAMA(R2) -> R1, R2
R1-.
( SE R1
  ENTAO EQTO R2
        FACA ( R2-.
               R3+.)
  SENAO
        R1+.).
```

Fonte: adaptado de Silva (2012).

3.3 ESPECIFICAÇÃO DA LAN

A linguagem é *case sensitive* e possui identificadores para declaração de atributos e as operações de incremento, decremento, adição, subtração, multiplicação, divisão e atribuição. Possui também a comparação de valor menor ou igual sobre cada um destes. A LAN dispõe ainda de instrução do tipo teste formada pela utilização das palavras reservadas `if` e `else`.

No Quadro 22 são mostradas as definições regulares da LAN, que são utilizadas na definição dos símbolos terminais, onde:

a) `letra`: conjunto de letras minúscula e maiúsculas do alfabeto;

b) `digito`: um valor natural compreendido entre 0 e 9;

c) `constante_inteira`: número natural formado por algarismos compreendidos entre 0 e 9.

Quadro 22 - Definições regulares da linguagem alto nível

```
letra : [A-Za-z]

digito: [0-9]

constante_inteira: {digito}+
```

No Quadro 23 são mostrados os *tokens* (símbolos terminais) da linguagem, onde:

a) `identificador`: define a formação de um identificador, que inicia com a palavra `id`, seguido por um *underline*, seguido por letras ou dígitos;

b) `constante_inteira`: define a formação de um valor natural composto por um ou mais dígitos;

c) `palavra_reservada`: define a formação de uma palavra reservada, composta por uma ou mais letras;

d) `nome_programa`: define a formação do nome do programa, iniciado por uma letra, seguido por letras, *underline* ou dígitos;

e) `run, if, else, loop, break`: *tokens* que são palavras reservadas da LAN;

f) `(, ), [, ], ., ,, =, <, <=, +, -, *, /, ++, --, ->`: símbolos especiais da LAN;

g) `[\s \t \n]`: notação para que o compilador ignore os caracteres de tabulação.

Quadro 23 - Símbolos terminais da LAN

```
identificador: id_ ((( {letra} ) | {digito} ) _? ) +

constante_inteira: {digito}+

palavra_reservada: {letra}+

nome_programa : ({letra} ({letra} | _ | {digito})*)

run = palavra_reservada : "run"
if = palavra_reservada : "if"
else = palavra_reservada : "else"
loop = palavra_reservada : "loop"
break = palavra_reservada : "break"

"("
")"
"["
"]"
"."
","
"="
"<"
"<="
"+"
"-"
"*"
"/"
"++"
"--"
"->"
:[\s \t \n]
```

No Quadro 24 são apresentadas as produções da gramática definida para a LAN, utilizando a notação BNF, juntamente com as ações semânticas.

Quadro 24 - Gramática da LAN

```
<forma_geral> ::= run "(" nome_programa #24 <parametros> "->" <retornos> ")"
"[" <declaracao_variaveis> #23 "]" "[" <lista_comandos> "]" #13;

<declaracao_variaveis> ::= ĩ | <declaracao_var>;
<declaracao_var> ::= <lista_de_id> #1 ".";
<lista_de_id> ::= identificador #2 "=" constante_inteira #7 <listaId>;
<listaId> ::= ĩ | "," <lista_de_id>;

<retornos> ::= identificador #22 <ids_retornos>;
<ids_retornos> ::= ĩ | "," <retornos>;

<parametros> ::= ĩ | "[" <ids_parametros> "]";
<ids_parametros> ::= identificador #25 <ids_parametros_aux> ;
<ids_parametros_aux> ::=  ĩ | "," <ids_parametros> ;

<lista_comandos> ::= <comando> <listaAux>;
<listaAux> ::= ĩ | <comando> <listaAux>;

<comando> ::= identificador #2 <comandoAux> |  <comando_if> #6 | <comando_repeticao> | break "." #21 ;
<comandoAux> ::="++" "." #3|"--" "." #4 | "=" <expressao> #5 ".";

<comando_if> ::= if <expressao> #8 "[" <lista_comandos> "]" else #9 "[" <lista_comandos> "]";

<comando_repeticao> ::= #19 loop "[" <lista_comandos> "]" #20 ;

<expressao> ::=  <aritmetica> <relacional>;

<relacional> ::= <operador_relacional> #10 <aritmetica> #11 | ĩ;

<operador_relacional> ::= "<" | "<=";

<aritmetica> ::= <termo><aritmetica1>;
<aritmetica1> ::= ĩ | "+" <termo> #12 <aritmetica1> | "-" <termo> #14 <aritmetica1>;

<termo> ::= <elemento_expressao> <termo1>;
<termo1> ::= ĩ | "*" <elemento_expressao> #15 <termo1> | "/" <elemento_expressao> #16 <termo1>;

<elemento_expressao> ::= identificador #17 | constante_inteira #18 | "(" <expressao> ")" |
              "+" <elemento_expressao>  | "-" <elemento_expressao> ;
```

A seguir são descritas as funcionalidades implementadas para cada ação semântica mostrada no Quadro 24, responsáveis por traduzir o programa de LAN para LIR. O número das ações semânticas não reflete a ordem em que são executadas. São definidas vinte e cinco ações, as quais são:

a) ação 1: responsável pelo controle da declaração de identificadores. Também checa duplicidade para evitar que o mesmo atributo seja declarado mais de uma vez. É nesta ação que também ocorre a associação do valor natural declarado no desenvolvimento do programa em alto nível. Para cada atributo declarado, é gerada uma instrução em LIR de atribuição de registrador. O controle do número natural que cada registrador irá receber é feito incrementalmente a cada vez que esta ação é executada;

b) ação 2: adiciona o nome do identificador declarado em uma lista de controle. O controle refere-se a manipulação do identificador. Ele poderá ser incrementado, decrementado ou receber o valor de uma nova expressão por atribuição;

c) ação 3: gera código para incremento de um registrador. É obtido o primeiro elemento da lista de controle manipulada na ação 2 e gerado o código em `String`

que é suportado pela LIR. A instrução é então adicionada a uma lista de código objeto LIR;

d) ação 4: mesma lógica de implementação da ação 3, porém, gerando instrução de decremento de registrador, adicionando-a na lista de código objeto;
e) ação 5: responsável por gerar o comando de atribuição do registrador identificado na ação 2. A atribuição é feita com a criação de registradores temporários nas ações semânticas subsequentes. Quando a ação 5 é executada, ela apenas desempilha o registrador temporário que contém o valor natural manipulado e é gerado o código de atribuição de registrador, sendo adicionado na lista de código objeto LIR;
f) ação 6: corresponde a saída do comando de teste. Responsável por atualizar o valor do desvio `va_para` no último comando do bloco `entao`, evitando que haja desvios para dentro do bloco `senao`;
g) ação 7: armazena os valores naturais declarados para os atributos de alto nível em uma lista. Estes valores são obtidos na ação 1 para associação do atributo com seu valor;
h) ação 8: este método empilha o endereço da instrução de teste para posterior atualização do bloco `senao / va_para`. É gerada a instrução de teste conforme a LIR e o valor do `va_para` é marcado simplesmente com a `String XY`. Esta `String` indica que ela será substituída posteriormente quando for identificado o bloco de `senao` na ação 9. Com a ação 9 é possível identificar qual é o valor de desvio correto para associar com o `va_para`, já que corresponde ao início do bloco `senao`;
i) ação 9: atualiza o comando de teste com o valor do `senao / va_para`. O valor marcado com `XY` na ação 8 pode ser agora atualizado com o endereço da instrução atual. Também é obtido o último comando de dentro do bloco `entao` e marcado o valor do `va_para` com `XZ`. Esta `String` representa o valor que será atualizado e substituído ao final do teste com o endereço da instrução atual, executado pela ação 6;
j) ação 10: armazena a `String` que representa o operador relacional;
k) ação 11: conforme o valor do operador relacional armazenado na ação 10, são geradas as instruções de igualdade ou comparação de valores. Estes valores são

desempilhados e é gerado o código compatível com a LIR. O empilhamento destes valores que são utilizados dá-se nas ações 17 e 18;

l) ação 12: método que cria a adição de registradores. São desempilhados os valores que serão utilizados, semelhante a ação 11;
m) ação 13: finaliza a tradução do programa, adicionando na lista de código objeto LIR a instrução `retorna`;
n) ação 14: método que cria a subtração de registradores. São desempilhados os valores que serão utilizados, como ocorre na ação 11;
o) ação 15: ação de multiplicação de registradores. O código objeto em LIR é gerado a partir do desempilhamento dos valores que ocorre nas ações 17 e 18;
p) ação 16: ação de divisão com representação da parte inteira de dois registradores. O código objeto em LIR é gerado a partir do desempilhamento dos valores que ocorre nas ações 17 e 18;
q) ação 17: método que verifica se o identificador utilizado na expressão foi declarado. Se estiver declarado, adiciona-o a uma pilha de operandos para posterior utilização com as instruções que criam código objeto LIR;
r) ação 18: método que cria um operando com o valor natural declarado no programa para adicionar à pilha de operandos. Este operando será utilizado pelas ações que criam o código objeto LIR, como nas ações 11, 12, 14, 15 e 16;
s) ação 19: método que armazena em uma lista o endereço da instrução atual. Ela indica a posição em que é iniciado o comando de repetição (`loop`);
t) ação 20: método responsável por gerar o comando em LIR que indica o desvio para o início do comando `loop` que é identificado na ação 19. Esta ação também é responsável por atualizar o comando LIR que é gerado na ação 21 para o rótulo que corresponde à saída do comando de repetição;
u) ação 21: método que gera o comando em LIR indicando a saída do comando de repetição. O valor do `va_para` que corresponderá à saída do comando `loop` é marcado no código objeto LIR como uma `String` com o valor `XL`. Esta `String` é então substituída na ação 20 com o valor do endereço da instrução atual, já que corresponde a finalização do comando de repetição;
v) ação 22: método que armazena em uma lista de controle os atributos declarados como retorno do programa. Estes atributos serão utilizados posteriormente na ação 23 para verificar a integridade da declaração de variáveis utilizadas como retorno;

w) ação 23: método responsável por analisar se os atributos declarados como retorno do programa estão presentes na declaração de variáveis;

x) ação 24: método responsável por armazenar o valor declarado que corresponde ao nome do programa;

y) ação 25: método responsável por armazenar em uma lista de controle os atributos utilizados como parâmetros no programa. Estes atributos terão um valor sequencial gerado de forma semelhante ao que ocorre na ação de número 1. Este número é a representação numérica do registrador na forma monolítica.

No Quadro 25 é apresentado um exemplo de programa escrito em LAN.

Quadro 25 - Programa escrito em LAN

```
run ( teste_Programa [id_xpto] -> id_xpto )
[id_xpto = 2.]
[id_xpto++.

loop [if (id_xpto < 15) [
        id_xpto++.
        if (id_xpto< 15) [ id_xpto++. ] else [ id_xpto--. ]
     ] else [
        break.
     ]
     id_xpto++.
    ]
]
```

No Quadro 26 é apresentado o programa LIR convertido que corresponde ao programa em LAN do Quadro 25.

Quadro 26 - Programa em LIR convertido a partir de LAN

```
programa teste_Programa(R1) -> R1
1: faca R1 = 2 va_para 2
2: faca inc(R1) va_para 3
3: faca R2 = 15 va_para 4
4: faca R3 = A_Menor_B(R1, R2) va_para 5
5: faca RT = R3 va_para 6
6: se T entao va_para 7 senao va_para 14
7: faca inc(R1) va_para 8
8: faca R4 = 15 va_para 9
9: faca R5 = A_Menor_B(R1, R4 ) va_para 10
10: faca RT = R5 va_para 11
11: se T entao va_para 12 senao va_para 13
12: faca inc(R1) va_para 15
13: faca dec(R1) va_para 15
14: faca RT = 0 va_para 17
15: faca inc(R1) va_para 16
16: faca RT = 0 va_para 3
17: retorna
```

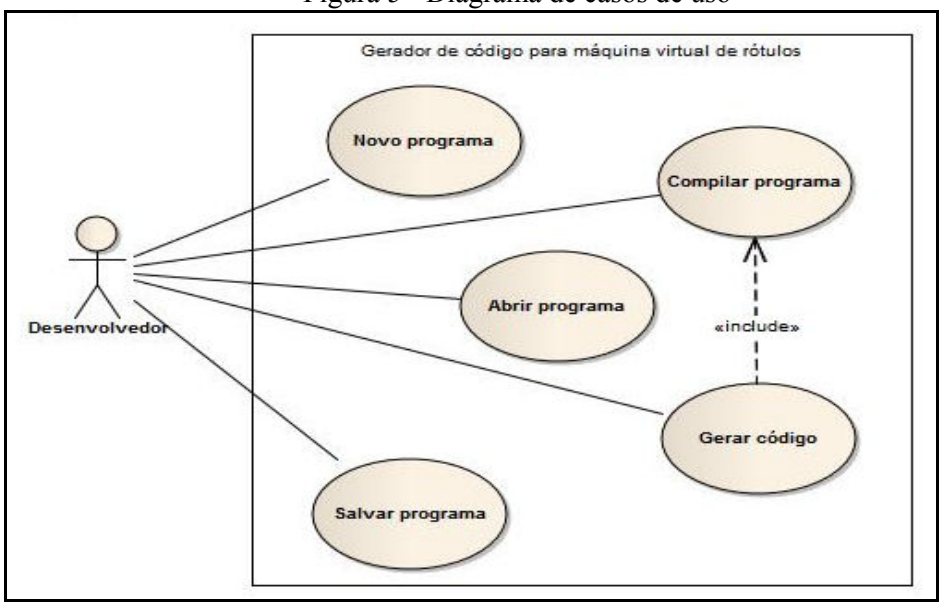
3.4 ESPECIFICAÇÃO DO AMBIENTE

A especificação deste trabalho foi desenvolvida utilizando a ferramenta Enterprise Architect, utilizando os conceitos do paradigma de orientação a objetos e por meio de diagramas da *Unified Modeling Language* (UML). Assim, nas próximas seções são apresentados os diagramas de casos de uso, classes e sequência da especificação do ambiente.

3.4.1 Casos de Uso (*Use Case* – UC)

Nesta subseção são descritos os casos de uso das funcionalidades principais do gerador de código. Foi identificado um ator que utilizará o ambiente, denominado `Desenvolvedor`. Este ator irá programar em LAN ou LIT, a sua escolha, para gerar código na linguagem objeto LIR. A Figura 5 apresenta os casos de uso da ferramenta.

Figura 5 - Diagrama de casos de uso



O caso de uso apresentado no Quadro 27 descreve a ação do `Desenvolvedor` que cria um novo programa.

Quadro 27 - Caso de uso UC01

UC01 – Novo programa	
Descrição	O desenvolvedor cria um novo programa selecionando o botão correspondente no ambiente gerador de código.
Cenário Principal	1. O desenvolvedor seleciona o botão Novo no ambiente já em execução. 2. O ambiente verifica se o programa corrente contém modificações que não estão salvas em arquivo. 3. O desenvolvedor confirma a ação para ser criado um novo programa.
Fluxo alternativo 1	4. No passo 3 o desenvolvedor pode não confirmar a ação e continuar utilizando o programa corrente. 5. No passo 3 caso o programa corrente contenha modificações não salvas, será apresentada uma tela informando desta condição ao desenvolvedor.

No Quadro 28 é apresentada a ação de abertura de um programa salvo pelo ambiente gerador de código para a MVR pelo `Desenvolvedor`.

Quadro 28 - Caso de uso UC02

UC02 – Abrir programa	
Descrição	O desenvolvedor abre um programa salvo selecionando o botão correspondente no ambiente gerador de código.
Pré-condição	Deve existir ao menos um programa salvo.
Cenário Principal	1. O desenvolvedor seleciona o botão Abrir no ambiente. 2. O ambiente verifica se o programa corrente contém modificações ou não está salvo em arquivo. 3. O desenvolvedor confirma a ação de abertura e seleciona um programa existente. 4. O ambiente carrega este programa exibindo seu código no editor.
Fluxo alternativo 1	5. No passo 3 o desenvolvedor pode não confirmar a ação e continuar utilizando o programa corrente.

No Quadro 29 é apresentado o caso de uso que salva um programa corrente em arquivo.

Quadro 29 – Caso de uso UC03

UC03 - Salvar programa	
Descrição	O desenvolvedor salva o programa selecionando o botão correspondente no ambiente gerador de código.
Cenário Principal	1. O desenvolvedor seleciona o botão Salvar no ambiente. 2. O ambiente apresenta janela para visualização de arquivos do sistema do desenvolvedor. 3. O desenvolvedor escolhe o destino e nomeia o arquivo a ser salvo. 4. O ambiente verifica se o arquivo já existe e salva o programa.
Fluxo alternativo 1	5. No passo 3 o desenvolvedor pode não confirmar a ação e continuar utilizando o programa corrente. 6. No passo 4 se o arquivo já existir, o desenvolvedor seleciona se deseja sobrescrevê-lo.

No Quadro 30 é apresentado o caso de uso que descreve uma das funções primordiais do ambiente gerador de código, a compilação do programa. O programa a ser compilado pode ser em LAN ou LIT.

Quadro 30 – Caso de uso UC04

UC04 - Compilar programa	
Descrição	O desenvolvedor compila o programa selecionando o botão correspondente no ambiente gerador de código.
Cenário Principal	1. O desenvolvedor seleciona uma linguagem fonte para escrever seu programa acionando o botão com o rótulo denominado de Linguagem. 2. O desenvolvedor codifica seu programa no editor de texto da ferramenta e aciona o botão Compilar. 3. O ambiente gerador de código compila o fonte efetuando análise léxica, sintática e semântica do código desenvolvido. 4. A ferramenta exibe em sua console o estado da compilação, se com sucesso ou descrevendo o erro encontrado, caso houver.
Fluxo alternativo 1	5. No passo 2 o desenvolvedor pode escolher por abrir um programa fonte já salvo em disco.

No Quadro 31 é apresentado o caso de uso de gerar código. O programa é compilado implicitamente ao acionar este comando.

Quadro 31 – Caso de uso UC05

UC05 – Gerar código	
Descrição	O desenvolvedor aciona a geração de código objeto LIR a partir de seu código desenvolvido selecionando o botão correspondente no ambiente.
Cenário Principal	1. O desenvolvedor seleciona uma linguagem fonte para escrever seu programa acionando o botão Linguagem no ambiente. 2. O desenvolvedor codifica seu programa no editor de texto da ferramenta e aciona o botão Gerar código. 3. O ambiente gerador de código compila o fonte efetuando análise léxica, sintática e semântica do código desenvolvido. 4. O ambiente gerador de código efetua a tradução do programa para LIR. 5. O ambiente exibe em sua console o estado da geração de código, se houve erro de compilação ou o programa convertido para o código objeto, caso esteja sem erros. 6. O ambiente salva o código objeto com o mesmo nome especificado pelo desenvolvedor no cabeçalho do programa. O arquivo é gerado no diretório raiz da ferramenta.
Fluxo alternativo 1	7. No passo 2 o desenvolvedor pode primeiramente compilar o programa fonte para checar por erros antes da geração de código.

3.4.2 Diagramas de classes

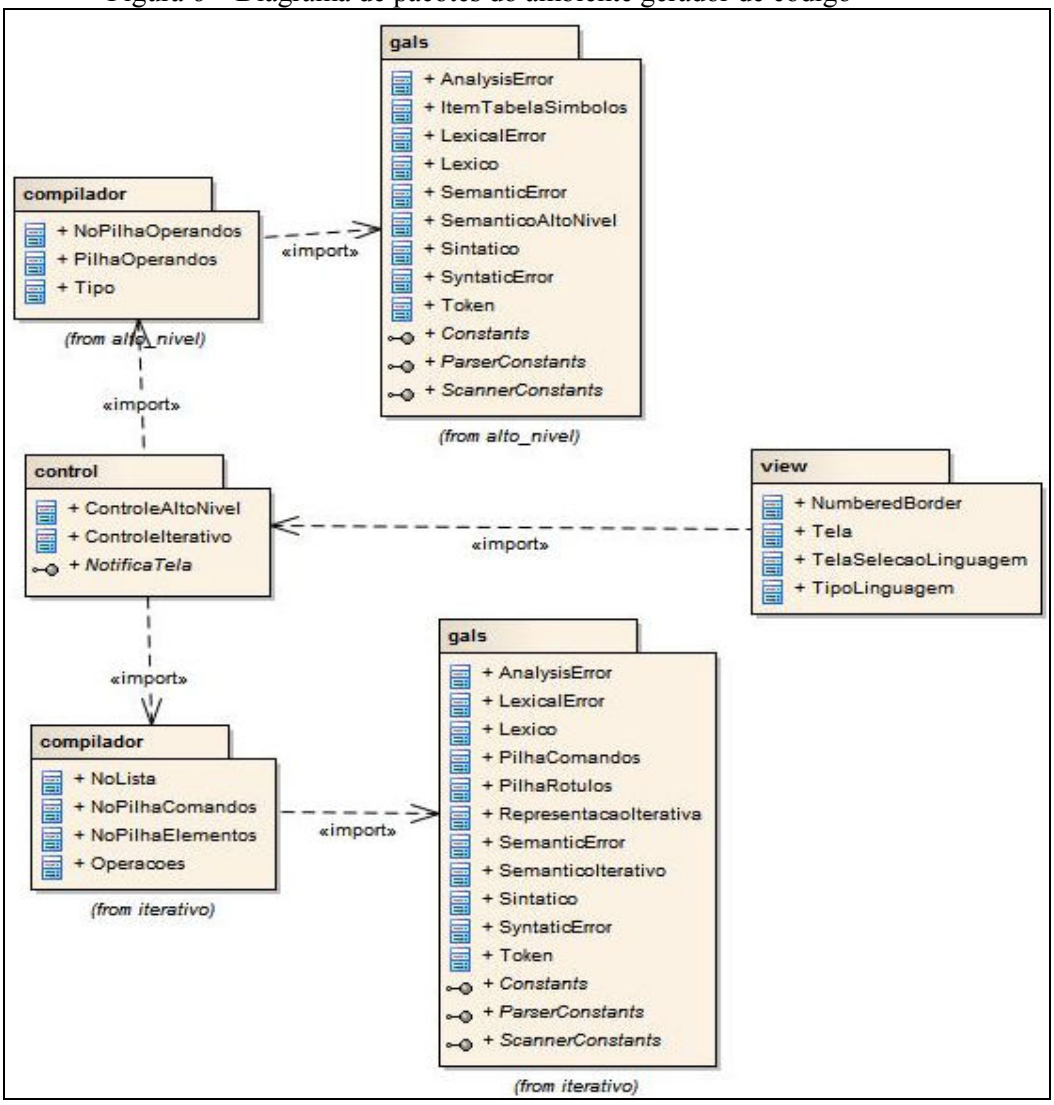
Nesta seção são apresentadas as classes desenvolvidas para a ferramenta, bem como seus relacionamentos e estrutura.

Organizadas estruturalmente seguindo o padrão *Model View Controller* (MVC), as classes de cada pacote são minimamente dependentes de outros pacotes. A partir do pacote `view` são instanciadas as classes de controle do pacote `control`, que se encarregarão da chamada dos métodos de análise léxica, sintática e semântica do programa fonte, bem como sua geração de código. As classes de controle, por sua vez, importam e acessam as classes dos pacotes fundamentais do ambiente. Os pacotes fundamentais do ambiente são: `alto_nivel.compilador`, `alto_nivel.gals`, `iterativo.compilador` e `iterativo.gals`, que podem ser visualizados na Figura 6. Estes pacotes contêm toda a especificação da LAN e LIT geradas a partir do GALS e suas ações semânticas que montam o código objeto na compilação dos programas.

Na Figura 6 são mostradas as dependências entre os pacotes definidos para o ambiente

juntamente com as classes que os compõem.

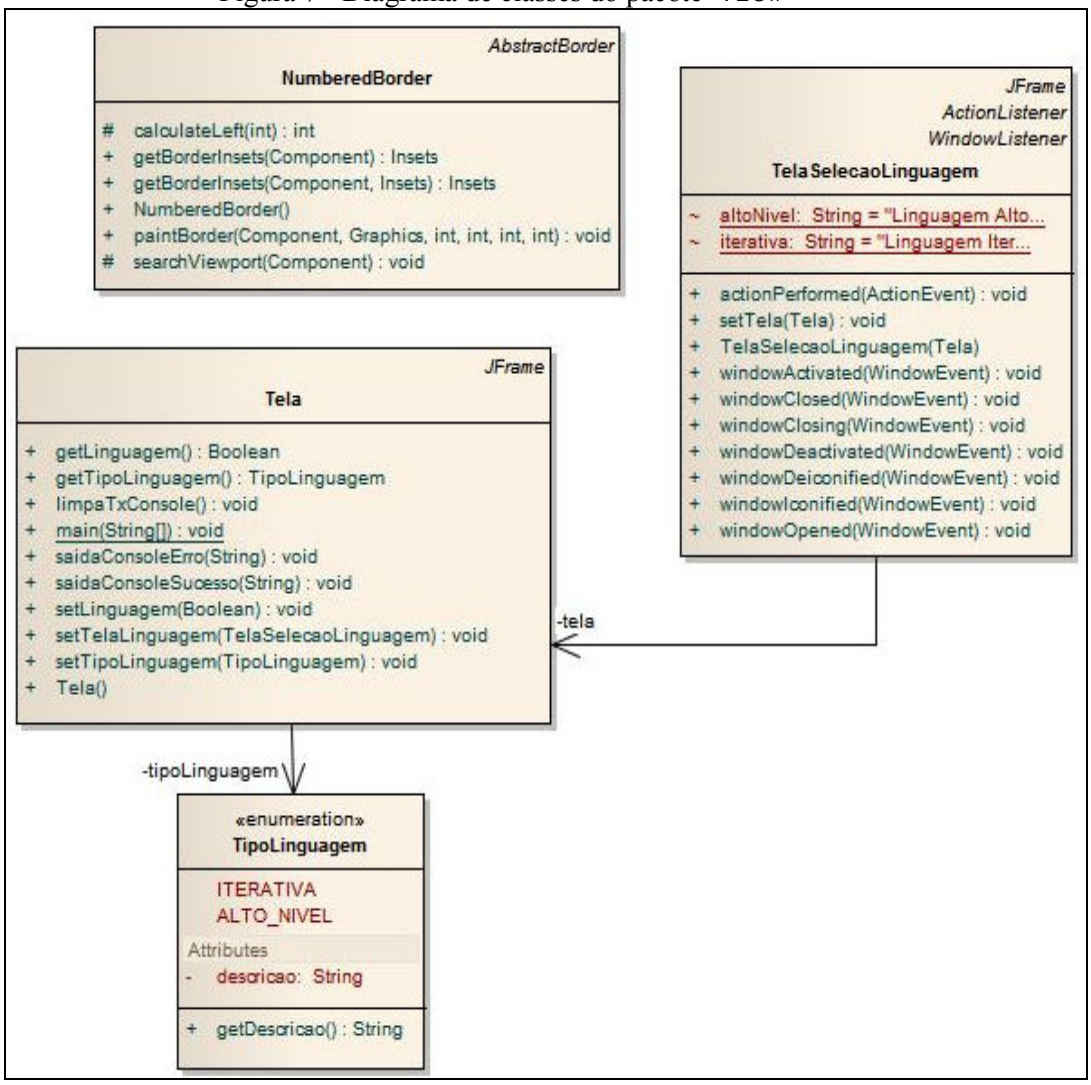
Figura 6 – Diagrama de pacotes do ambiente gerador de código



3.4.2.1 Pacote `view`

O pacote denominado `view` possui as classes responsáveis pela criação da tela disponibilizada ao usuário para interação com a ferramenta geradora de código. A Figura 7 representa o diagrama de classes deste pacote. Todos os atributos e métodos privados das classes foram removidos do diagrama para uma melhor visualização.

Figura 7 - Diagrama de classes do pacote `view`



A classe `Tela` é a principal do pacote `view`. A partir de sua instância, seu construtor encarrega-se de desenhar os componentes visuais e instanciar as classes `ControleAltoNivel` e `ControleIterativo` do pacote `control` (Figura 6 e Figura 8). Com as instâncias destas classes no escopo da `Tela`, é possível delegar *listeners* para os botões principais da janela do ambiente, o botão de compilar e de gerar código. Dependendo da seleção do tipo de linguagem fonte que o desenvolvedor escolher, uma das classes de controle invocará os métodos respectivos para compilação e geração de código objeto. É nesta classe (`Tela`) que existe o método `main` responsável pela inicialização do ambiente. A classe `Tela` implementa a interface `NotificaTela` (Figura 8) do pacote `control`. Esta interface contém métodos que funcionam como *listeners* para alteração de conteúdo visual. Se ocorrer alguma ação nas classes de controle que demanda alguma notificação ao usuário, como erros de compilação ou sucesso na geração de código, não é realizado um acesso direto à classe `Tela` (o que violaria os princípios do modelo MVC), mas sim notificados os objetos do tipo `NotificaTela` nas

classes de controle do pacote `control` para que efetuem a chamada do método desejado para atualização da tela. Com a invocação dos métodos `saidaConsoleSucesso`, `saidaConsoleErro` e `limpaTxConsole`, que são implementados pela `Tela`, é feito o ajuste requisitado para interação com o usuário.

O pacote possui ainda classes auxiliares como a `NumberedBorder`. Esta classe constrói a representação numérica em cada linha do editor de texto do ambiente. O método público `paintBorder` calcula o tamanho de cada linha e a posição atual no editor para representar o número de cada linha do mesmo alinhado à esquerda.

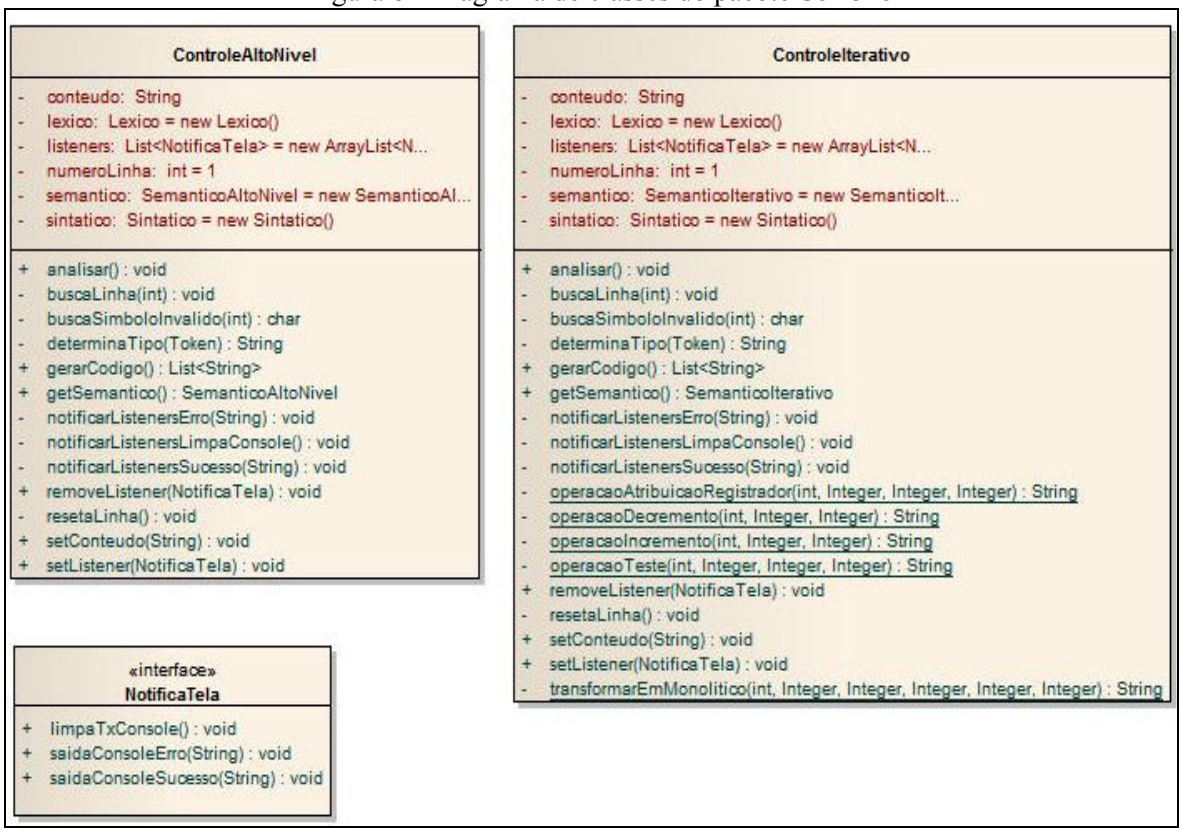
A enumeração `TipoLinguagem` diz respeito ao tipo das linguagens que o desenvolvedor pode escrever seu programa, sendo elas LAN e LIT.

A classe `TelaSelecaoLinguagem` constrói a tela que é exibida ao usuário quando o mesmo seleciona uma das linguagens desejadas para desenvolver seu código. Apresenta um grupo de botões para selecionar uma das linguagens e um botão de confirmação da ação.

3.4.2.2 Pacote `control`

As classes do pacote `control` são responsáveis pela implementação dos métodos que invocam a análise léxica, sintática e semântica da LAN e LIT, bem como a chamada das funções que geram o código objeto das mesmas. Todas as classes do presente pacote são independentes entre si. A classe de controle de código fonte da LAN importa apenas o pacote `alto_nivel`, assim como a classe de controle de código fonte da LIT importa apenas classes do pacote `iterativo`, conforme visualizado na Figura 6. O diagrama das classes contidas no pacote `control` é apresentado na Figura 8.

Figura 8 - Diagrama de classes do pacote **control**



A classe `ControleAltoNivel` é responsável por instanciar as classes do pacote `alto_nivel` que efetuam a análise léxica, sintática e semântica do código fonte. A partir do método público `analisar` é fornecido pela `Tela` o conteúdo do editor de texto no momento da ação do botão de compilar. Com esta informação são realizadas as análises de código. Para cada um dos tipos de erros possíveis (léxico, sintático e semântico) é realizado o tratamento de exceção com o recurso *try catch* disponível pela linguagem Java. A classe possui o método `gerarCodigo`, que obtém a partir de uma instância da classe `SemanticoAltoNivel` (Figura 12) no pacote `alto_nivel` a lista com o código objeto gerado.

Semelhante as funções da classe de alto nível, a classe `ControleIterativo` também é responsável pela análise de código, mas da linguagem iterativa fornecida. O método `gerarCodigo` desta classe monta o código objeto a partir da representação de valores inteiros em tabela criada pela classe `SemanticoIterativo` (Figura 10), consultando cada linha e coluna com as informações dos números dos registradores que foram definidos. Com estes valores de cada coluna, é possível identificar qual é a instrução correta que deve ser gerada para o código objeto. A geração deste código a partir da identificação do significado dos valores de cada coluna é feita pelo método privado auxiliar `transformarEmMonolitico` da classe `ControleIterativo`. Este método efetua a comparação dos valores e delega as chamadas dos métodos `operacaoTeste`, `operacaoAtribuicaoRegistrador`,

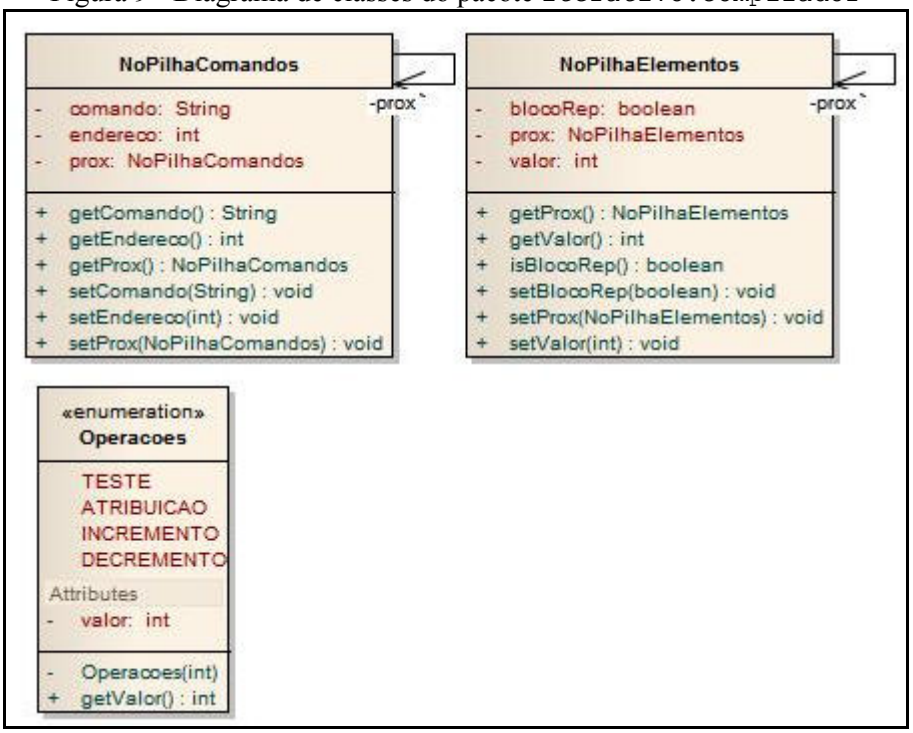
`operacaoIncremento` e `operacaoDecremento`. Estes métodos constróem o código objeto LIR.

A interface `NotificaTela` atua como uma intermediária entre as duas classes de controle e a classe `Tela`. Cada classe de controle contém uma lista de objetos desta interface que, ao ter um de seus métodos invocados, será implementada e executada pela classe `Tela`.

3.4.2.3 Pacote `iterativo.compilador`

Neste pacote são apresentadas as estruturas de dados auxiliares para as ações do compilador da linguagem iterativa. A Figura 9 representa o diagrama de classes deste pacote.

Figura 9 - Diagrama de classes do pacote `iterativo.compilador`



A enumeração `Operacoes` contém os tipos de operações disponíveis pela linguagem iterativa e um valor inteiro atribuído para construir a representação da tabela iterativa.

A classe `NoPilhaElementos` é instanciada por uma pilha de rótulos que armazena valores inteiros. Estes valores representam os endereços das instruções na execução iterativa. Cada um destes endereços pode indicar uma instrução sucessora ou um endereço pendente a ser atualizado (como blocos de repetição ou finalização de teste).

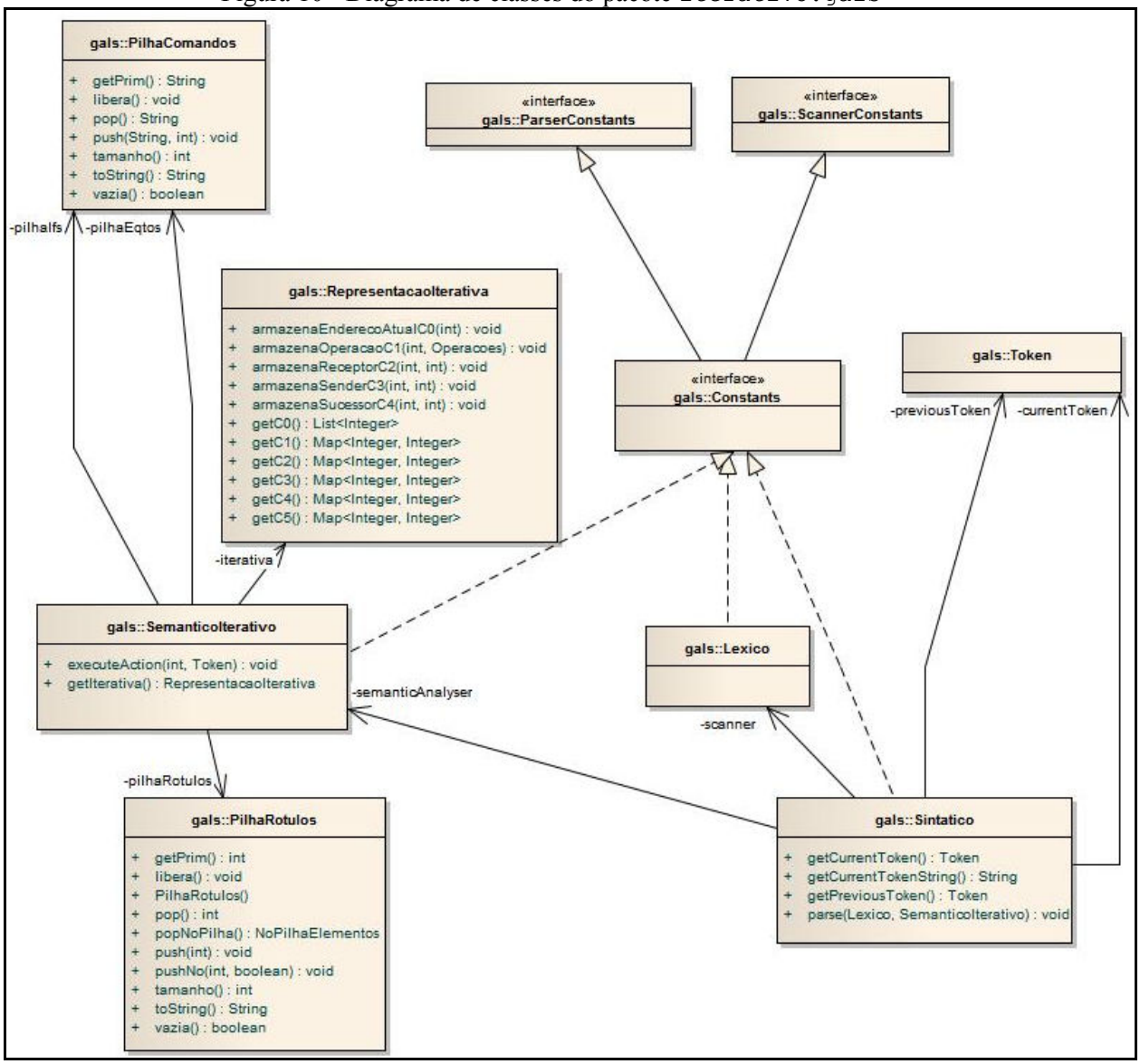
A classe `NoPilhaComandos` é utilizada na criação de uma uma pilha com os comandos que foram escritos. Auxilia o controle de desvios em casos de utilização de comandos `eqto`

dentro de comandos `if`, por exemplo, identificando qual o endereço correto para as próximas instruções definirem seus sucessores.

3.4.2.4 Pacote `iterativo.gals`

Neste pacote encontram-se as duas principais classes para o funcionamento da compilação e transformação de código iterativo, `SemanticoIterativo` e `RepresentacaoIterativa`. Este pacote possui ainda as classes geradas automaticamente pelo GALS com a implementação da especificação da linguagem iterativa.

Figura 10 - Diagrama de classes do pacote `iterativo.gals`



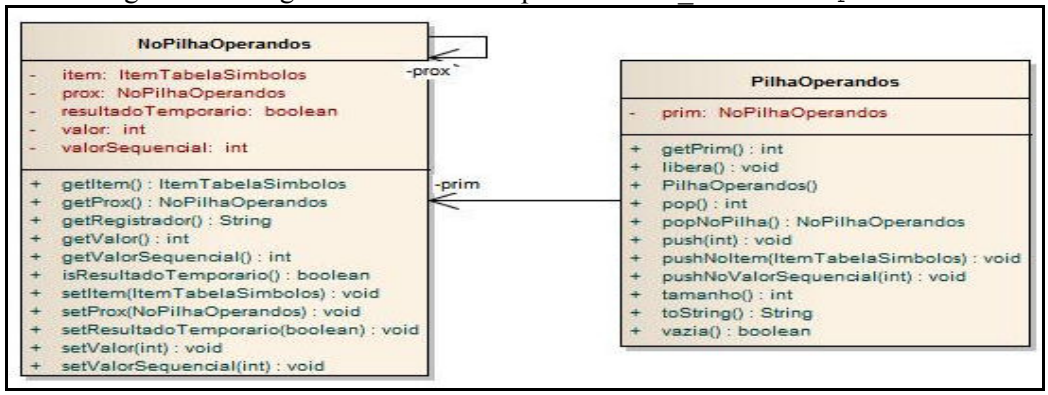
Todos os métodos e atributos privados foram suprimidos para melhor visualização. As interfaces `Constants`, `ScannerConstants` e `ParserConstants` foram geradas pelo GALS.

As classes `Sintatico`, `Lexico` e `Token` também foram geradas a partir do GALS. A classe `Sintatico` é responsável pela análise sintática do programa fonte, enquanto que a classe `Lexico` é responsável pela análise léxica. A classe `Token` identifica os segmentos de texto que identificam os símbolos na linguagem iterativa. A classe `SemanticoIterativo` teve sua estrutura gerada pelo GALS, embora suas ações semânticas tenham sido totalmente implementadas para tradução do programa iterativo em monolítico. O método `executeAction` espera como parâmetros um valor inteiro (identifica o número da ação semântica definida no GALS) e um *token* (*token* reconhecido no momento da ação semântica). Para cada ação é utilizado o comando `switch` para separar as ações semânticas e efetuar sua implementação. Nesta classe são utilizadas ainda as estruturas auxiliares `PilhaComandos` e `PilhaRotulos`, utilizadas para armazenar informações como os endereços das instruções que estão sendo traduzidas e os comandos as quais pertencem. A classe `RepresentacaoIterativa` é a especificação da definição apresentada no Quadro 12. Sua estrutura baseia-se em uma lista de valores inteiros para armazenar os endereços que foram criados para armazenar as informações da tabela iterativa e *hashmaps* com uma chave de valor inteiro que identifica o endereço da coluna e seu valor descritivo também inteiro, que identifica qual é o endereço que será utilizado na construção das operações ao traduzir esta representação para o código monolítico. Cada coluna representará um endereço sucessor ou o endereço correspondente ao comando `senao`.

3.4.2.5 Pacote `alto_nivel.compilador`

Neste pacote estão as estruturas de dados auxiliares para as ações do compilador da linguagem de alto nível. A Figura 11 apresenta o diagrama de classes deste pacote.

Figura 11 - Diagrama de classes do pacote `alto_nivel.compilador`

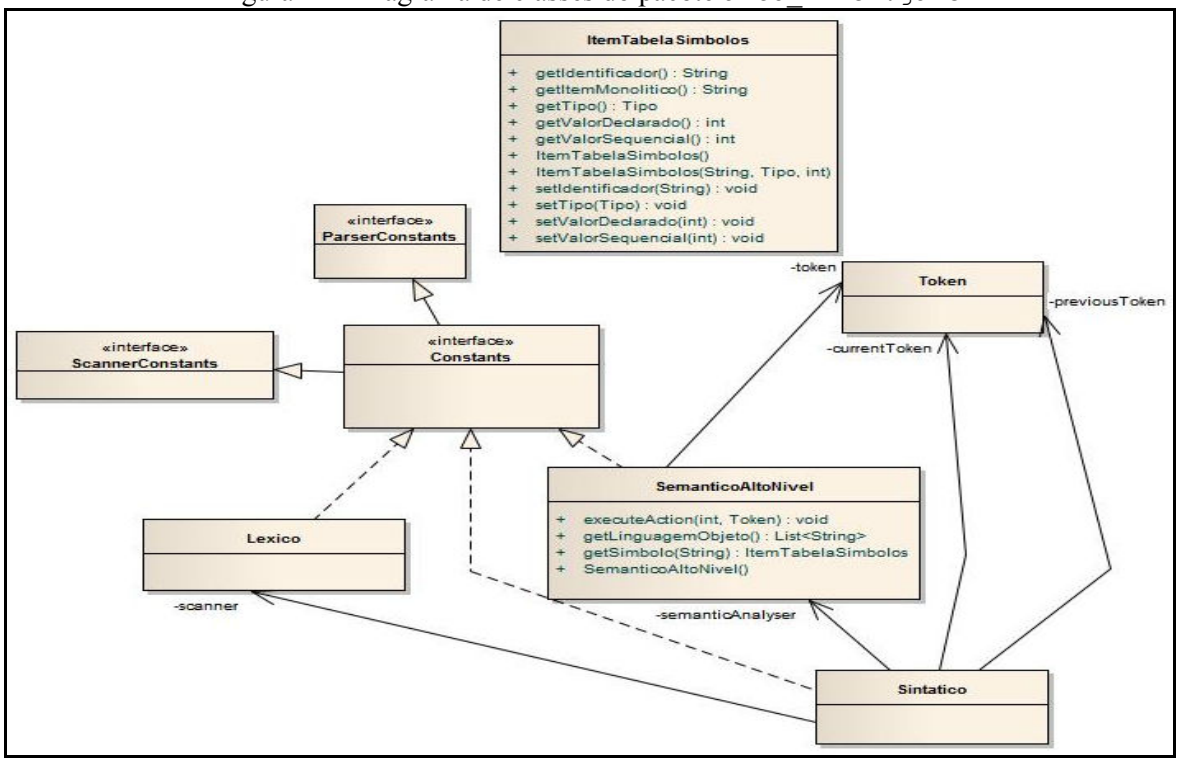


A classe `NoPilhaOperandos` é utilizada para armazenar os operandos da `PilhaOperandos`. Essa estrutura é necessária para controlar os identificadores e valores das constantes inteiras nas operações com o gerador de código.

3.4.2.6 Pacote `alto_nivel.gals`

As classes deste pacote são responsáveis pela análise léxica, sintática e semântica do código fonte de alto nível. A principal classe deste pacote é a `SemanticoAltoNivel` que executa as ações semânticas para traduzir o programa de alto nível para programa monolítico. O diagrama das classes deste pacote pode ser visualizado na Figura 12.

Figura 12 - Diagrama de classes do pacote `alto_nivel.gals`



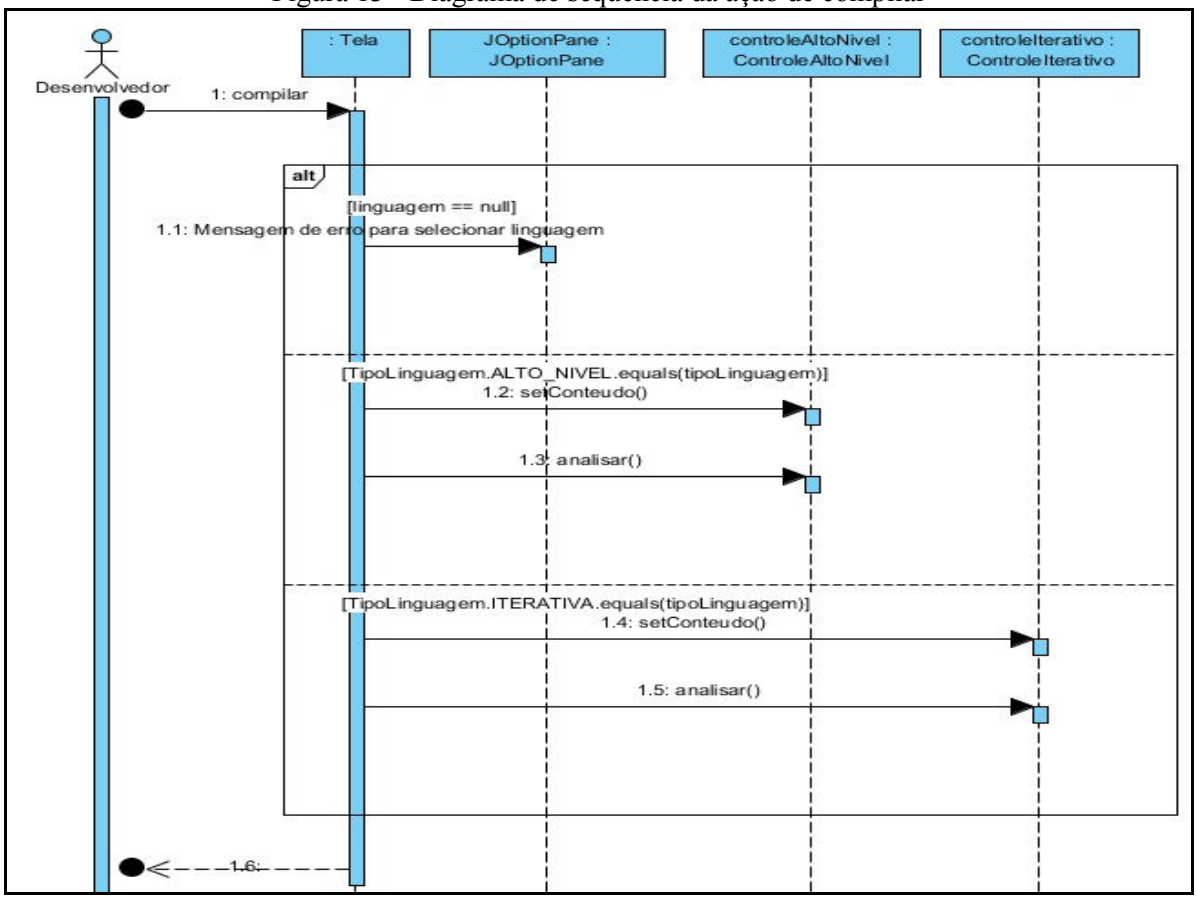
As classes deste pacote seguem a mesma lógica de implementação do pacote apresentado na Figura 10. As classes são geradas a partir da especificação definida no GALS. A diferença reside no fato de que apesar da classe `SemanticoAltoNivel` ter sua estrutura gerada pelo GALS, suas ações semânticas foram totalmente implementadas e modificadas para tradução do programa de alto nível em monolítico. Ela não utiliza a representação intermediária apresentada no Quadro 12 para traduzir o programa de LAN para LIR. A tradução para LIR ocorre diretamente em cada ação semântica especificada no Quadro 24 e o código resultante é agrupado em uma lista de `String` a medida que é convertido. Esta classe

utiliza ainda uma instância de `ItemTabelaSimbolos` para armazenar os identificadores declarados, seu valor declarado e o valor atribuído incrementalmente na compilação que determina o número de registrador monolítico associado ao identificador.

3.4.3 Diagramas de sequência

Esta seção apresenta diagramas de sequência que descrevem como acontece a interação do usuário da aplicação que faz uso do ambiente gerador de código para a MVR. A Figura 13 mostra o diagrama da ação de compilar.

Figura 13 - Diagrama de sequência da ação de compilar

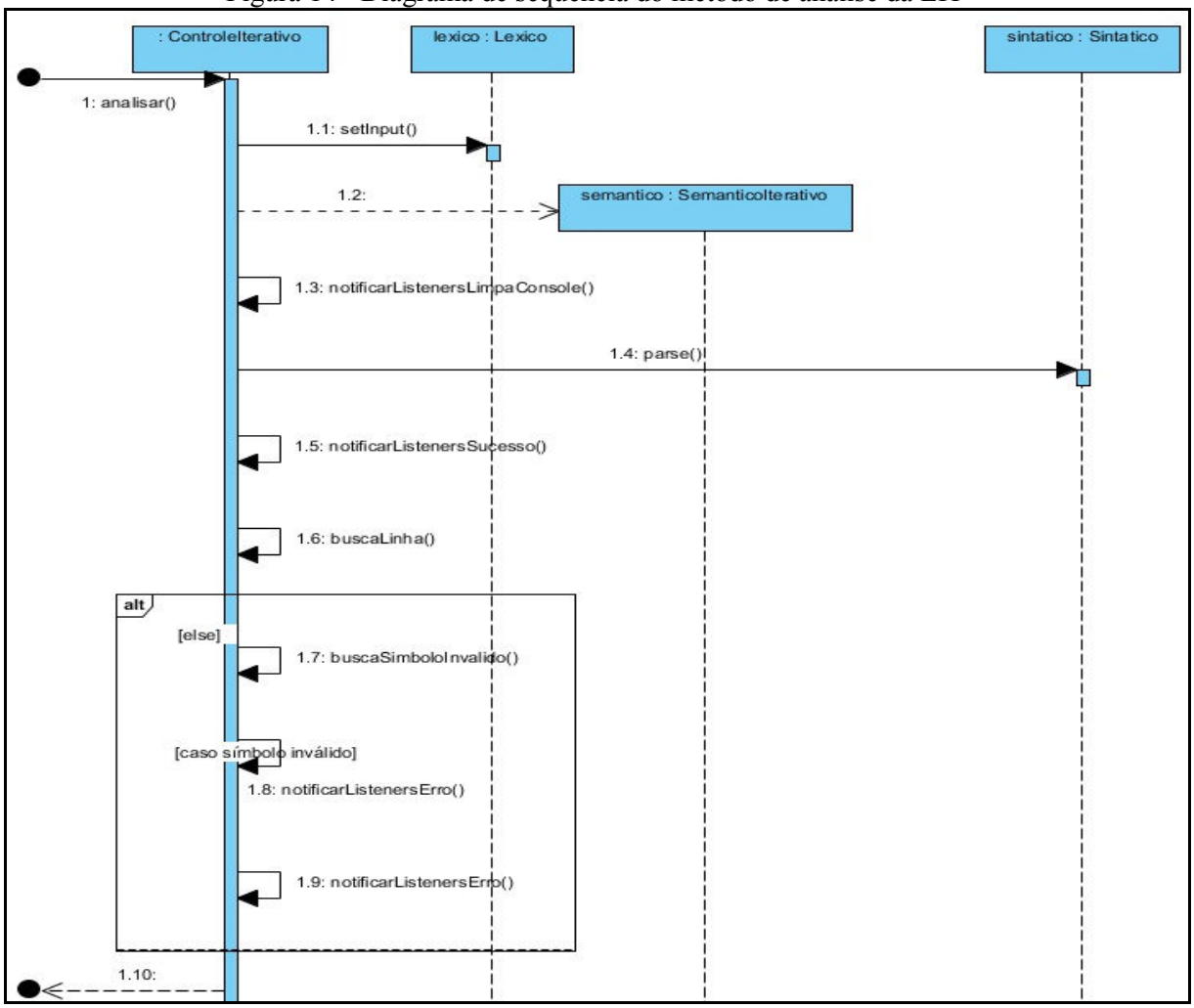


No diagrama apresentado, o desenvolvedor irá acionar o método para compilar um programa. Fazendo isso, o ambiente analisa o tipo da linguagem selecionada. Caso não haja, apresenta uma tela com a mensagem de erro. Caso seja LAN, cria-se uma instância da classe `ControleAltoNivel`, invocando o método `setConteudo` para informar o conteúdo do editor de texto que contém o programa fonte e, por fim, é acionado o método `analisar`, que efetuará as análises de código e consequente tradução para monolítico a partir da implementação das ações semânticas. A mesma lógica de implementação é seguida para programas na linguagem iterativa, com a diferença de se instanciar a classe

`ControleIterativo`.

Para uma melhor visualização, as demais ações oriundas da ação de compilar foram separadas em novos diagramas. O ator nestes diagramas é sempre o desenvolvedor, tendo em vista que a ação originária é a de compilar um programa, conforma apresentado na Figura 13. Na Figura 14 é possível visualizar a ação de analisar que é efetuada na classe `ControleIterativo`.

Figura 14 - Diagrama de sequência do método de análise da LIT



A ação de análise da classe `ControleIterativo` consiste em criar instâncias da classe `Lexico`, `Sintatico` e `SemanticoIterativo`, acionar o método `setInput` da classe `Lexico` e informar por argumento no construtor da classe `Sintatico` o objeto criado da classe `SemanticoIterativo`. A partir destas ações é realizada a chamada do método `parse` da classe `Sintatico`. Este método possui chamada para o método de ações semânticas da classe `SemanticoIterativo`, o qual efetuará a tradução do programa. A classe `ControleIterativo` possui uma lista de objetos da interface `NotificaTela` (Figura 8), os quais funcionam como *listeners* para alterar o estado da classe `Tela`. Desta forma, o padrão MVC não é violado. A

classe `ControleIterativo` não conhece a implementação dos métodos de notificação da tela, apenas executa suas chamadas a partir dos objetos `NotificaTela`. Portanto, a linha de execução dos métodos de notificação no diagrama não aponta para outra classe, mas sim para a própria classe que controla a instância de `NotificaTela`.

A Figura 15 apresenta o diagrama de sequência da ação de gerar código a partir da linguagem iterativa. É feito um laço de repetição dos números de endereço de instrução que foram criadas na tabela de representação iterativa e acessado cada um dos valores das colunas armazenadas em *hashmaps*. Cada um destes valores é interpretado pelo método `transformarEmMonolitico` da classe `ControleIterativo` (Figura 8), que define o significado de cada valor. A interpretação dos valores destas colunas armazenadas em *hashmaps* segue a mesma lógica de resolução apresentada a partir do Quadro 12. A Figura 16 mostra este método interpretador que gera o código final para a linguagem objeto LIR, adicionando o resultado em uma lista de *String* que representa cada linha do código objeto que foi gerado.

Figura 15 - Diagrama de sequência da ação de gerar código a partir de programa iterativo

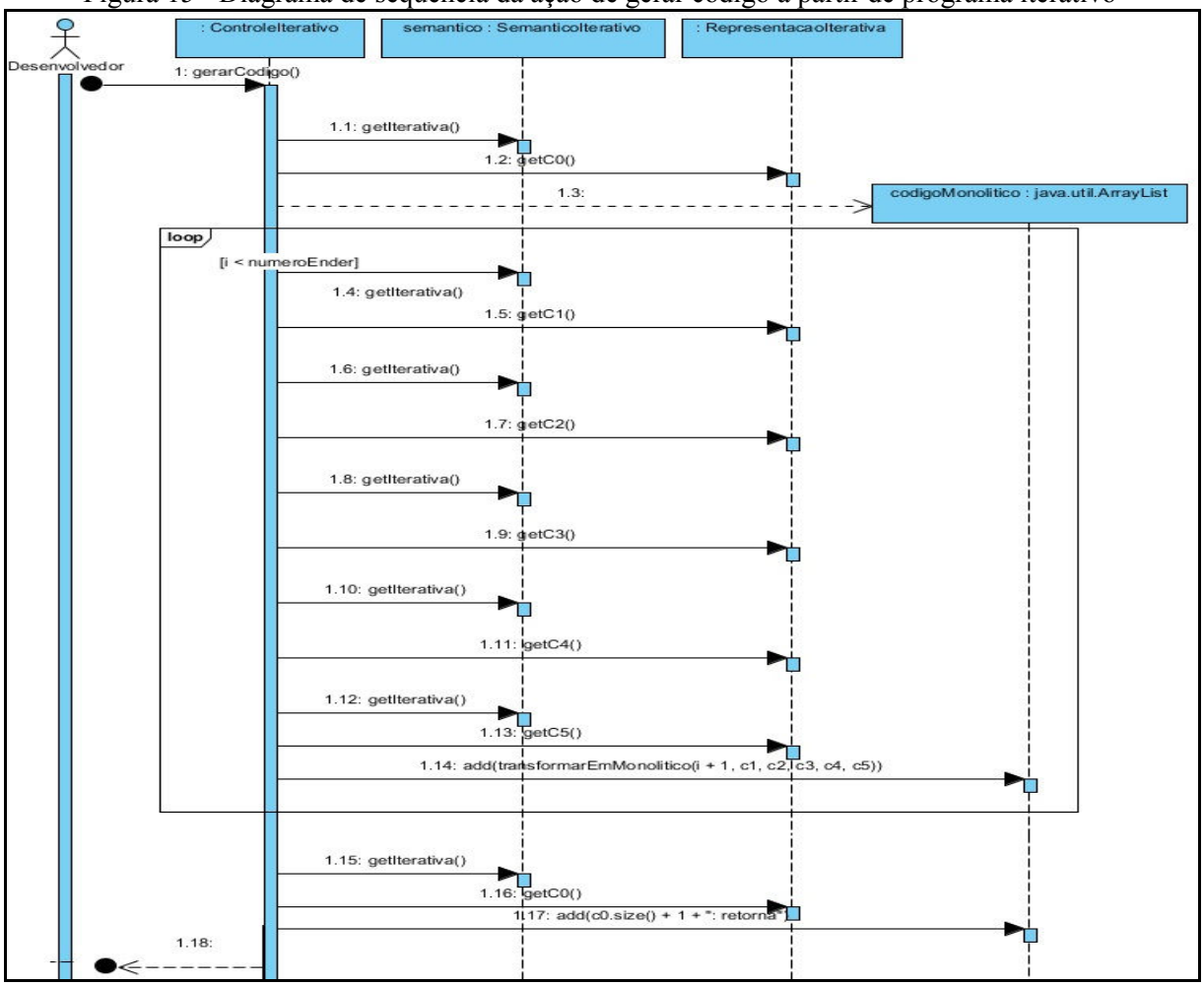
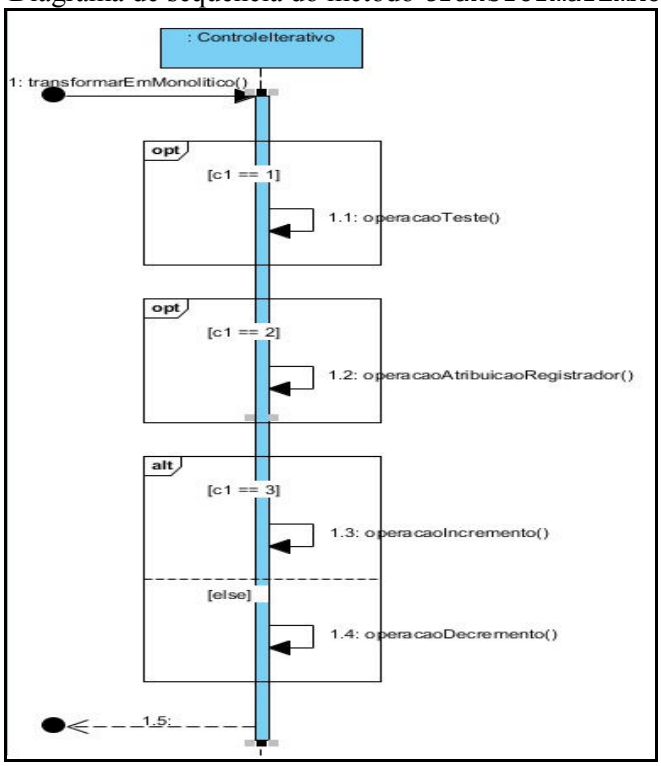


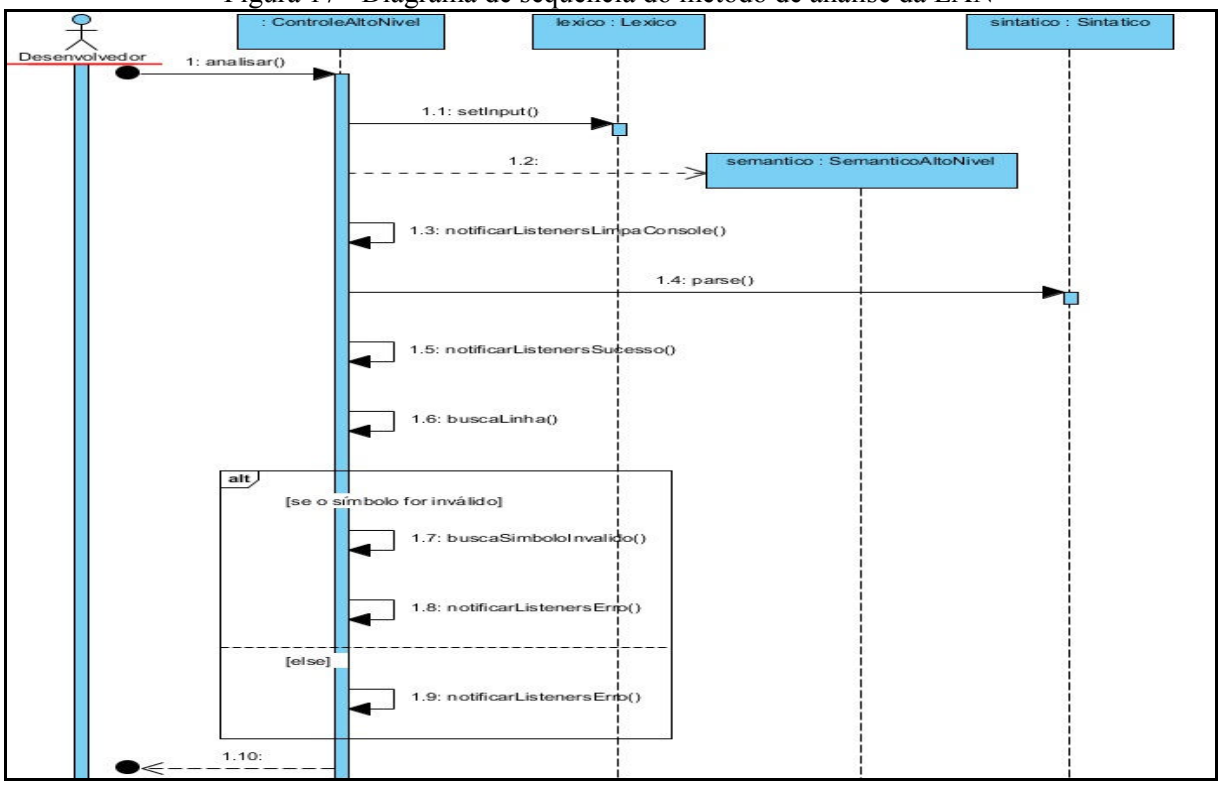
Figura 16 - Diagrama de sequência do método `transformarEmMonolitico`



Por ser invocado a partir do método de gerar código, o ator foi suprimido do diagrama da Figura 16 por estar implicitamente associado à ação que aciona o método de transformação.

Na Figura 17 é apresentado o método de análise da classe `ControleAltoNivel`.

Figura 17 - Diagrama de sequência do método de análise da LAN



O método segue a mesma lógica de implementação do método de análise da classe `ControleIterativo`. Sua única diferença é fazer uso das instâncias respectivas de `Lexico`, `SemanticoAltoNivel` e `Sintatico` do pacote de alto nível. O código de alto nível é transformado em monolítico a partir das ações semânticas implementadas pela classe `SemanticoAltoNivel`.

3.5 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação e a sua operacionalidade.

3.5.1 Técnicas e ferramentas utilizadas

O desenvolvimento do ambiente gerador de código para a MVR foi realizado utilizando a linguagem de programação Java. O ambiente utilizado para a construção do código da ferramenta foi o Eclipse Juno. Para maior interdependência entre os pacotes de classes, foi adotado o padrão MVC na implementação de código.

3.5.2 O ambiente gerador de código para a MVR

Após a geração das classes para análise léxica e sintática através do GALS, foi implementada a integração entre o código gerado em Java pela ferramenta com o código do ambiente gerador de código para a MVR. O GALS também gera uma estrutura de seleção com `switch` e `case` para controlar e implementar as ações semânticas. A classe `Tela` é responsável por criar as instâncias de controle de alto nível ou iterativa, com base na seleção do usuário. Quando é acionado o botão de compilar, a classe de controle respectiva a uma das linguagens efetuará as análises léxica, sintática e semântica, tratando seus erros caso existam. A ferramenta GALS gera a classe `Sintatico` que possui o método `parse`. Este método tem como argumentos uma instância da classe `Lexico` e uma instância da classe `Semantico`. A partir de sua execução é possível tratar os três tipos de exceção esperados no compilador: exceções léxicas, sintáticas e semânticas. Quando uma das exceções é tratada, é gerada uma mensagem formatada de acordo com o tipo de erro e notificada a interface `NotificaTela` que é responsável pela comunicação entre os pacotes `control` e `view`. Esta notificação da

interface fará com que seja executado o método `saidaConsoleErro` da classe `Tela` que exibirá a mensagem de erro ao desenvolvedor na console do ambiente. O método que executa estas análises pode ser visualizado no Quadro 32.

Quadro 32 - Método de análise de código

```
public void analisar() {

    lexico.setInput(conteudo);

    semantico = new SemanticoAltoNivel();

    notificarListenersLimpaConsole();

    try {

        sintatico.parse(lexico, semantico);

        notificarListenersSucesso("\t\tPrograma compilado com sucesso!");

    } catch (LexicalError e) {

        buscaLinha(e.getPosition());

        if (e.getMessage().equals("símbolo inválido")) {

            char invalid = buscaSimboloInvalido(e.getPosition());

            notificarListenersErro("Erro na linha "
            + numeroLinha + " - "
                + invalid + " "
            + e.getMessage());

        } else {

            notificarListenersErro("Erro na linha " + numeroLinha + " - " + e.getMessage());
        }

        resetaLinha();

    } catch (SyntaticError e) {

        buscaLinha(e.getPosition());

        String tipo = determinaTipo(sintatico.getCurrentToken());

        notificarListenersErro("Erro sintático na linha  "
            + numeroLinha + ", "
            + " encontrado "
            + tipo + " "
            + sintatico.getCurrentTokenString()
            + ", " + e.getMessage());

        resetaLinha();

    } catch (SemanticError e) {
        buscaLinha(e.getPosition());
        notificarListenersErro("Erro na linha " + numeroLinha + " - " + e.getMessage());
        resetaLinha();
    }
}
```

No caso da programação em linguagem iterativa, as ações semânticas da classe `SemanticoIterativo` são responsáveis por converter o programa fornecido em uma representação intermediária iterativa, como apresentado no Quadro 12. Com esta representação é possível analisar os valores da tabela construída e converter o código para LIR. Esta análise e conversão é mostrada no Quadro 33, onde é possível visualizar os métodos `gerarCodigo` e `transformarEmMonolitico`.

Quadro 33 - Métodos conversores de linguagem iterativa para monolítica

```
public List<String> gerarCodigo() {

    int numeroEnder = semantico.getIterativa().getC0().size();

    List<String> codigoMonolitico = new ArrayList<String>();

    for (int i = 0; i < numeroEnder; i++) {

        Integer c1 = semantico.getIterativa().getC1().get(i + 1);
        Integer c2 = semantico.getIterativa().getC2().get(i + 1);
        Integer c3 = semantico.getIterativa().getC3().get(i + 1);
        Integer c4 = semantico.getIterativa().getC4().get(i + 1);
        Integer c5 = semantico.getIterativa().getC5().get(i + 1);

        codigoMonolitico.add(transformarEmMonolitico(i + 1, c1, c2, c3, c4, c5));
    }

    List<Integer> c0 = semantico.getIterativa().getC0();
    codigoMonolitico.add(c0.size() + 1 + ": retorna");

    return codigoMonolitico;
}

private static String transformarEmMonolitico(int indice, Integer c1,
        Integer c2, Integer c3, Integer c4, Integer c5) {

    if (c1 == 1) {
        return operacaoTeste(indice, c1, c4, c5);

    }
    if (c1 == 2) {
        return operacaoAtribuicaoRegistrador(indice, c1, c3, c4);

    }
    if (c1 == 3) {
        return operacaoIncremento(indice, c2, c4);

    } else {
        return operacaoDecremento(indice, c2, c4);
    }
}
```

Na implementação da LAN, a conversão é realizada diretamente nas ações semânticas da classe `SemanticoAltoNivel`, sem a necessidade de criação de uma representação intermediária para posterior análise e conversão para monolítico.

Um exemplo de ação semântica pode ser visualizado no Quadro 34. Nele é mostrada a ação semântica número um, responsável por tratar a declaração de atributos e checar por duplicidade dos mesmos.

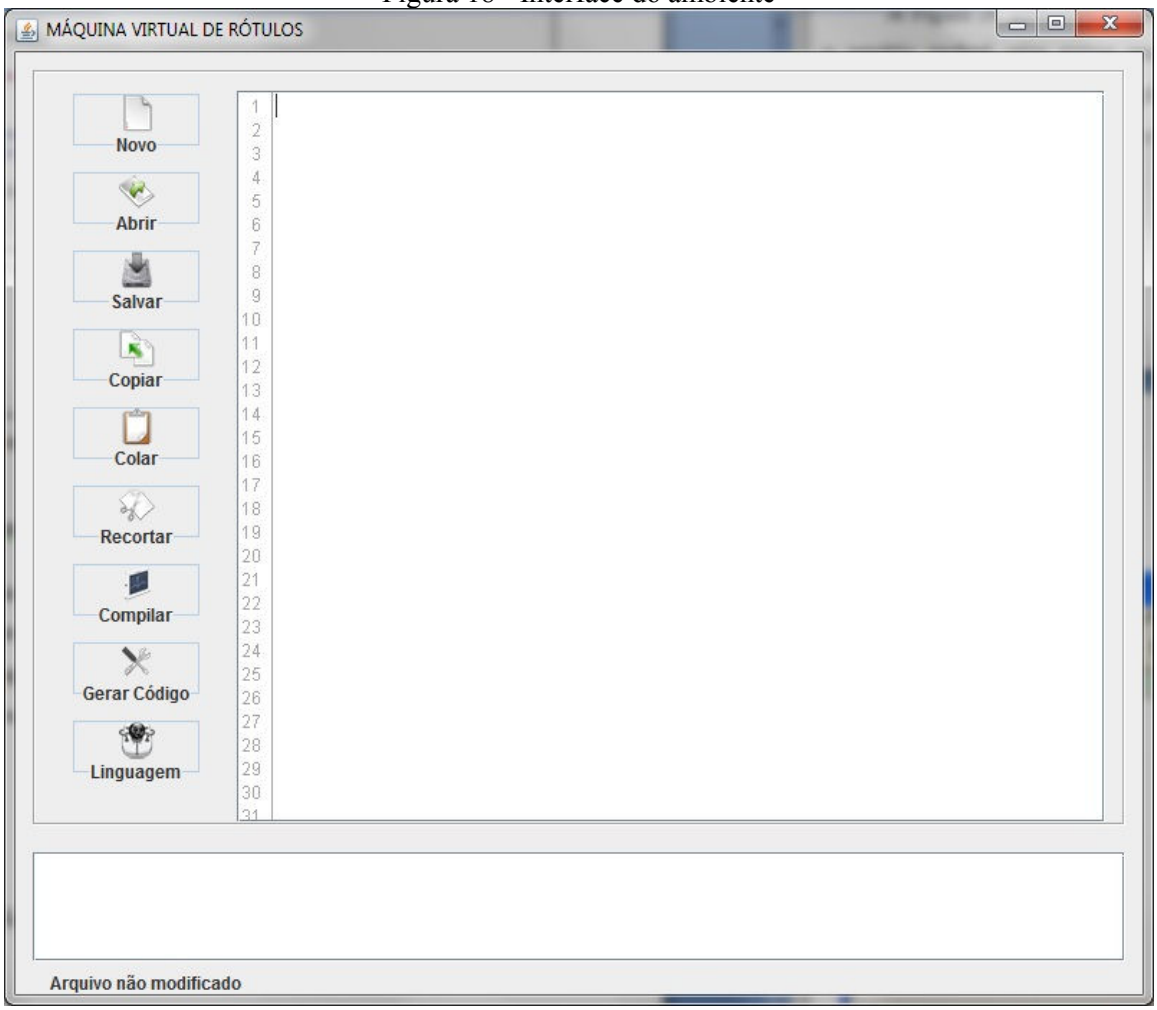
Quadro 34 - Ação semântica número 1 da linguagem alto nível

```
/**
 * Ação 01: declaração de atributos. Checa duplicidade e cria os objetos para tabela de símbolos.
 *
 * @throws SemanticError
 */
private void acao01() throws SemanticError {

    for (int i = 0; i < idsAltoNivel.size(); i++) {

        String id = idsAltoNivel.get(i);

        ItemTabelaSimbolos item = new ItemTabelaSimbolos();
        item.setTipo(Tipo.INT);
        item.setIdentificador(id);
        item.setValorSequencial(++numeroRegistrador);
        item.setValorDeclarado(valoresAltoNivel.get(i));

        criarInstrucaoDeclaracaoAtributo(item);

        if (getSimbolo(id) != null) {
            String msg = token.getLexeme() + " identificador declarado mais de uma vez";
            throw new SemanticError(msg, token.getPosition());
        }
        tabelaSimbolos.add(item);
    }

    idsAltoNivel.clear();
    posTipo = 0;
}
```

3.5.3 Operacionalidade da implementação

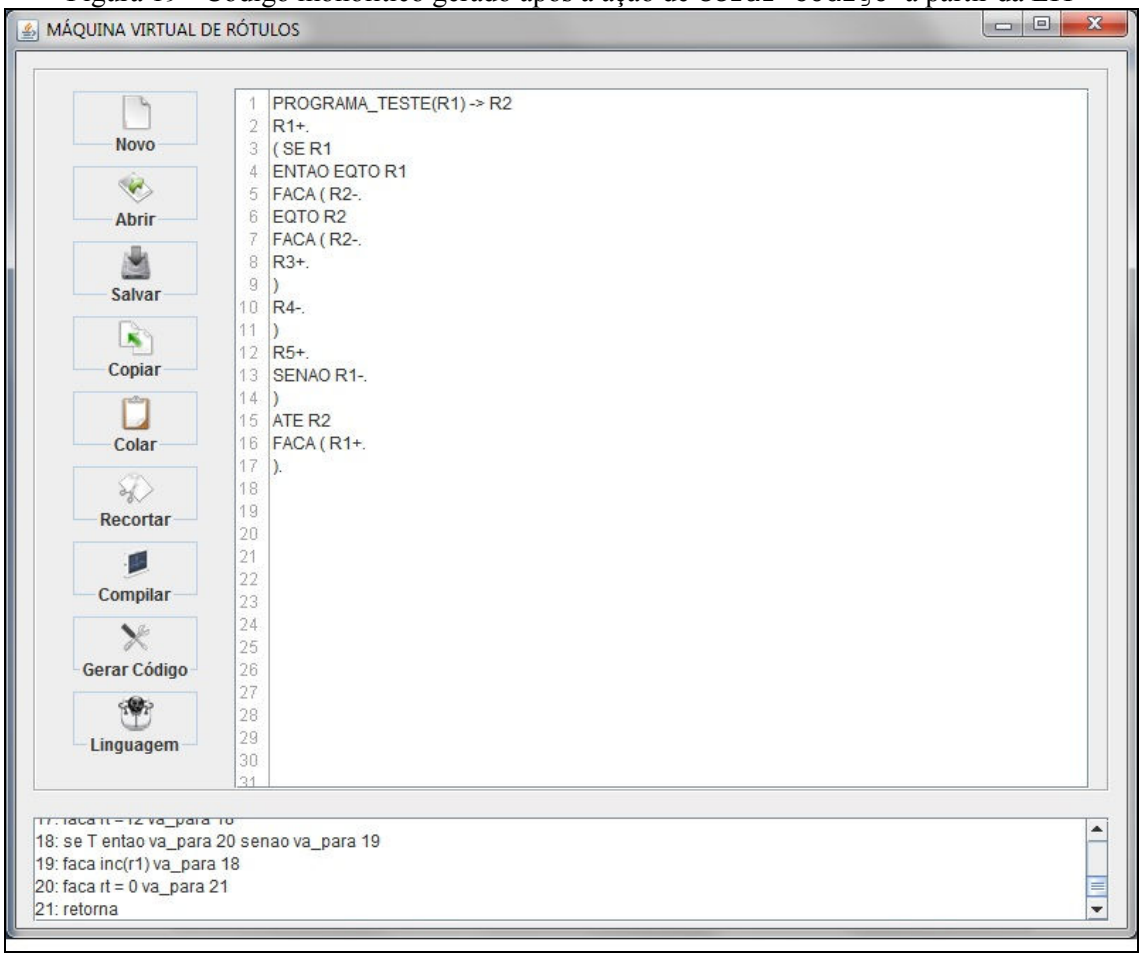
A Figura 18 apresenta a interface do ambiente com o usuário no papel de desenvolvedor. Através desta interface, o desenvolvedor poderá criar novos programas em linguagem iterativa ou alto nível, abrir programas existentes, salvar, compilar e gerar código para utilização na ferramenta de Silva (2004).

Figura 18 - Interface do ambiente



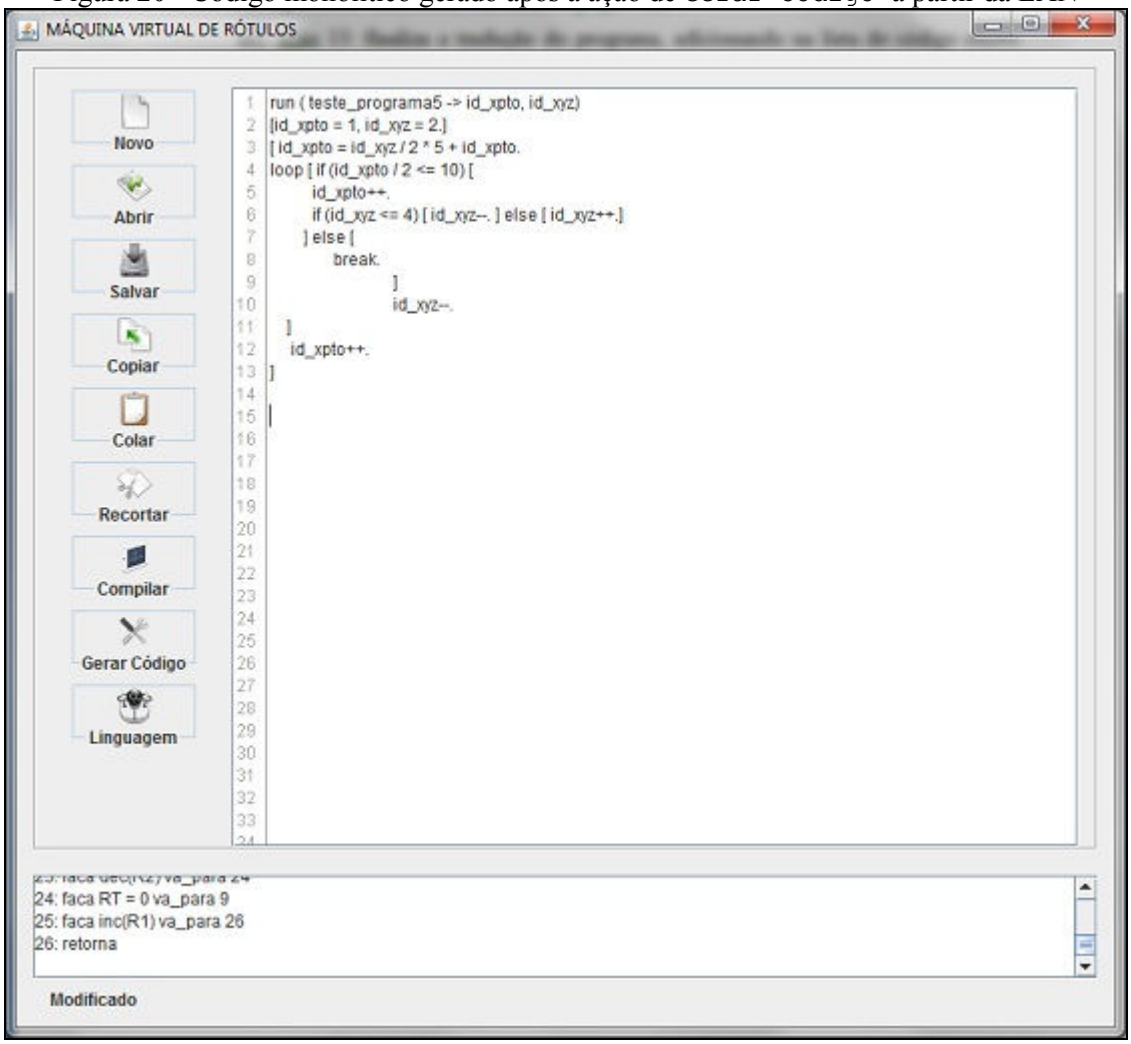
Na Figura 19 é possível visualizar um programa na forma iterativa sendo compilado e com sucesso e sendo convertido para a forma monolítica. O usuário seleciona a linguagem em que irá desenvolver selecionando o botão `Linguagem`, escreve seu código utilizando o editor de texto disponível e aciona o botão `Gerar Código` que fará as ações de análise e compilação do código. O resultado é demonstrado na console e também é gerado um arquivo de extensão `.mon`, compatível com o ambiente desenvolvido por Silva (2004). O arquivo gerado é criado com o nome correspondente ao que está declarado no cabeçalho do programa. O arquivo é salvo no diretório raiz do ambiente gerador de código.

Figura 19 - Código monolítico gerado após a ação de `Gerar Código` a partir da LIT



Na Figura 20 é apresentado um programa escrito em LAN, sendo compilado com sucesso e convertido na forma monolítica. A mesma lógica de implementação para geração de código é aplicada tanto para a geração a partir de linguagem iterativa quanto alto nível: é feita a análise e compilação e, caso não ocorram erros, é gerado o código em LIR. Como ocorre com a linguagem iterativa, na conversão da linguagem de alto nível também é gerado um arquivo de extensão `.mon` contendo a linguagem objeto gerada para utilização com o ambiente de Silva (2004).

Figura 20 - Código monolítico gerado após a ação de `Gerar Código` a partir da LAN



No Apêndice A é apresentado um programa escrito em LIT e LAN, bem como seu código objeto correspondente em LIR para melhor exemplificar o processo de geração de código.

3.6 RESULTADOS E DISCUSSÃO

Os resultados obtidos após o término do desenvolvimento do ambiente foram satisfatórios. Foi possível comprovar a funcionalidade da MVR e sua utilização com mais de uma linguagem fonte, sendo LAN ou LIT. Sua implementação facilitará o uso e demonstração didática de programas iterativos e monolíticos, bem como apresentar a conversão de uma linguagem de alto nível para baixo nível (monolítico).

De início pretendia-se utilizar apenas uma linguagem de alto nível e convertê-la para

uma forma iterativa usando a representação intermediária de tabela. Porém não foi possível atingir este objetivo, pois a LIT teria que suportar atribuição de registradores e também instrução de teste que não considerasse apenas um valor de registrador igual a zero, de modo que foram criadas então duas linguagens: LAN e LIT.

O ambiente possui algumas limitações, entre elas o fato de trabalhar apenas com números naturais. Outra limitação diz respeito ao número de operações disponíveis na linguagem de alto nível que dispõe apenas de um comando de repetição, o comando `loop`. Para simular a funcionalidade dos comandos `eqto` ou `ate` com a LAN é necessário utilizar o comando `loop` em conjunto com um comando `if`.

Foi possível gerar um código monolítico totalmente compatível com o ambiente de Silva (2004), formando uma solução de programação que agrega vários recursos.

4 CONCLUSÕES

Este trabalho apresenta o desenvolvimento de um ambiente capaz de converter duas linguagens, LAN e LIT, para LIR. Com isso foi atingido o principal objetivo.

A utilização da ferramenta GALS para a construção dos analisadores léxico e sintático foi de suma importância para o desenvolvimento, pois ela gerou todas as classes para estes analisadores, facilitando o desenvolvimento do compilador para a LAN e LIT.

Para a implementação deste ambiente foi necessário um estudo detalhado da ferramenta desenvolvida por Silva (2004) para correta utilização das macros criadas pela mesma e sua sintaxe. Também foram objetos de estudo os programas monolíticos e iterativos e suas propriedades. Com estes estudos foi criada ainda uma representação alternativa para os programas iterativos em forma de tabela, o que facilita a tradução para programas monolíticos.

Com duas linguagens disponíveis para escrita de programas e demonstração da sua conversão para LIR, o presente trabalho comprova na prática a utilização da MVR proposta em Silva (2003). Ficou clara a possibilidade de representar linguagens de alto nível e iterativas em uma representação de baixo nível, no caso, a LIR.

Este trabalho poderá ser utilizado na disciplina de Teoria da Computação da FURB. Ele demonstra na prática a conversão de programas e possibilita o ensino dos conceitos associados as linguagens fonte e o código objeto LIR. O ambiente apresenta limitações, como trabalhar apenas com números naturais e dispor de apenas um comando de repetição para a LAN. Entretanto, são disponibilizadas duas linguagens para programação com a ferramenta, o que pode facilitar a prática e aprendizado para conversão em programas monolíticos.

4.1 EXTENSÕES

Para ampliar os recursos do ambiente, sugere-se a implementação de mais operações para as linguagens iterativa e de alto nível, como a atribuição de valores na LIT e `while` na LAN, agregando maior disponibilidade de recursos. Há possibilidade também de se implementar a visualização da transformação passo a passo dos programas LIT e LAN em LIR, facilitando a compreensão. Poderá ser estudada também uma forma de converter a

linguagem de alto nível primeiramente para a representação iterativa na forma de tabela que foi especificada, para então utilizá-la na conversão para LIR. Visando um único ambiente de desenvolvimento para programas monolíticos, o ambiente de Silva (2004) poderia ser convertido para a linguagem Java e integrado ao presente trabalho. Com isso facilitaria a utilização e acesso aos recursos em uma única ferramenta.

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: Instituto de Informática da UFRGS, 2008; Bookman, 2008.

FERNANDES, Cláudia S. et al. **Programas recursivos e conversão de programas monolíticos.** Rolândia, [2004?]. Disponível em: <www2.unoeste.br/~chico/artigo_programas_recursivos.pdf>. Acesso em: 05 fev. 2013.

GESSER, Carlos E. **GALS**: gerador de analisadores léxicos e sintáticos. [Florianópolis], [2003?]. Disponível em: <http://gals.sourceforge.net>. Acesso em: 05 fev. 2013.

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

SILVA, José R. V. da. **Proposta de um novo algoritmo para transformação de um programa monolítico em um programa com instruções rotuladas compostas**. Artigo não publicado. Blumenau: Universidade Regional de Blumenau, 2003.

_____. **Definição intermediária de programa iterativo**. Artigo não publicado. Blumenau: Universidade Regional de Blumenau, 2012.

SILVA, Oliver M. da. **Ambiente para auxiliar o desenvolvimento de programas monolíticos.** 2004. 103 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

APÊNDICE A – Programa escrito em LAN e LIT e código objeto resultante

No Quadro 35 é apresentado um programa escrito em LIT e no Quadro 36 o seu código em LIR após a compilação.

Quadro 35 - Programa escrito em LIT

```
PROGRAMA_TESTE(R1, R2) -> R2
R1+.
( SE R1
ENTAO R2+.
SENAO R2-.).
```

Quadro 36 - Programa resultante em LIR após compilação e conversão do programa do Quadro 35

```
programa PROGRAMA_TESTE(R1,R2) -> R2
1: faca inc(r1) va_para 2
2: faca rt = r1 va_para 3
3: se T entao va_para 4 senao va_para 6
4: faca inc(r2) va_para 5
5: faca rt = 0 va_para 7
6: faca dec(r2) va_para 7
7: retorna
```

No Quadro 37 é apresentado um programa escrito em LAN, equivalente ao programa do Quadro 35 e no Quadro 38 o seu código em LIR após a compilação.

Quadro 37 - Programa escrito em LAN equivalente ao programa do Quadro 35

```
run ( TESTE_PROGRAMA [id_xpto, id_xys] -> id_xys ) [ ]
[ id_xpto++.
if (id_xpto < 1) [ id_xys++. ] else [ id_xys--. ]
]
```

Quadro 38 - Programa resultante em LIR após compilação e conversão do programa do Quadro 37

```
programa TESTE_PROGRAMA(R1,R2) -> R2
1: faca inc(R1) va_para 2
2: faca R3 = 1 va_para 3
3: faca R4 = A_Menor_B(R1, R3) va_para 4
4: faca RT = R4 va_para 5
5: se T entao va_para 6 senao va_para 7
6: faca inc(R2) va_para 8
7: faca dec(R2) va_para 8
8: retorna
```