UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Algoritmos de Otimização de Código Aplicado ao *Bytecode* Java

Autor
Michel Durante Zimmermann

Banca Examinadora
Prof. Olinto José Varela Furtado, Dr. (Orientador)
Prof. Leandro José Komosinski, Dr.
Prof. José Eduardo de Lucca, Dr.

Palavras-chave: **Otimização, *Bytecode*, JVM**

Florianópolis, 1 de Dezembro de 2004

Sumário

Lista de Figuras iv
Lista de Tabelas v
Lista de Siglas e Abreviaturas vi
Glossário vii
Resumo viii
Abstract ix
Introdução 1
1.1 Justificativas 1
1.2 Objetivos 2
1.3 Estrutura do trabalho 2
A Linguagem de Programação Java 3
2.1 História 3
2.2 Alguns conceitos Java 4
2.2.1 Bytecode 4
2.2.2 A Máquina Virtual Java (JVM) 4
2.2.2.1 Tipos 5
2.2.2.2 Áreas de Dados em Tempo de Execução 5
2.2.2.3 Frame 6
2.2.2.4 Conjunto de Instruções da JVM 6
2.2.3 O Formato de Arquivo Class 8
2.2.3.1 A Estrutura de um Arquivo Class 8
2.2.3.1.1 A Estrutura Method 10
2.2.3.1.2 O Atributo Code 12

2.2.3.2 Lendo um Arquivo Class....14

2.3 A Execução de um Programa Java....16

Otimização de Código....18

3.1 As Principais Fontes de Otimização....19

3.2 Otimização *Peephole*....20

3.3 Algoritmos....21

3.3.1 Separação em Blocos Básicos....21

3.3.2 Construção do laço natural de um lado refluente....22

3.3.2 Definições Incidentes....23

3.3.3 Cômputo das Expressões Disponíveis....25

3.3.4 Análise das Variáveis Vivas....26

3.3.5 Eliminação Global das Sub-expressões Comuns....27

3.3.6 Propagação de Cópias....28

3.3.7 Detecção de Cômputos Laços Invariantes....29

3.3.8 Movimentação de Código....30

3.3.9 Eliminação das Variáveis de Indução....31

Implementação....35

Testes e Resultados....39

Conclusões e Trabalhos Futuros....41

Referências Bibliográficas....43

Anexo A....45

Anexo B....50

Anexo C....62

Lista de Figuras

Figura 1 – Programa para leitura do *Bytecode* Java 15
Figura 2 – Leitura de um método 15
Figura 3 – Índices da *constant_pool* 16
Figura 4 – Grafo de fluxo [AHO] 22
Figura 5 – Sub-expressões comuns potenciais através de blocos [AHO] 25
Figura 6 – Introdução de um pré-cabeçalho [AHO] 29
Figura 7 – Interface Gráfica Principal 36
Figura 8 – Abrindo Um Arquivo 36
Figura 9 – Seleção Das Otimizações A Serem Feitas 37
Figura 10 – Exibição Do Grafo De Fluxo 38
Figura 11 – Otimização 39

Lista de Tabelas

Tabela 1 - Interpretação das flags de acesso. 9
Tabela 2 - Interpretação das flags de um método...11

Lista de Siglas e Abreviaturas

API: *Application Programming Interface*

CPU: *Central Processing Unit* ou Unidade Central de Processamento. Processador.

DU: Definição-Uso

J2SE: *Java 2 Platform, Standard Edition.*

JVM: *Java Virtual Machine* ou Máquina Virtual Java.

NOP: *No operation* ou nenhuma operação

SDK: *Software Development Kit* ou Kit de Desenvolvimento de Software.

UD: Uso-definição.

WWW: *World Wide Web*

Glossário

APPLET: um programa aplicativo Java que usa o navegador do cliente para prover a interface com o usuário. **[FRE]**

ARQUIVO CLASS: Arquivo onde toda informação é armazenada no formato *class*.

ASSINATURA DE UM MÉTODO: Uma combinação que especifica o método, citando seu modificador de acesso (no caso de Java podem são três possíveis: *private*, *protected* e *public*), seu tipo de retorno, o seu nome e seus parâmetros.

BYTECODE: É o conjunto dos bytes armazenados em um arquivo organizado de acordo com o formato da estrutura *class*. São os bytes que serão lidos posteriormente pela JVM para a execução de um programa Java.

FORMATO CLASS: Formato de arquivo onde são armazenados os bytes correspondentes

INTERFACE: No caso desde trabalho, o sentido de interface será o de um tipo de dado comum na linguagem Java, e nunca no sentido de interface gráfica. Ao menos que seja especificado o contrário.

JAVAC: Um compilador Java que transforma o código-fonte Java no *bytecode* Java.

MULTITHREADING: Compartilhamento de um único CPU por várias tarefas ou *threads*.**[FRE]**

OPCODE: É a representação numérica de uma instrução da JVM. Toda instrução contém um mnemônico e seu opcode, podendo estar no formato decimal ou hexadecimal.

THREADS DAEMON: Uma *thread* é uma linha de controle dentro de um processo. E *threads daemon* são as *threads* que são executadas quando o processador possui algum tempo disponível, por exemplo, o coletor de lixo.

Resumo

A máquina virtual Java (JVM) é a grande responsável pela independência de hardware da linguagem, então são apresentados primeiramente conceitos para o seu entendimento. Em seguida vem a parte de otimização de código e a explicação de alguns dos principais algoritmos. Foi feito um programa para que se pudesse aplicar a otimização. Nele estão incluídas as opções para otimização e também um gerador dos grafos de fluxo. Infelizmente por questões de tempo só foi possível fazer a otimização peephole, mas a base foi feita e pode-se perfeitamente implementar tais algoritmos. Por fim, são apresentados os testes feitos e os resultados obtidos, com a visualização de uma otimização.

Abstract

The Java virtual machine (JVM) is the responsible one for the hardware independence of the language, then concepts for its agreement are presented first. After that it comes the part of code optimization and the explanation of some of the main algorithms. A program was made so that if it could apply the optimization. In it the options for otimização are included also a generator of the flow graphs. Unhappyly for time questions alone peephole was possible to make the optimization, but the base was made and can perfectly be implemented such algorithms. Finally, the tests and the gotten results are presented, with the visualization of an optimization.

Capítulo 1

Introdução

É notável a crescente da tecnologia Java nos dias de hoje. Entre os vários pontos positivos da linguagem Java está a independência de máquina, que na opinião do autor é a maior delas. Na contramão de todas as vantagens que a tecnologia Java trouxe, temos também uma grande desvantagem. Como Java faz uso de uma máquina virtual para se valer da condição de multiplataforma, tal desvantagem diz respeito ao desempenho de programas Java, sejam eles um simples aplicativo, *applet*, etc. Isto pode ser percebido por um usuário que execute tais programas com uma interface gráfica, por exemplo. Alheio a tudo isso, esta desvantagem serviu como um grande incentivo para este estudo.

Como solução para este tipo de problema, é natural que se pense logo em otimização de código. Entra aí uma parte interessante da área de compiladores. Foram para isso, pesquisados na bibliografia e escolhidos para serem apresentados alguns dos algoritmos julgados principais.

Esta monografia foi feita para quem sabe apontar um caminho para uma melhoria na eficiência de Java, e não no intuito de achar uma solução definitiva para o problema.

1.1 Justificativas

Embora os computadores pessoais de hoje em dia estejam cada vez mais rápidos, deve-se também levar em consideração que os aplicativos Java são desenvolvidos também em sistemas embutidos (processadores de uso específico) usado em máquinas de lavar roupas, geladeiras e celulares entre outros. Como esses dispositivos não têm a capacidade de processamento de um

computador pessoal, necessita-se de programas que executem com o mínimo de exigência do hardware e que forneçam um desempenho adequado.

Como desempenho não é o forte de Java, este trabalho visa pesquisar a influência dos algoritmos de otimização, na melhoria de performance do *bytecode*.

1.2 Objetivos

Este trabalho tem como objetivo:

- Estudar e compreender o *Bytecode* JAVA.
- Obter um bom conhecimento dos algoritmos utilizados para a otimização de código.
- Aplicar alguns dos algoritmos de otimização de código ao *Bytecode* JAVA.
- Inferir a influência destes algoritmos na melhoria de performance do bytecode.

1.3 Estrutura do trabalho

A estrutura do trabalho está assim definida:

Primeiramente, encontra-se uma breve apresentação da linguagem Java, bem como de alguns de seus conceitos. Procura se dar ênfase aos conceitos julgados úteis ao bom entendimento do trabalho, tais como a Máquina Virtual Java e seu *bytecode*.

Em seguida vem a parte de otimização de código, introduzindo seus fundamentos mais básicos, tal como as principais fontes de otimização, juntamente com a apresentação de seus algoritmos.

Posteriormente é feita uma explanação de como foi feita a implementação, seguido pela apresentação dos testes realizados e resultados obtidos.

Por fim, são feitas as considerações finais e também são lançadas idéias para aperfeiçoamento e extensão do trabalho.

Capítulo 2

A Linguagem de Programação Java

Java é uma linguagem de programação de alto-nível de propósito geral e orientada a objetos. Ela é fortemente tipada, tendo sua sintaxe muito parecida com C e C++. Possui muitas características interessantes, como: independência de máquina e de sistema operacional, suporte a *multithreading*, coleta de lixo automática, tratamento de exceções, uma grande biblioteca de classes (API), entre outras.

2.1 História

Em 1991 a *Sun Microsystems* financiava um projeto de codinome *Green* na área de dispositivos eletrônicos inteligentes destinados ao consumidor final. Como resultado, teve-se uma linguagem baseada em C e C++, chamada por James Gosling, seu criador, de *Oak* (carvalho) em homenagem à árvore que dava para a janela de seu escritório na *Sun*. Mais tarde, descobriu-se que já havia uma linguagem chamada de *Oak*. Foi quando numa cafeteria uma equipe da *Sun* viu o nome Java. O nome da cidade de origem de um tipo de café importado foi sugerido e pegou.

Infelizmente naquela época o projeto *Green* atravessava dificuldades, pois o mercado para dispositivos eletrônicos inteligentes não estava se desenvolvendo tão rapidamente quando o esperado. Para piorar a situação, um contrato importante pelo qual a *Sun* competia fora concedido a uma empresa concorrente. O projeto *Green* corria sérios riscos de ser cancela, quando em 1993 houve a explosão de popularidade da *World Wide Web (WWW)*. A partir daí, um grande potencial em Java foi visto para a criação de páginas com conteúdo dinâmico. Esta foi a alavanca que tanto o projeto necessitava.

Foi quando numa conferência em 1995, a *Sun* anuncia Java formalmente. Devido ao grande interesse das pessoas na WWW, Java chamou bastante atenção também. De lá para cá Java vem sendo usada nas mais diversas áreas.

2.2 Alguns conceitos Java

Nesta seção serão apresentados alguns conceitos de Java úteis para o desenvolvimento e entendimento do trabalho. Para mais detalhe, deve-se consultar as referências citadas.

2.2.1 *Bytecode*

Em uma linguagem como o C++, por exemplo, o código é compilado para uma máquina específica. Como conseqüência disso, o programa pode executar somente nessa máquina alvo. Diferentemente, o compilador Java não gera código de máquina, ou seja, instruções de hardware. Ao invés disso, ele gera os *bytecodes*, que são conjuntos dos bytes armazenados em um arquivo organizado de acordo com o formato da estrutura *class*, que será explicado posteriormente. Estes bytes que serão lidos posteriormente pela JVM para a execução de um programa Java. Nele está contida toda a informação necessária para a execução de um programa Java, como todos os *opcodes*, ou seja, as instruções da JVM (semelhante a uma linguagem assembly), juntamente com demais informações.

2.2.2 A Máquina Virtual Java (JVM)

Uma máquina virtual é uma simulação de um hardware, contendo suas principais características, como entrada e saída, interrupções, conjunto de instruções, etc.

A JVM é uma máquina abstrata responsável pela independência de hardware e de sistema operacional de Java. A JVM não sabe nada a respeito da linguagem de programação Java, somente de um formato binário particular, que é o formato de arquivo *class* que será explicado mais adiante. Entretanto, qualquer linguagem de programação que possa ser expressa nos termos de um arquivo *class* válido, pode ser usada pela JVM. Para se implementar uma JVM

corretamente, deve-se ser capaz de ler o formato de arquivo *class* executar corretamente suas operações.

Podem existir diversas implementações da JVM, cada qual com características próprias. Todo o estudo deste trabalho foi feito em cima da especificação de [LIN]. A versão usada foi a JVM vinda em conjunto com o *Java 2 Platform, Standard Edition, Software Development Kit* (J2SE 1.4.2 SDK) disponível em <http://www.java.sun.com/>.

2.2.2.1 Tipos

São dois os tipos que operam na JVM: os tipos primitivos e o tipo de referências. Estes tipos são aqueles armazenados em variáveis, passados como argumentos e retornados por métodos.

Os tipos primitivos dizem respeito aos valores numéricos inteiros(int, byte, short, long, char), aos valores numéricos de ponto flutuante(*float*, *double*), ao valor booleano(true, false) e ao valor de endereço de retorno. Este último tipo diz respeito a ponteiros para os opcodes das instruções da JVM, e é o único tipo primitivo que não está diretamente associado com nenhum tipo da linguagem de programação Java. Outro fato interessante é o de que a JVM não possui operações dedicadas para valores booleanos. Estes valores são compilados para serem usados como o tipo de dado int.

E os tipos de referências são os tipos de classes, de *array* e de interfaces. Seus valores são referências para instâncias de classes criadas dinamicamente, *arrays* ou instâncias de classes ou *arrays* que implementam interfaces. Um valor de referência pode ainda assumir a referência especial *null*, uma referência a nenhum objeto.

2.2.2.2 Áreas de Dados em Tempo de Execução

A JVM define várias áreas de dados que são usadas durante a execução de um programa. Algumas destas áreas são criadas na inicialização da JVM e são destruídas somente na sua saída. Algumas outras áreas são criadas por *thread*, estas são criadas quando as *threads* são criadas e destruídas quando as *threads* são destruídas.

Cada JVM possui seu próprio registrador pc (*program counter*), sua própria pilha e um *heap* que é compartilhado por todas as *threads* da JVM. Um *heap* é um lugar na memória para a

alocação de instâncias de classes e *arrays*. Possui também uma *method area*, que também é compartilhada entre todas as *threads* da JVM. A *method area* é análoga a área de armazenamento para o código compilado de uma linguagem convencional. As áreas de dados em tempo de execução têm também uma representação da constante tabela constant_pool no arquivo *class*. Podem possuir ainda pilhas de métodos nativos que, como o nome diz, suporta métodos nativos escritos em outra linguagem que não Java.

2.2.2.3 *Frame*

Um frame é usado para armazenar dados e resultados parciais, executar linkagem dinâmica, valores de retorno para métodos e enviar mensagens de exceções. Um novo frame é criado quando um método é invocado, e destruído quando a invocação do método é completada. Cada frame contém um *array* de variáveis locais da JVM, uma pilha de operandos. Para a linkagem dinâmica do código do método, cada frame possui uma referência para *runtime constant pool* para o tipo do método corrente.

2.2.2.4 Conjunto de Instruções da JVM

Como os hardwares reais, a JVM também possui o seu conjunto de instruções. Uma instrução da JVM consiste em um *opcode* de um byte que especifica a operação a ser executada, seguido de zero ou mais operandos. Muitas instruções não têm operandos e consistem somente em um *opcode*. Ignorando as exceções, o laço interno de um interpretador da JVM pode ser visto através do seguinte algoritmo:

faça
    busca um *opcode*;
    se existem operandos então busca os operandos;
    executa a ação correspondente ao *opcode*;
  enquanto existem mais *opcodes*;

São definidos vários tipos de instruções para a JVM. Dentre elas estão as instruções *load and store*, aritméticas, de conversão de tipos, manipulação e criação de objetos, gerenciamento da

pilha de operandos, transferência de controle, de retorno e invocação de métodos, sincronização e de lançamento de exceções.

Em anexo pode se encontrar uma tabela contendo todas as instruções da JVM, mostrando seus mnemônicos, *opcodes* decimais e hexadecimais, bem como o grupo no qual a instrução foi encaixada.

Uma outra coisa bastante importante para se possa fazer as otimizações são algumas seqüências de instruções que são geradas a partir de determinado código alto-nível. As instruções da JVM devem ser familiares a alguém que já tenha visto um código *assembly*. Cada instrução tem o seguinte formato: <opcode> [<operando1> <operando2> ...], onde o opcode é o número indicando o mnemônico da instrução. Ele é seguido de zero ou mais operandos, dependendo da instrução.

Agora vamos verificar algumas seqüências de instruções geradas a partir de algum código alto-nível. Este código gerado pode variar conforme o compilador, porém está de acordo com **[LIN]** e foi o que obteve-se em alguns testes com o compilador javac vindo com J2SE 1.4.2 SDK. Por exemplo, uma atribuição do tipo int t = 100; se tornaria[1]:

bipush 100

istore_1

A primeira instrução carrega a constante 100 para o topo da pilha, enquanto a segunda armazena a constante carregada. Já uma instrução aritmética como por exemplo a multiplicação de uma variável por 1, que é alvo de uma simplificação algébrica (explicada posteriormente) se pareceria com:

iload_1

iconst_1

imul

Neste caso, a primeira instrução armazena uma variável na pilha, a segunda carrega a constante 1 e a terceira multiplica os dois valores no topo da pilha, retirando-os e armazenando o resultado.

[1] As instruções estão representadas com seus mnemônicos ao invés de seus *opcodes* apenas para se tornar mais legível, o que não ocorre no arquivo class.

2.2.3 O Formato de Arquivo Class

Como dito anteriormente, o código-fonte Java é compilado em *bytecode* e posto em um arquivo *class* (cada definição de classe ou interface gerará um arquivo deste). A seguir será explicada toda a estrutura do arquivo *class*.

2.2.3.1 A Estrutura de um Arquivo Class

Um arquivo *class* possui a seguinte estrutura:

```
ArquivoClass {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Os tipos u1, u2 e u4 representam a quantidade de bytes sem sinal, um, dois ou quatro bytes respectivamente.

- *magic*: este item fornece o número que identifica o formato de arquivo *class* (0xCAFEBABE);

- *minor_version* e *major_version*: são respectivamente a menor e a maior versão deste arquivo *class*. Juntos, estes valores indicam a versão do arquivo. Se o arquivo contém em *major_version* o número M e em *minor_version* o número m, a versão do arquivo é denotada por M:m;
- *constant_pool_count*: seu valor é igual ao número de entradas na tabela *constant_pool* acrescido de um. Um índice *constant_pool* é considerado válido se ele é maior que zero e menor que *constant_pool_count*, com exceção às constantes do tipo long e double;
- *constant_pool*[]: é uma tabela de estruturas representando várias constantes string, nomes de classes e interfaces, nomes de campos, e outras constantes;
- *access_flags*: são máscaras das *flags* usadas para denotar as permissões de acesso para esta classe ou interface. A interpretação de cada *flag* é mostrada na tabela abaixo:

Nome do Flag	**Valor**	**Interpretação**
ACC_PUBLIC	0x0001	Declarada público; pode ser acessado de fora do seu pacote.
ACC_FINAL	0x0010	Declarada final; não é permitido herdar desta classe.
ACC_SUPER	0x0020	Trata métodos de superclasse especialmente quando invocada pela instrução invokespecial.
ACC_INTERFACE	0x0200	É uma interface, e não uma classe.
ACC_ABSTRACT	0x0400	Declarada abstrata; não pode ser instanciada.

Tabela 1 - Interpretação das flags de acesso.

Uma interface distingue de uma classe através do seu *flag* ACC_INTERFACE sendo setado. Caso contrário, este arquivo indica uma classe e não uma interface. E se o *flag* ACC_INTERFACE deste arquivo não é setado, quaisquer outros *flags* podem ser setados, exceto os *flags* ACC_FINAL e ACC_ABSTRACT. Por fim, o *flag* ACC_SUPER, quando setado, serve para indicar qual das duas alternativas semânticas para a instrução *invokeespecial* está sendo expressada. Isto tem como finalidade a compatibilidade com códigos gerados por compiladores antigos de Java.

- *this_class*: seu valor precisa ser um índice válido da tabela *constant_pool*. A entrada *constant_pool* desse índice precisa ser uma estrutura *CONSTANT_Class_info* representando a classe ou interface definida por este arquivo *class*;
- *super_class*: seu valor deve ser igual a zero ou um índice válido na tabela *constant_pool*. Se for diferente de zero, a entrada deste índice deve ser uma estrutura *CONSTANT_Class_info* representando a superclasse direta da classe definida neste arquivo. Se o valor for zero, então a classe definida no arquivo herda da classe *Object*;
- *interfaces_count*: seu valor corresponde ao número de *superinterfaces* diretas desta classe ou interface;
- *interfaces*[]: cada valor deste vetor é um índice válido na tabela *constant_pool* que corresponde a uma estrutura *CONSTANT_Class_info* representando uma *superinterface* direta desta classe ou interface definida no arquivo *class*;
- *fields_count*: seu valor corresponde ao número de estruturas *field_info* na tabela *fields*;
- *fields*[]: cada valor deste vetor deve indicar uma estrutura *field_info*;
- *methods_count*: seu valor corresponde ao número de estruturas *method_info* na tabela *methods*;
- *methods*[]: cada valor na tabela *methods* deve ser uma estrutura *method_info* correspondendo a uma descrição completa de um método desta classe ou interface. Neste ponto do *bytecode* que será dado o enfoque, mais exatamente no atributo *Code*, pois é nele que se encontram as instruções a serem otimizadas;
- *attributes_count*: dá o número atributos na tabela *attributes* desta classe;
- *attributes*[]: cada valor da tabela *atributes* deve ser uma estrutura do tipo *attribute*;

2.2.3.1.1 A Estrutura *Method*

Como já dito, dentro da estrutura method se encontram as instruções da JVM que serão otimizadas. Segue a estrutura:

```
method_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
```

u2 attributes_count;

attribute_info attributes[attributes_count];

}

- *access_flags*: o valor deste item é uma máscara usada para denotar a permissão e propriedades do método em questão. A tabela abaixo mostra os valores setados para configurar estes flags.

Nome do flag	Valor	Interpretação
ACC_PUBLIC	0x0001	Declarado como *public*.
ACC_PRIVATE	0x0002	Declarado como *private*.
ACC_PROTECTED	0x0004	Declarado como *protected*.
ACC_STATIC	0x0008	Declarado como *static*.
ACC_FINAL	0x0010	Declarado como final.
ACC_SYNCHRONIZED	0x0020	Declarado como synchronized.
ACC_NATIVE	0x0100	Declarado como native.
ACC_ABSTRACT	0x0400	Declarado como abstract.
ACC_STRICT	0x0800	Declarado como strictfp.

Tabela 2 - Interpretação das flags de um método.

- *name_index*: o valor deste item deve ser um índice válido na tabela *constant_pool*. A entrada na tabela *constant_pool* correspondente a este item deve ser uma *CONSTANT_Utf8_info* representando um dos nomes especiais <init> ou <clinit> ou ainda um nome de método válido em Java;
- *descriptor_index*: o valor deste item deve ser um índice válido na tabela *constant_pool*. A entrada na tabela *constant_pool* correspondente a este item deve ser uma *CONSTANT_Utf8_info* representando um descritor válido do método. Um descritor é uma string representando o método. Nele contém informações sobre seus parâmetros e tipo de retorno.
- *attributes_count*: o valor deste item dita o número de atributos adicionais deste método
- *attributes*: esta tabela tem em cada índice uma estrutura definindo um atributo para o método. São eles: *Exceptions* (indicam quais exceções o método pode lançar), *Synthetic*

(indicam os membros da classe que não aparecem no código-fonte), *Deprecated* (indica que o método já foi substituído por uma versão mais recente) e *Code* (explicado a seguir).

2.2.3.1.2 O Atributo *Code*

O atributo *Code* é encontrado nas estruturas *method_info*. Como já dito, é nele que estão contidas as instruções da JVM em conjunto com outras informações auxiliares de um simples método. Este atributo deve ser único em cada estrutura *method_info*. Um atributo *Code* possui a seguinte estrutura:

```
Atributo Code{
        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {       u2 start_pc;
                u2 end_pc;
                u2  handler_pc;
                u2  catch_type;
        }       exception_table[exception_table_length];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

- *attribute_name_index*: o valor deste item deve ser um índice válido na tabela *constant_pool*. A entrada na tabela *constant_pool* correspondente a este item deve ser uma *CONSTANT_Utf8_info* representando a string "*Code*";
- *attribute_length*: o valor deste item indica o tamanho do atributo, excluindo os seis bytes iniciais;

- *max_stack*: o valor deste item dá a profundidade máxima da pilha de operandos deste método em qualquer ponto durante a execução do mesmo;
- *max_locals*: o valor deste item dá o número de variáveis locais no *array* de variáveis locais alocadas durante a invocação deste método, incluindo aquelas passadas como parâmetro;
- *code_length*: o valor deste item dá o número de bytes no *array code* deste método. Deve ser maior que zero, pois o *array code* não pode ser vazio;
- *code*[]: é neste *array* que estarão contidos os bytes que representam as instruções da máquina virtual Java que implementam este método. Cada byte representa um número válido de um *opcode* (ver anexo A) seguido de zero ou mais bytes subseqüentes representando os operandos. Portanto é aqui onde deverão ser aplicadas as otimizações de código;
- *exception_table_length*: o valor deste item dita o número de entradas na tabela *exception_table*;
- *exception_table*[]: cada entrada nesta tabela descreve um *handler* de exceção, com a ordem dos *handlers* sendo relevante. Cada entrada nesta tabela possui os quatro seguintes itens:
    - *start_pc, end_pc*: estes dois valorem dão os intervalos no *array code* onde cada *handler* de exceção está ativo. O *start_pc* é inclusivo, enquanto *end_pc* (tem o invervalo fechado) é exclusivo (tem o intervalo aberto);
    - *handler_pc*: o valor deste item indica o começo do *handler* com um índice do *array code* válido, apontando, para um *opcode*;
    - *catch_type*: o valor deste item deve ser um índice válido na tabela *constant_pool*, indicando uma *CONSTANT_Class_info* que representa uma classe de exceções que este *handler* é capaz de tratar. Esta classe deve ser uma classe lançada ou uma de suas subclasses. O *handler* será chamado apenas se a exceção lançada for instância da classe dada ou de uma de suas subclasses. Se o valor de *catch_type* for igual a zero, o *handler* é chamado para todas exceções, sendo usado para implementar a construção de linguagem *finally*;
- *attributes_count*: o valor deste item indica o número de entradas da tabela *attribute*;
- *attributes*[]: cada entrada desta tabela representa uma estrutura de atributo. O atributo *Code*, por sua vez, pode ter um número variável de atributos associados a ele.

Atualmente, na especificação da segunda versão da JVM, são definidos dois atributos. São eles *LineNumberTable* e *LocalVariableTable*, ambos contém informações de depuração.

2.2.3.2 Lendo um Arquivo *Class*

Para o verdadeiro entendimento do formato de um arquivo *class*, foi necessário além de estudar a estrutura através de sua especificação, ler esta estrutura armazenada no arquivo. Para tal tarefa, foi desenvolvido um programa que lê toda a informação contida em um determinado arquivo *class*.

A linguagem escolhida para a implementação foi Java, por dois motivos principais. O primeiro é obviamente a afinidade da linguagem com o objetivo do programa. O segundo, e menos importante, é a afinidade do autor com a linguagem.

O programa feito lê os bytes contidos num arquivo *class* gerado pelo compilador. Os testes realizados foram feitos sobre arquivos gerados pelo compilador javac. Após aberto o arquivo, o programa então lê os bytes separando-os conforme forem armazenados. Como um inteiro é representado por 4 bytes, é fundamental que se leia esses 4 bytes contíguos, assim interpretando-os. De nada adiantaria lermos byte por byte e não termos nenhum significado. A API Java disponibiliza uma interface chamada *DataInput* do pacote java.io que implementa os métodos necessários para essas leituras.

A seguir, uma exibição da leitura de um arquivo class tirado das demonstrações vindas com o pacote Java 2 SDK, *Standard Edition*, versão 1.4.2. Este pacote pode ser descarregado direto no sítio de endereço http://www.java.sun.com/.

Figura 1 – Programa para leitura do *Bytecode* Java

Nesta outra figura temos a visualização de um método deste mesmo arquivo que foi lido acima.

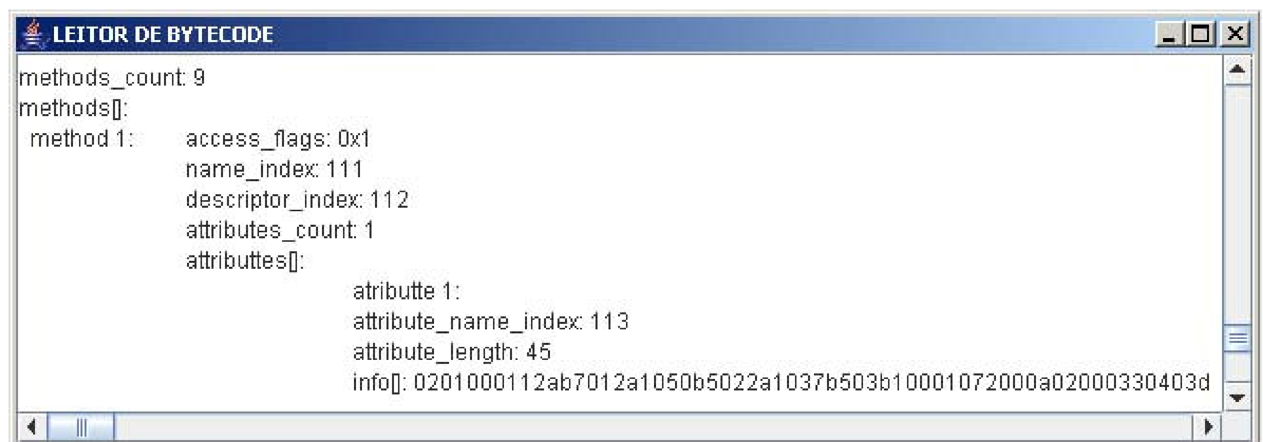

Figura 2 – Leitura de um método

Podemos ver o número de métodos (9) da classe. Tem-se também seu flag (0x1) informando que o método foi declarado como público, juntamente com os índices na tabela *constant_pool* de seu nome e descritor. Seguido pelo seu atributo que no caso é um atributo Code onde estão contidas as instruções JVM representado em info[]. Esta outra figura mostra a representação dos índices citados.

Figura 3 – Índices da *constant_pool*

O código-fonte do programa acima mostrado está em anexo.

2.3 A Execução de um Programa Java

Para saber como é lido o *bytecode* pela JVM, é necessário que se tenha no mínimo uma noção da execução de um programa Java. A seguir, serão brevemente esclarecidos alguns dos detalhes considerados pelo autor importantes.

A JVM começa a execução de um programa invocando o método *main* (que deve ser declarado como público, estático e sem tipo de retorno) de uma classe especificada à JVM de alguma forma (por exemplo, pela linha de comando), e passa então a este método como argumento um *array* de strings. É feita então a carga desta classe, linkagem e inicialização. Carga é o processo de busca da representação binária desta classe, e a criação de uma classe com essa representação binária. A linkagem envolve três passos: verificação, preparação e, opcionalmente, resolução.

A verificação na fase de linkagem checa se a representação da classe está bem formada, isto quer dizer que o código que implementa a classe em questão obedece aos requisitos semânticos da JVM. Caso não obedeça, um erro é lançado. Já a preparação diz respeito à alocação do armazenamento estático e quaisquer estrutura de dados usada internamente pela

máquina virtual, como a tabela de métodos. E a resolução é o processo que verifica as referências simbólicas da classe dada às outras classes ou interfaces, fazendo assim a carga destas outras classes e interfaces mencionadas, e checando se estas referências estão corretas. A inicialização de uma classe consiste na execução de seus incializadores estáticos e estáticos de *fields* (campos) declarados na classe em questão. No caso da inicialização de uma interface, seria a execução dos *fields* na interface.

Após este procedimento de inicialização, temos a criação de instâncias de classes que ocorre quando da avaliação de uma expressão de criação de instância de classe cria uma nova instância da classe cujo nome aparece nesta mesma expressão, ou ainda quando a invocação de um método de uma nova instância de uma classe cria uma nova instância de uma classe representada pelo objeto da classe a qual o método foi invocado. Os dois casos citados anteriormente, diz-se que uma nova instância de classe foi criada explicitamente. Também podem ser criadas instâncias de classes na carga de uma classe ou interface que contenham um literal *string*, criando assim o objeto da classe *String* representando aquele literal, ou com a execução do operador de concatenação de *strings* que não faça parte de uma expressão constante às vezes cria um objeto *String* para representar o resultado. Nestas duas últimas ocasiões, diz-se que as instâncias foram criadas implicitamente. Quando uma instância é criada explicita ou implicitamente, um espaço de memória é alocado para variáveis declaradas na classe e também em cada superclasse. Caso não haja espaço suficiente, então a execução termina abruptamente com um objeto *OutOfMemoryError*.

Para se finalizar uma instância de classe, o método finalize é chamado. Este é um método protegido da classe Object, porém pode ser sobrescrito por outras classes. Antes que o coletor de lixo recupere o espaço armazenado, a JVM invoca o método finalizador do objeto.

Como último passo, temos o término das atividades da JVM com a sua saída quando todas *threads* que não são do tipo *daemon* terminam ou quando alguma *thread* invoca o método *exit* da classe *Runtime* ou da classe *System* e quando essa operação é permitida pelo gerenciador de segurança.

Capítulo 3

Otimização de Código

No processo de compilação o ideal seria que todo o código gerado fosse tão bom quanto o programado manualmente por um experiente programador. Infelizmente apenas em raras ocasiões isto se torna uma verdade. Para contornar essa situação, uma solução bastante utilizada, inclusive pelos próprios compiladores, é fazer uso de algoritmos para efetuar melhoras no código gerado pelo compilador. Estes algoritmos são conhecidos como algoritmos de otimização de código.

Apesar do termo ser tradicional, deve-se perceber que ele é um tanto quanto equivocado. Isto se deve ao fato de que os algoritmos conhecidos como de otimização de código na grande maioria das vezes não conseguem produzir um código ótimo. Mesmo assim, podemos obter melhoras significativamente grandes no código, sejam elas na execução mais rápida do código, sejam na forma de fazê-lo ocupar menos espaço, ou ainda em ambas situações.

Para obtermos tais melhorias, é preciso visar as partes mais freqüentemente usadas do código, de forma a torná-las tão eficazes quanto o possível. Surgem aí alvos como os laços de repetição.

Neste ponto, é válido citar algumas das principais propriedades dos algoritmos de otimização de código. Uma primeira a ser percebida é a de que um algoritmo de otimização de código deve sempre preservar o significado de um programa, de forma a obter equivalência entre o código original e o código otimizado. Em outras palavras, a saída dos programas deve permanecer a mesma. Outra propriedade relevante é a de que uma transformação precisa acelerar os programas por um fator significativo. Nem sempre uma alteração melhora todo programa, e

em alguns casos pode até vir a retardar ligeiramente o mesmo. Isso ocorre em detrimento de uma melhora na média. Por fim, uma transformação precisa recompensar o esforço. De nada adianta gastarmos um tempo no processo de otimização se este esforço todo não valer a pena na hora da execução do programa. Como este estudo é feito com fins acadêmicos, isto pode nem sempre ser verdadeiro na implementação deste trabalho.

A seguir serão vistas as principais fontes de otimização.

3.1 As Principais Fontes de Otimização

A seguir são citadas algumas das principais fontes visadas para a obtenção de uma melhoria no código gerado. São elas:

- Sub-expressões Comuns: são aquelas em que se pode evitar o recômputo de determinada expressão fazendo-se uso dos valores calculados previamente.
- Propagação de cópias: a idéia da propagação de cópias é sempre se substituir g por f quando possível, após um enunciado do tipo f := g. A princípio isto pode não parecer uma otimização, mas pode-se notar que temos a oportunidade de eliminar a atribuição f := g.
- Eliminação de código morto: código morto consiste em todo aquele código que por mais que se execute dado programa ou se troque suas entradas, ele jamais chegará a ser executado. A eliminação de todo esse código inútil também é uma forma de otimização.
- Movimentação de código: Aplicada a um laço, esta transformação pega uma expressão que sempre produz o mesmo resultado independentemente do número de execuções feitos do laço e movimenta esta expressão para fora do mesmo. Esta é uma das grandes otimizações que podem ser feitas, ainda mais naqueles laços mais internos.
- Eliminação de variáveis de indução e redução de capacidade: Analisando atribuições do tipo j := j – 1 seguida de x := 4 * j em um laço, podemos chegar a conclusão de que j e x permanecem par e passo; a cada vez que o valor de j decresce de 1, o de x decresce de 4. Tais identificadores são chamados de variáveis de indução. Quando temos duas ou mais variáveis de indução, pode ser possível eliminar todas, com exceção de uma variável de indução.

- Otimizações de laços: é fato de que o tempo de execução de um programa é despendido em sua grande maioria nos laços de repetição, principalmente aqueles mais internos. Melhorando o código exatamente nesses pontos se conseguem grandes melhoras na qualidade do código gerado. As principais técnicas para otimização usadas nos laços de repetição são as citadas a seguir.

3.2 Otimização *Peephole*

Esta técnica diz respeito à observação de pequenas seqüências de instruções-alvo (ditas *peepholes*) e trocando-se estas instruções por uma outra seqüência melhorada, no sentido de ser mais curta ou mais rápida. São exemplos de transformações de programa que são características da otimização *peephole*:

- Eliminação de instruções redundantes: expressões que calculam por duas vezes uma mesma expressão e códigos inatingíveis são exemplos de eliminação de instruções redundantes.
- Otimizações de fluxo de controle: esporadicamente um gerador de código pode produzir desvios para desvios, desvios para desvios condicionais ou ainda desvios condicionais para desvios. Estes desvios desnecessários são também alvos da otimização peephole.
- Simplificações algébricas: Exemplos deste tipo de otimização poderia ser expressões do tipo x := x + 0 ou x := x * 1, pois elas não alteram o valor da variável à esquerda da atribuição. Ainda podemos pensar na troca de substituições mais custosas por outras equivalentes mais baratas na máquina-alvo. Um exemplo seria se trocar x := x * 2 por x := x + x.
- Uso de dialetos de máquina: Algumas máquinas, por exemplo, possuem modos de endereçamento auto-incrementantes e decrementantes. O uso desses modos melhora consideravelmente a qualidade do código gerado. Como Java é independente de máquina, fica apenas apresentado a título de curiosidade este tipo de otimização.

3.3 Algoritmos

3.3.1 Separação em Blocos Básicos

Este algoritmo tem por finalidade particionar um determinado conjunto de instruções, no nosso caso um conjunto de instruções da JVM, em blocos básicos. Ele é primordial para este trabalho, pois é a partir destes blocos básicos que será aplicada a otimização. Bloco básico é uma seqüência de enunciados consecutivos, na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final **[AHO]**. São comuns sobre os blocos básicos as seguintes transformações:

- Eliminação de sub-expressões comuns;
- Eliminação de código morto;
- Renomeação de variáveis temporárias e
- Intercâmbio de dois enunciados adjacentes independentes.

Segue o algoritmo para a separação de instruções em blocos básicos:

Entrada: um conjunto de instruções ou enunciados.

Saída: uma lista de blocos básicos.

1. Determinamos primeiro o conjunto de líderes, os primeiros enunciados dos blocos básicos. As regras que usamos são as seguintes:

a) O primeiro enunciado é um líder.

b) Qualquer enunciado que seja objeto de um desvio condicional ou incondicional é um líder.

c) Qualquer enunciado que siga imediatamente um enunciado de desvio condicional ou incondicional é um líder.

2. Para cada líder, seu bloco básico consiste no líder e em todos os enunciados até, mas não incluindo o próximo líder ou o final do programa.

Após termos obtido os blocos básicos, pode-se formar o que chamamos então de grafo de fluxo. O grafo de fluxo é um grafo direcionado onde seus nós são os próprios blocos básicos computados. Um nó é dito inicial quando o líder do bloco em questão é o primeiro enunciado. Segue um exemplo de Grafo de fluxo tirado de [AHO].

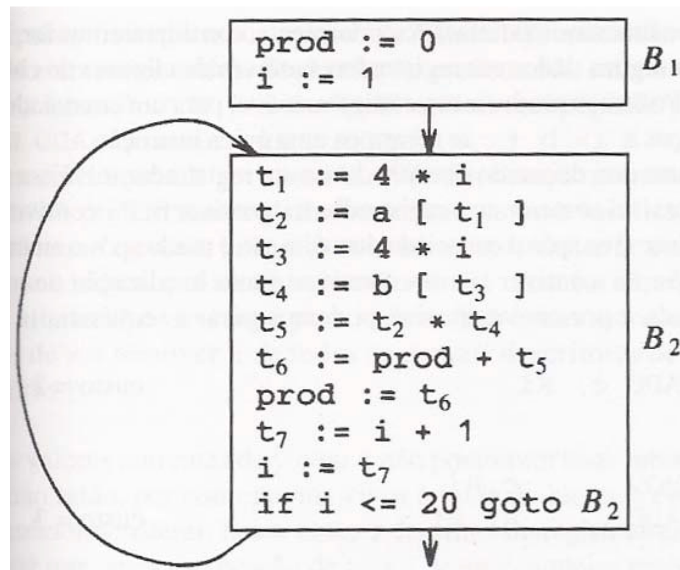

Figura 4 – Grafo de fluxo [AHO]

3.3.2 Construção do laço natural de um lado refluente

Antes de mais nada, é preciso que conheçamos o conceito de um nó dominando outro nó. É dito que um nó A domina um nó B se no percurso que vai do nó inicial ao nó B, for obrigatória a passagem pelo nó A. Um algoritmo para se encontrar os dominadores será apresentado posteriormente.

Em laços onde são possíveis as melhorias no código, são comuns duas propriedades. Primeiramente o laço precisa ter um único ponto de entrada, intitulado de cabeçalho, sendo este ponto o que domina todos os outros nós do laço. Segundo, deve existir pelo menos um percurso de volta para seu cabeçalho. E lados refluentes são aqueles lados onde as cabeças dominam as caudas, ou seja, sendo a → b um lado, b é a cabeça e a é a cauda. Conseqüentemente, os lados afluentes são aqueles que formam um grafo acíclico, no qual cada nó pode ser atingido a partir do nó inicial do grafo de fluxo.

Entrada: Um grafo de fluxo G e um lado refluente n → d.

Saída: O conjunto laço consistindo em todos os nós do laço natural de n → d.

Método: Começando pelo nó n, consideramos cada nó m ≠ d que sabemos que está em laço, de forma a assegurar que os predecessores de m sejam também colocados em laço. O algoritmo é dado a seguir. Cada nó em laço, exceto d, é colocado uma vez em pilha, de forma que seus

predecessores serão examinados. Note-se que, como d é colocado no laço inicialmente, nunca examinaremos seus predecessores e, por conseguinte, iremos encontrar aqueles nós que atingem n sem nunca passar através de d.

Procedimento inserir(m);
Se m não está em laço **então início**
    Laço := laço U {m};
    Empilhar m em pilha
Fim;

/* segue o programa principal */

pilha := vazio;
laço := {d};
inserir(n);
enquanto pilha não estiver vazia **faça início**
    desempilhar m, o primeiro elemento de pilha
    para cada predecessor p de m **faça** inserir(p)
fim

3.3.2 Definições Incidentes

Uma definição nada mais é do que um enunciado que atribui ou pode atribuir um valor a alguma variável. Num bloco básico é dito ter um ponto entre dois enunciados adjacentes, bem como antes do primeiro enunciado e após o último. E percurso uma seqüência de pontos, podendo ser ou não pertencentes ao mesmo bloco básico.

Uma definição d é incidente sobre um ponto p se existir um percurso do ponto imediatamente após d até p, sendo que d não esteja morta ao longo do percurso. E uma definição é morta entre dois pontos se existir uma nova definição para ela neste percurso.

Comumente armazenamos as informações sobre definições incidentes na forma de cadeias uso-definição, ou simplesmente cadeias-ud. As cadeias-ud nada mais são do que listas de todas as definições que atingem aquele uso, para cada uso de uma variável.

Outro aspecto importante para o entendimento deste algoritmo são os conjuntos de entrada, saída, geradas e mortas. Explicando, o conjunto de entrada é aquele contido pelas definições que atingem o início de um bloco S. Contrapondo, o conjunto de saída são aquelas definições para o final de S. Em geradas estão as definições que atingem o final de S, porém sem seguir por percursos fora de S. E em mortas estão contidas as definições que não atingem o final de S. Vale ressaltar que os conjuntos de geradas[S] e mortas[S] de um bloco básico S são computados de baixo para cima, ou seja da última definição para a primeira, enquanto os conjuntos entrada[S] e saída[S] são computados do primeiro enunciado para o último. Segue abaixo o algoritmo:

Entrada: Um grafo de fluxo para o qual mortas[B] e geradas[B] tenham sido computados para cada bloco B.

Saída: Entrada[B] e Saída[B] para cada bloco B.

Método: Usamos uma abordagem iterativa, começando com a “estimativa” entrada[B] = Ø para todos os conjuntos B e convergindo para os valores desejados de entrada e saída. Na medida em que precisamos iterar até que os conjuntos entrada (e, por conseguinte, os conjuntos de saída) convirjam, usaremos a variável mudou para registrar, a cada passagem através dos blocos, se algum conjunto entrada foi modificado. O algoritmo é exibido a seguir.

/* inicializar saída na suposição de que entrada[B] = Ø, para todos os B’s */

para cada bloco B **faça** saída[B] := geradas[B];

mudou := **true**; /*para fazer o laço while “pegar no tranco”*/

enquanto mudou **faça início**

    mudou := **false**;

    para cada bloco B **faça** início

        entrada[B] :=   U saída[P]; /*P é um predecessor de B*/

        saída_anterior := saída[B];

        saída[B] := geradas[B] U (entrada[B] – mortas[B]);

        se saída[B] ≠ saída_anterior **então** mudou := **true**;

    fim

fim

3.3.3 Cômputo das Expressões Disponíveis

Basicamente o uso de expressões disponíveis está ligado à detecção de sub-expressões comuns. Na figura abaixo a expressão 4 * i no bloco B_3 será uma sub-expressão comum se 4 * i estiver disponível no ponto de entrada do mesmo bloco B_3, ou seja, i não tenha recebido nenhum valor em bloco B_2 ou se como em (b) tenha sido recomputado após tal operação.

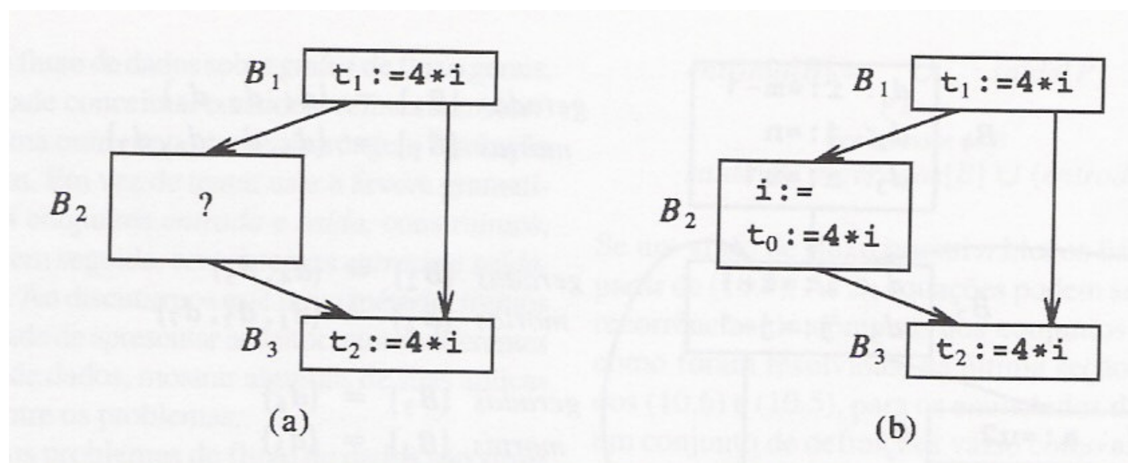

Figura 5 – Sub-expressões comuns potenciais através de blocos [AHO]

Diz-se que uma expressão está disponível em um ponto p se para cada percurso a partir do nó inicial até p, avalia x + y e, se não existir nenhuma atribuição à *x* ou a *y* posterior ao ponto p. Se houver subseqüentemente alguma atribuição a *x* ou a *y* e o valor de x + y não for recomputado, é dito que o bloco em questão mata a expressão x + y. Contrariamente, se um bloco avalia x + y e não redefine posteriormente x ou y, diz-se que o bloco gera a expressão x + y.

Na notação usada neste algoritmo, leia-se U como o conjunto universal de todas expressões que figurem no lado direito de um ou mais enunciados do programa, e_mortas[B] como sendo o conjunto de expressões mortas de U mortas em B e e_geradas[B] como sendo as expressões geradas pelo bloco B.

Entrada: Um grafo de fluxo G com e_mortas[B] e e_geradas[B] comparados para cada bloco B. O bloco inicial é B_1.

Saída: O conjunto entrada[B] para cada bloco B.

Método: Executar o algoritmo seguinte. A explicação dos passos é similar àquela de definições incidentes.

```
entrada[B1] := Ø ;
saída[B1] := geradas_e[B1];
para B ≠ B1 faça saída[B] := U – e_mortas[B];
mudou := true;
enquanto mudou faça início
        mudou := false;
        para B ≠ B1 faça início
                entrada[B] := U saída[P]; /*P é um predecessor de B*/
                saída-anterior := saída[B];
                saída[B] := e_geradas[B] U (entrada[B] – e_mortas[B]);
                se saída[B] ≠ saída-anterior então mudou := true;
        fim
fim
```

3.3.4 Análise das Variáveis Vivas

Uma variável x está viva em um ponto p se seu valor puder ser usado em algum percurso durante o bloco B a partir deste mesmo ponto p, caso contrário diz-se que a variável está morta. Este conceito é fundamentalmente usado na geração de um código objeto, por exemplo, para sabermos se é necessário ou não armazenar determinado valor num registrador.

Os conceitos para a compreensão deste algoritmo estão em torno dos conjuntos entrada, saída, definidas e usos. Entrada são as variáveis vivas no ponto inicial de B e saída no último ponto do mesmo B. O conjunto das definidas é constituído por aquelas variáveis que tenham valores atribuídos em B antes de qualquer uso dessas variáveis em B enquanto que usos é composto pelas variáveis que possam ser usadas em B antes de quaisquer definições dessas variáveis.

Entrada: Um grafo de fluxo com os conjuntos de definidas e usos computados para cada bloco.
Saída: Saída[B], o conjuntos de variáveis vivas à saída de cada bloco B do grafo de fluxo.
Método: Executar o programa dado a seguir.

para cada bloco B **faça** entrada[B]:= Ø;

enquanto ocorrerem mudanças a qualquer um dos conjuntos entrada **faça**

para cada bloco B **faça** início

saída[B] := U entrada[S] /*S é um sucessor de B*/

entrada[B] := usadas[B] U (saída[B] – definidas[B])

fim

fim

3.3.5 Eliminação Global das Sub-expressões Comuns

Os algoritmos apresentados anteriormente não eram em si para a otimização de código. Eles formam uma base, para os algoritmos que irão efetivamente otimizar um código em questão. Todo um cômputo é necessário ser feito para cada algoritmo, tais como os que foram apresentados. Deste algoritmo em diante é que de fato, são de otimizações de código.

Este algoritmo, como demonstra o nome, tem por finalidade a eliminação das sub-expressões comuns.

Entrada: Um grafo de fluxo com informações a respeito das expressões disponíveis.
Saída: Um grafo de fluxo revisado.
Método: Para cada enunciado s da forma x := y + z tal que y + z esteja disponível ao início do bloco de *s* e nem *y* nem *z* sejam definidos antes do enunciado *s* naquele bloco, fazer o seguinte.

1. Para descobrir as avaliações de y + z que atingem o bloco de *s*, seguimos os lados do grafo de fluxo, procurando de frente para trás a partir do bloco de *s*. No entanto, não seguimos através de qualquer bloco que avalie y + z. A última avaliação de y + z em cada bloco encontrado é uma avaliação de y + z que atinge *s*.
2. Criar uma nova variável u.
3. Substituir cada enunciado w := y + z encontrado em (1) por

   u := y + z

   w := u
4. Substituir o enunciado s por x := u.

3.3.6 Propagação de Cópias

Os geradores de código comumente inserem enunciados de cópia na forma de x := y. Este é o caso também de alguns dos algoritmos de otimização de código. Se conseguirmos localizar todos os locais onde os enunciados de cópia forem usados, é possível eliminá-los fazendo a substituição de *x* por *y* em todos os locais. Para isso algumas condições devem ser atendidas:

1. O enunciado *s* precisa ser a única definição de *x* que atinja *u* (isto é, a cadeia-ud para o uso u consiste somente de *s*).
2. A cada percurso de *s* para *u*, incluindo os percursos que vão através de *u* várias vezes (mas não passam por *s* uma segunda vez), não existam atribuições a *y*.

A primeira condição pode ser averiguada através das informações contidas nos encadeamentos-ud. Já para satisfazer a segunda condição temos de redefinir os conjuntos já citados. Num bloco básico b, o conjunto entrada[B] fica sendo o conjunto de cópias tais que cada percurso a partir do nó inicial até o início de B contém o enunciado s e subseqüentemente a y dentro de B. Saída[B] passa a ser o correspondente definido só que com relação ao final de B. U será o conjunto universal de todas os enunciados de cópia no programa. Temos novos conjuntos como c_geradas[B] representando o conjunto das cópias geradas no bloco B, enquanto c_mortas[B] é o conjunto de cópias mortas no bloco B. Segue o algoritmo.

Entrada: Um grafo de fluxo G com cadeias-ud fornecendo as definições incidentes ao bloco B e com c_entrada[B] representando o conjunto de cópias x := y que atingem ao bloco B ao longo de cada percurso, sem atribuições a *x* ou *y* se seguindo à última ocorrência de x := y no percurso. Precisamos também das cadeias-du fornecendo os usos de cada definição.

Saída: Um grafo de fluxo revisado.

Método: Para cada cópia s: x := y, fazer o seguinte:

1. Determinar aqueles usos de *x* que são atingidos por esta definição de *x*, nominalmente, s: x:= y.
2. Determinar se para cada uso de *x* encontrado em (1), *s* está em c_entrada[B], onde B é o bloco deste uso particular e, sobretudo, se nenhuma definição de *x* ou de *y* ocorre antes desse uso de *x* dentro de B. Relembrar que se *s* está em c_entrada[B], então *s* é a única definição de *x* que atinge B.
3. Se s atende às condições de (2), então remover *s* e substituir por *y*, todos os usos de *x* encontrados em (1).

3.3.7 Detecção de Câmputos Laços Invariantes

Um laço invariante como sugere o nome, é uma computação feita dentro de um laço onde os valores envolvidos não se alteram durante a execução deste mesmo laço. Como exemplo de laço invariante pode-se ter uma atribuição x := x + z dentro de um laço onde todas as definições possíveis de y e z estejam fora deste laço, pois o valor de x não vai se alterar na execução do laço.

Para descobrirmos os laços invariantes podemos fazer uso das cadeias-ud quanto as cadeias-du (que é a computação de um conjunto de usos s de uma variável x para um ponto p, tal que exista um percurso de p até s que não redefina x) que nos informam onde um valor poderá ser usado nos restando somente verificar quais usos desta variável em questão não utiliza outra definição desta mesma variável.

As atribuições laço invariantes poderão ser deslocadas para os pré-cabeçalhos que são blocos criados a partir de um grafo de fluxo, sendo que este bloco tem como seu sucessor somente o cabeçalho do bloco em questão como mostra a figura a seguir.

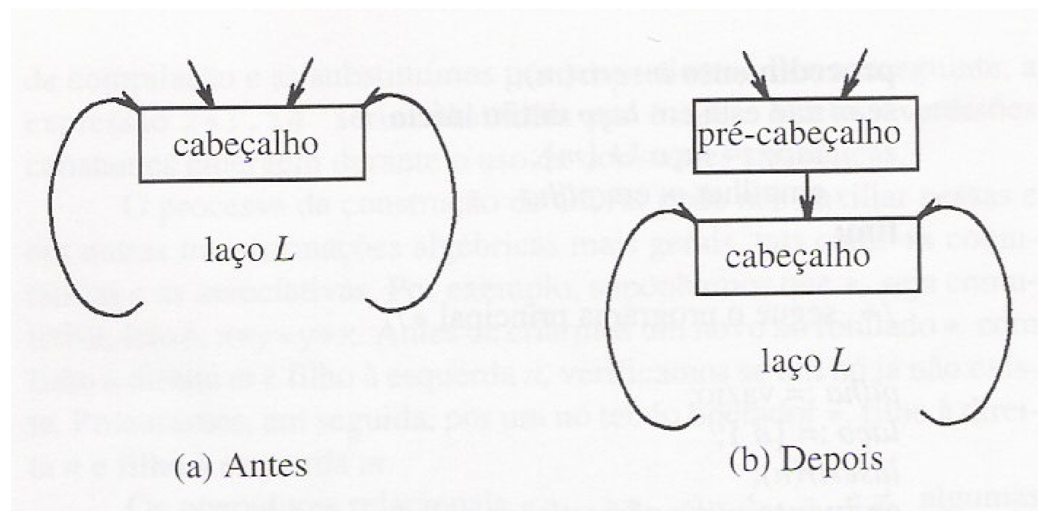

Figura 6 – Introdução de um pré-cabeçalho [AHO]

Abaixo temos o algoritmo.

Entrada: Um laço L consistindo em um conjunto de blocos básicos, cada bloco contendo uma seqüência de enunciados de três endereços. Assumimos que as cadeias-ud, computadas como na seção 3.3.5, estejam disponíveis para os enunciados individuais.
Saída: O conjunto de enunciados de três endereços que computa o mesmo valor a cada vez que for executado, do momento em que o controle entra no laço L até o instante em que o deixa.
Método: Forneceremos uma especificação um tanto informal do algoritmo, acreditando que os princípios ficarão claros.

1. Marcar como "invariantes" aqueles enunciados cujos operandos sejam todos constantes ou tenham todas as suas definições incidentes fora do laço.
2. Repetir o passo (3) até que em alguma repetição não haja novos enunciados marcados como "invariantes".
3. Marcar como invariantes todos aqueles enunciados não previamente marcados dessa forma, cujos operandos sejam ou constantes, ou tenham todas as suas definições incidentes fora do laço, ou tenham exatamente uma definição incidente e que essa definição seja um enunciado do laço, marcado como invariante.

3.3.8 Movimentação de Código

Depois de termos encontrado as atribuições invariantes dentro de um laço podemos realizar então a movimentação de código contanto que esta não altere o valor que o programa computa. Logicamente é melhor não fazer uma alteração se não existe a certeza de que ela é segura, ou seja, não altera a computação do programa. Para isso, existem algumas condições a serem verificadas: **[AHO]**

1. Um bloco contendo s domina todos os nós de saída do laço, onde uma saída de um laço é um nó com um sucessor fora do laço.
2. Não há outro enunciado no laço que atribua valor a x. De novo, se x for uma variável temporária, recebendo atribuição somente uma vez, esta condição é certamente satisfeita e não precisa ser verificada.

3. Nenhum uso de x no laço é atingido por qualquer definição de x que não s. Esta condição também está satisfeita, normalmente, se x for uma variável temporária.

Entrada: Um laço L com informações do encadeamento-ud e informações a respeito do dominador.

Saída: Uma versão revisada do laço com um pré-cabeçalho e (possivelmente) alguns enunciados movidos para o pré-cabeçalho.

Método:

1. Usar o algoritmo da seção 3.3.7 para encontrar os enunciados laço invariantes.
2. Para cada enunciado *s* definindo *x*, encontrado no passo (1), verificar:
    a. Se está num bloco que domine todas as saídas de L
    b. Se x não é definido em algum local de L e
    c. Se todos os usos em L de x podem somente ser atingidos pela definição de *x* no enunciado *s*.
3. Mover, na ordem encontrada pelo algoritmo da seção 3.3.7, cada enunciado *s* encontrado em (1) e, atendendo às condições (a), (b) e (c), para o novo pré-cabeçalho criado, providenciado que quaisquer operandos de *s*, que sejam definidos no laço L (no caso de *s* ter sido encontrado no passo (3) do algoritmo da seção 3.3.7) tenham tido seus enunciados de definição previamente movidos para o pré-cabeçalho.

3.3.9 Eliminação das Variáveis de Indução

Variável de indução é aquela que dentro de um certo laço é incrementada ou decrementada por alguma constante a cada iteração. Neste algoritmo será abordado apenas para aquelas variáveis *x* que são consideradas básicas, ou seja, que se encontram numa atribuição do tipo i := i ± c, onde c é uma constante. Em seguida é apresentado o algoritmo que detecta tais variáveis.

Entrada: Um laço L com informações das definições incidentes de do cômputo laço-invariante (provenientes do algoritmo da seção 3.3.7).

Saída: Um conjunto de variáveis de indução. Associada a cada variável de indução *j* está uma tripla (*i*, *c*, *d*), onde *i* é uma variável básica de indução *e, c* e *d* são constantes tais que o valor de *j* é dado por c * i + d, no ponto onde *j* é definida. Dizemos que *j* pertence à família de *i*. A variável básica de indução *i* pertence à sua própria família.

Método:

1. Encontrar todas as variáveis básicas de indução esquadrinhando os enunciados de L. Usamos aqui as informações do cômputo laço-invariante. Associada a cada variável básica de indução está a tripla (i, 1, 0).
2. Procurar pelas variáveis *k*, com uma única atribuição a *k* dentro de L, tendo um das seguintes formas:

   k := j * b, k := b * j, k := j / b, k := j ± b, k := b ± j

   onde *b* é uma constante e *j* uma variável de indução, básica ou não.

Depois de ter encontrado as variáveis de indução, trocamos então as instruções para computar uma variável de indução de forma que use adições ou subtrações ao invés de multiplicações. Essa substituição de uma operação mais lenta por outra mais rápida consiste no que chamamos de redução de capacidade. A seguir é demonstrado o algoritmo de redução de capacidade aplicada a variáveis de indução.

Entrada: Um laço L com informações das definições incidentes e das famílias de variáveis de indução, computadas usando o algoritmo descrito acima.

Saída: Um laço revisado.

Método: Consideremos uma variável básica de indução *i* de cada vez. Para cada variável de indução *j*, na família de *i*, com tripla (i, c, d):

1. Criar uma nova variável *s* (mas, se duas variáveis j_1 e j_2 tiverem triplas iguais, criar somente uma nova variável para ambas).
2. Substituir a atribuição de *j* por j := s.
3. Imediatamente após cada atribuição i := i + n em L, onde n é uma constante, atrelar s := s + c * n; onde a expressão c * n é avaliada como uma constante, já que *c* e *n* são constantes. Colocar s na família de *i*, com tripla (i, c, d).
4. Resta assegurar que *s* seja inicializada com c * i + d à entrada do laço. A inicialização pode ser colocada ao final do pré-cabeçalho. A inicialiação consiste em s := c * i, somente

s := se *c* for igual a 1; s := s + d, omitindo *d* quando este for igual a 0. Note-se que *s* é uma variável de indução da família de *i*.

Agora, após a aplicação dos dois algoritmos anteriores, é que podemos eliminar as variáveis de indução. Para se fazer isso, basta seguir os passos deste algoritmo bastante descritivo:

Entrada: Um laço L com informações sobre definições incidentes, cômputos laço-invariantes e variáveis vivas.
Saída: Um laço revisado
Método:

1. Considerar cada variável básica de indução *i* cujos únicos usos sejam para computar outras variáveis de indução em sua família e em desvios condicionais. Considerar algum *j* na família de *i*, preferivelmente uma em que *c* e *d* em sua tripla (i, c, d) sejam tão simples quanto possível (isto é, preferimos c = 1 e d = 0) e modificamos cada teste em que *i* apareça de modo a usar j no lugar. Assumimos, no que se segue, que c seja positivo Um teste da forma if *i op-relacional x goto B*, onde *x* não seja uma variável de indução é substituído por

   r := c * x
   r := r + d
   if j op-relacional goto B

   onde *r* é uma nova variável temporária. O caso *if x op-relacional i goto B* é tratado analogamente. Se existirem duas variáveis de indução *i* e *i* no teste *if i op-relacional i goto B*, checamos então se ambas, i_1 e i_2, podem ser substituídas. O caso simples é quando temos j_1 com tripla (i_1, c_1, d_1) e j_2 com tripla (i_2, c_2, d_2) e $c_1 = c_2$ e $d_1 = d_2$. Então, i_1 op-relacional i_2 é equivalente a j_1 *op-relacional* j_2. Em casos mais complexos, a substituição do teste pode não valer a pena, porque podemos necessitar introduzir dois passos multiplicativos e um aditivo, enquanto apenas dois passos podem ser economizados eliminando-se i_1 e i_2. Finalmente, remover todas as atribuições às variáveis de indução eliminadas a partir do laço L, na medida em que agora serão inúteis.
2. Consideremos agora cada variável de indução *j*, para as quais um enunciado j := s foi gerado pelo algoritmo descrito acima. Primeiro, verificamos que podem não haver atribuições a s entre o enunciado introduzido j := s e o uso de *j*. Na situação usual, *j* é usada no bloco no qual é definida, simplificando essa verificação; em caso contrário, as

informações a respeito das definições incidentes mais alguma análise do grafo é necessitada para implementar a verificação. Substituir, então, todos os usos de *j* por usos de *s* e remover o enunciado j := s.

Capítulo 4

Implementação

Na aplicação prática dos algoritmos apontados no capítulo anterior, foi feita uma implementação para que o experimento fosse realizado. Foi também adicionada a funcionalidade de contrução do grafo de fluxo de acordo com as instruções retiradas dos métodos pertencentes ao arquivo a ser lido. A codificação deste programa para a otimização foi construída também na linguagem Java (por motivos já citados).

A figura abaixo ilustra a interface gráfica desenvolvida para que se possa amigavelmente fazer uso da otimização. Pode se ver duas caixas de texto no programa. A caixa à esquerda exibe o resultado da leitura do *bytecode* original, sem as otimizações. Para isto foi usado o programa desenvolvido e mostrado na seção 2.2.3.2. Na caixa de texto é mostrado o *bytecode* otimizado, lido pelo mesmo programa. Pode se ver cinco botões na interface gráfica. O primeiro da esquerda pra direita é utilizado para abrir o arquivo *class* no qual as otimizações serão realizadas. O segundo abre também um arquivo, porém o qual será o arquivo com as instruções otimizadas. Entre outras palavras, o primeiro abre um arquivo para a leitura e o outro para a escrita. A figura 8 ilustra a caixa de seleção para a escolha de um arquivo usado por ambos botões explanados. A terminologia usada foi arquivo origem e destino, respectivamente.

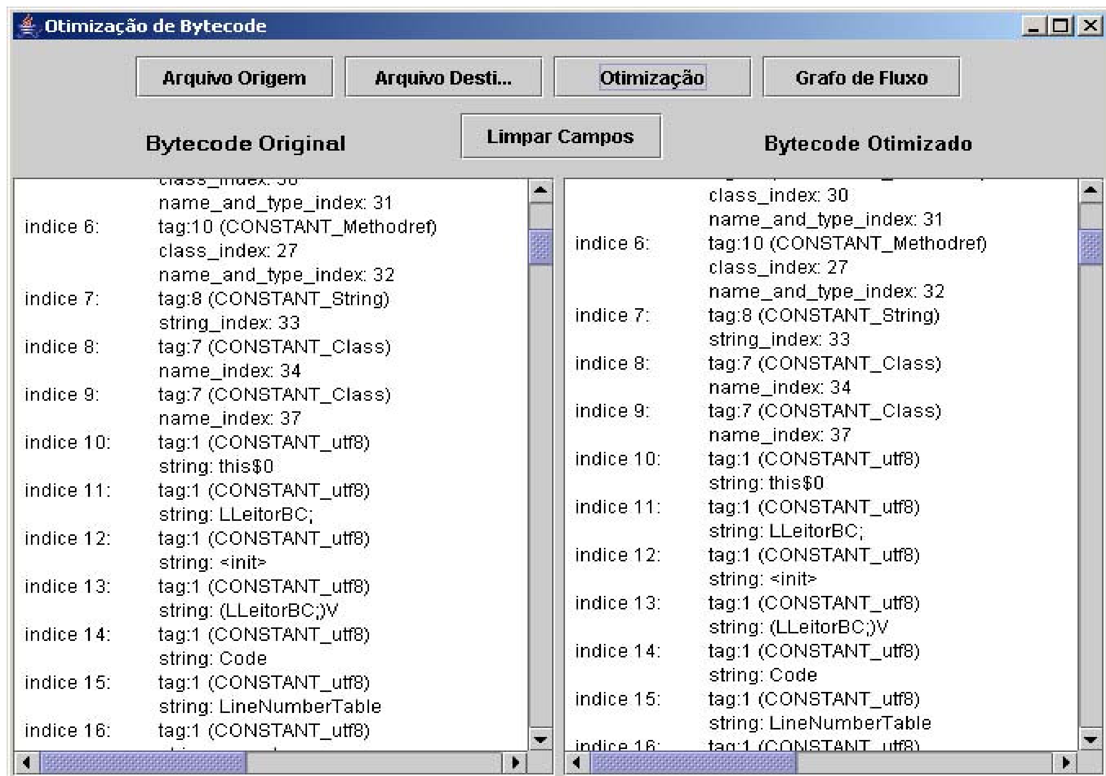

Figura 7 – Interface Gráfica Principal

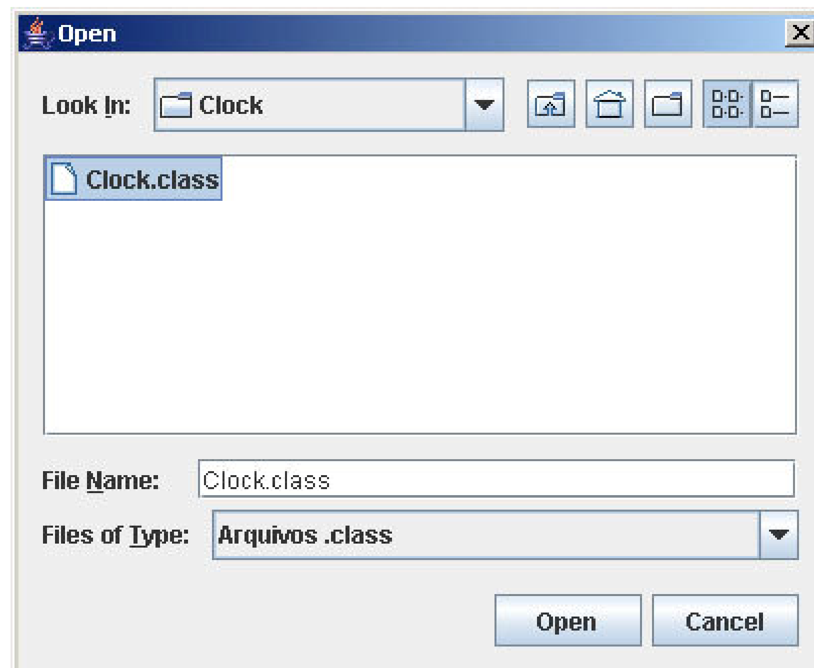

Figura 8 – Abrindo Um Arquivo

No terceiro botão é que está a otimização. Ele só será habilitado depois ter sido escolhido os dois arquivos (origem e destino). Esta figura abaixo mostra a janela que é aberta após o clique no botão. Nesta janela podem ser escolhidos quais os tipos de arquivo que se deseja aplicar. Como padrão, estão selecionadas todas otimizações.

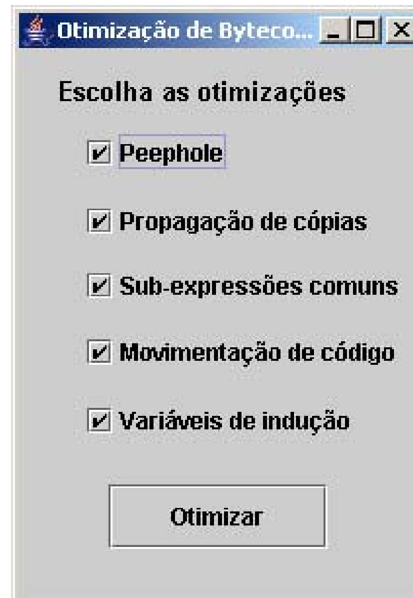

Figura 9 – Seleção Das Otimizações A Serem Feitas

Após a otimização, o *bytecode* gerado é automaticamente lido e exibido na caixa de texto da esquerda. A próxima figura ilustra a exibição de um grafo de fluxo, no qual é a funcionalidade do quarto botão. Nesta janela são exibidas as instruções de cada método existente no *bytecode* lido. Abaixo disto estão as mesmas instruções, porém já separadas em blocos básicos. Os blocos são numerados de um até *n*, onde *n* é o número de blocos básicos deste método. Seguido dos blocos básicos vem uma tabela representando o grafo de fluxo. A tabela é composta dos blocos (representando as linhas) x entrada/saída(representando as colunas). A tabela indica quais blocos tem uma aresta entrando neste bloco em questão. Da mesma forma, indica para quais outros blocos saem suas arestas.

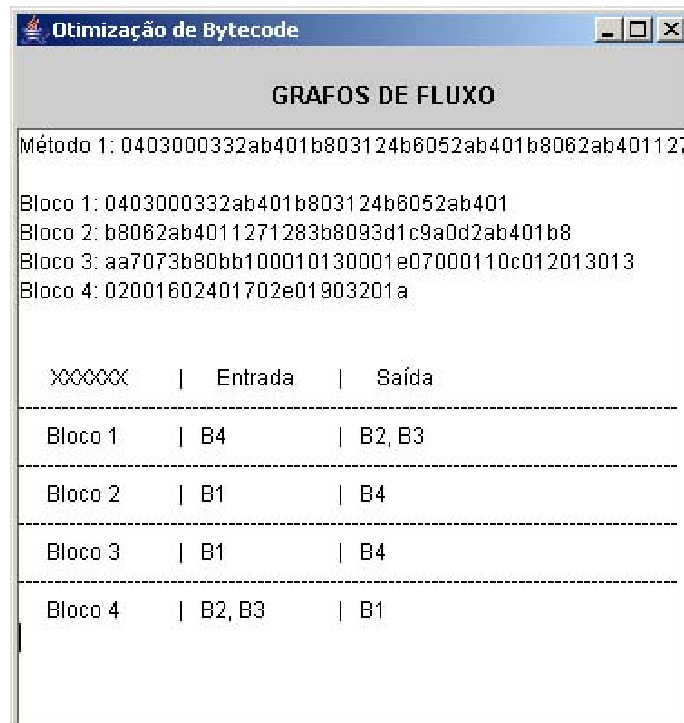

Figura 10 – Exibição Do Grafo De Fluxo

O último botão, situado abaixo dos outros quatro, da figura 7 tem a função de limpar os dois campos de texto, para caso queira se abrir outro arquivo para otimização.

Infelizmente por questão de tempo, foi implementada apenas a otimização *peephole* de cálculo de simplificação algébrica e de desvios para desvio. Como as instruções de desvio contêm como operando o endereço alvo do desvio, seria necessário recalcular o endereço quando se retirasse uma instrução. Para evitar este problema, estas instruções ao invés de serem retiradas são substituídas pelo *opcode* 0, que representa a instrução nop (*no operation*). Tomemos como exemplo uma simplificação algébrica. Se tivermos a instrução iconst_0 seguida pela instrução isub, ambas podem ser substituídas pela instrução nop, pois iconst_0 carrega a constante 0, e isub subtrai os dois valores no topo da pilha de operandos e os substitui pelo resultado. Como qualquer número subtraído de zero é igual a ele próprio, podemos então retirar estas duas instruções. Os resultados obtidos são apresentados na seção seguinte.

Capítulo 5

Testes e Resultados

Para que se possa fazer alguma afirmação em torno do tempo de execução ganho com a otimização, é necessário um estudo que analise a aplicação em um grande volume de arquivos. Infelizmente não foi possível fazer tal análise. Entretanto é válido mostrar como a otimização foi feita e seu resultado para que se tenha certeza que a otimização funciona corretamente.

Para de fato se ver as otimizações foi usado o programa desenvolvido e apresentado na seção 2.2.3.2. Através dele podemos ver que as instruções realmente estão sendo aplicadas como mostra a figura a seguir.

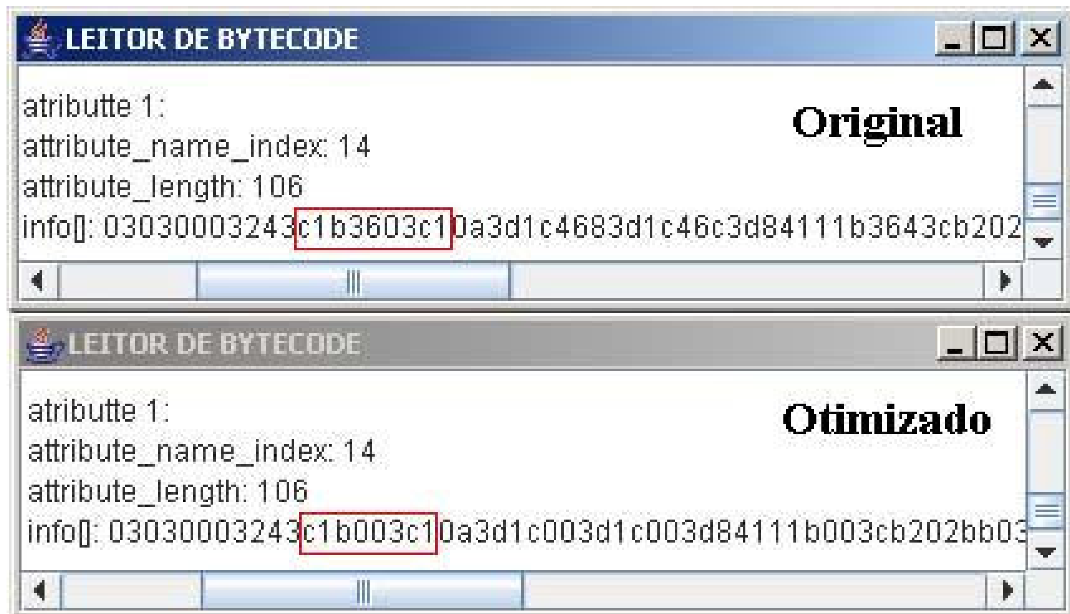

Figura 11 – Otimização

Na figura são mostrados dois trechos de bytecodes lidos através do programa de leitura. Em cima temos o *bytecode* original, anterior à otimização, enquanto que embaixo é mostrado o mesmo *bytecode*, porém já otimizado. Note-se nos trechos explicitados com um retângulo vermelho temos explicitado o ponto onde ocorreu uma otimização através da inserção do *opcode* da instrução nop (*opcode* 0).

Primeiramente foram feitos alguns códigos forçados, ou seja, que era escritos de forma a precisar de otimização. Isto foi feito apenas para ter a certeza de que a otimização iria agir. Depois foram testados alguns outros programas. Estes programas foram na sua maioria os trabalhos desenvolvidos ao longo de todo o curso. Foi pego também trabalhos de colegas para que se tivesse maior material para teste. Foram testados ainda as *applets* de demonstração com o pacote Java 2 SDK, *Standard Edition*, versão 1.4.2.

Como era esperado, após a realização das otimizações, os programas testados aparentaram funcionar corretamente com a execução do arquivo class otimizado.

Capítulo 6

Conclusões e Trabalhos Futuros

Independentemente dos resultados obtidos e apresentados anteriormente, é inegável que este trabalho tenha sido de grande valia para o autor. Este trabalho pode ser visto como um passeio ao longo de todo o curso. Para o entendimento do conteúdo empregado, fez-se necessário uma lembrança ou reaprendizado de vários temas base de ciências da computação vistos ao longo do curso. Na parte de otimização de código, foram revistos tópicos de fundamentos matemáticos como a teoria de conjuntos e algumas de suas operações mais básicas como união, intersecção e a diferença entre dois conjuntos. A teoria dos grafos também se fez presente com alguns conceitos. Conceitos ainda da parte de sistemas operacionais, como *threads* e máquina virtual juntamente com seu funcionamento, na área de compilação como a própria otimização e a parte que tem junção com aquelas advindas das estruturas de dados (em sua maioria a de árvores) foram aprendidas e/ou revistos. Também há de se ressaltar a revisão feita em torno da linguagem *assembly* com o foco na aprendizagem nas instruções da JVM.

A quantidade de conhecimentos aprendidos neste curso de graduação e que foram revistos no desenvolvimento desta monografia ainda pode ser acrescido por aqueles de parte mais prática, como a constante melhoria na programação da linguagem Java e suas bibliotecas. Foram abordados ainda, uma explanação da JVM seu funcionamento e, principalmente, a organização do seu *bytecode*.

Visto tudo isso, chega-se à conclusão de que o trabalho, à parte dos resultados, foi uma grande fonte de aprendizado e revisão de vários tópicos que são os pilares de toda a ciência da computação.

Sobre a parte de trabalhos futuros, podem-se deixar idéias como a de um complemento dos algoritmos, implementando-se novos algoritmos. Pode-se realizar uma intensa pesquisa para ver o que já foi feito para melhorar a performance de Java. Tem-se também a possibilidade de uma análise detalhada da diferença entre as instruções antes e pós-otimização, e também uma análise da diferença dos tempos de execução.

Referências Bibliográficas

[AHO] AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores: princípios, técnicas e ferramentas.** Rio de Janeiro: LTC, c1995. 344p.

[AHO2] AHO, Alfred V. Principles of compiler design. Reading: Addison-Weslwy, 1979.

[API] Java™ 2 Platform, Standard Edition, v 1.4.2 **API Specification.** Disponível em: <http://www.java.sun.com>. Acesso em: 13 set. 2004.

[BAR] BARRET, Willians A. **Compiler construction: theory and practice.** Chicago: Science Research Associates, 1979.

[BAU] BAUER, F. L. **Compiler construction: an advanced course.** New York: Springer-Verlag, 1976.

[BEN] BENNETT, J. P. (Jeremy Peter). **Introduction to compiling techniques : a first course using ANSI C, LEX and YACC.** London: McGraw-Hill, c1990. 242p.

[COU] COURANT COMPUTER SCIENCE SYMPOSIUM, 5., New York, Mar. 29-30, 1971. **Design and optimization of compliers***.* Englewood Cliffs: Prentice-Hall, 1972.

[DEI] DEITEL, Harvey M.; DEITEL, Paul J. **Java como programar.** 3. ed. Porto Alegre: Bookman, 2001. 1201p.

[FRE] freeDicionary.org; Disponível em: <http://www.freedictionary.org/>. Acesso em: 14 out. 2004

[GOS] GOSLING, James; JOY, Bill; stelle, Guy; Bracha, Gilad. **The Java Language Specification**. 2 ed. Disponível em: <http://www.java.sun.com>. Acesso em: 22 fev. 2004.

[GRI] GRIES, David. **Compiler construction for digital computer.** New York: J. Wiley, 1971.

[HEN] HENNESSY, John L.; PATTERSON, David A; LARUS, James R. **Organização e projeto de computadores: a interface hardware/sofware**. 2. ed. Rio de Janeiro: LTC, c2000. 551p.

[LIN] LINDHOLM, Tim; YELLIN, Frank. **The Java™ Virtual Machine Specification**. 2. ed. Disponível em: <http://www.java.sun.com/>. Acesso em: 31 jan. 2003.

[MAK] MAK, Ronald. **Writing compilers and interpreters: an applied approach.** New York: J. Wiley, c1991. 516p.

[OLI] FURTADO, OLINTO J. V., **Apostila de aula da disciplina de Linguagens formais e compiladores. Disponível** em: <http://www.inf.ufsc.br/~olinto/>. Acesso em: 03 out. 2004.

[RAB] RABUSKE, Márcia Aguiar. **Introdução a teoria dos grafos.** Florianópolis: Ed. da UFSC, 1992. 173p.

[TAN] TANENBAUM, Andrew S.; WOODHULL, Albert S**. Sistemas Operacionais: Projeto e Implementação.** 2.ed. São Paulo: Bookman, 2002. 800p.

[TRE] TREMBLAY, Jean-Paul; SORENSON, P. G. **The theory and practice of compiler writing.** New York: McGraw-Hill, c1985. 796p.

Anexo A

Tabela de Instruções da JVM.

mnemônico	opcode (dec)	opcode (hex)	GRUPO
nop	0	0x00	
aconst_null	1	0x01	Load/Store
iconst_m1	2	0x02	Load/Store
iconst_0	3	0x03	Load/Store
iconst_1	4	0x04	Load/Store
iconst_2	5	0x05	Load/Store
iconst_3	6	0x06	Load/Store
iconst_4	7	0x07	Load/Store
iconst_5	8	0x08	Load/Store
lconst_0	9	0x09	Load/Store
lconst_1	10	0x0a	Load/Store
fconst_0	11	0x0b	Load/Store
fconst_1	12	0x0c	Load/Store
fconst_2	13	0x0d	Load/Store
dconst_0	14	0x0e	Load/Store
dconst_1	15	0x0f	Load/Store
bipush	16	0x10	Load/Store
sipush	17	0x11	Load/Store
ldc	18	0x12	Load/Store
ldc_w	19	0x13	Load/Store
ldc2_w	20	0x14	Load/Store
iload	21	0x15	Load/Store
lload	22	0x16	Load/Store
fload	23	0x17	Load/Store
dload	24	0x18	Load/Store
aload	25	0x19	Load/Store
iload_0	26	0x1a	Load/Store
iload_1	27	0x1b	Load/Store
iload_2	28	0x1c	Load/Store
iload_3	29	0x1d	Load/Store
lload_0	30	0x1e	Load/Store
lload_1	31	0x1f	Load/Store
lload_2	32	0x20	Load/Store
lload_3	33	0x21	Load/Store
fload_0	34	0x22	Load/Store
fload_1	35	0x23	Load/Store
fload_2	36	0x24	Load/Store
fload_3	37	0x25	Load/Store
dload_0	38	0x26	Load/Store
dload_1	39	0x27	Load/Store
dload_2	40	0x28	Load/Store

dload_3	41	0x29	Load/Store
aload_0	42	0x2a	Load/Store
aload_1	43	0x2b	Load/Store
aload_2	44	0x2c	Load/Store
aload_3	45	0x2d	Load/Store
iaload	46	0x2e	Load/Store
laload	47	0x2f	Load/Store
faload	48	0x30	Load/Store
daload	49	0x31	Load/Store
aaload	50	0x32	Load/Store
baload	51	0x33	Load/Store
caload	52	0x34	Load/Store
saload	53	0x35	Load/Store
istore	54	0x36	Load/Store
lstore	55	0x37	Load/Store
fstore	56	0x38	Load/Store
dstore	57	0x39	Load/Store
astore	58	0x3a	Load/Store
istore_0	59	0x3b	Load/Store
istore_1	60	0x3c	Load/Store
istore_2	61	0x3d	Load/Store
istore_3	62	0x3e	Load/Store
lstore_0	63	0x3f	Load/Store
lstore_1	64	0x40	Load/Store
lstore_2	65	0x41	Load/Store
lstore_3	66	0x42	Load/Store
fstore_0	67	0x43	Load/Store
fstore_1	68	0x44	Load/Store
fstore_2	69	0x45	Load/Store
fstore_3	70	0x46	Load/Store
dstore_0	71	0x47	Load/Store
dstore_1	72	0x48	Load/Store
dstore_2	73	0x49	Load/Store
dstore_3	74	0x4a	Load/Store
astore_0	75	0x4b	Load/Store
astore_1	76	0x4c	Load/Store
astore_2	77	0x4d	Load/Store
astore_3	78	0x4e	Load/Store
iastore	79	0x4f	Load/Store
lastore	80	0x50	Load/Store
fastore	81	0x51	Load/Store
dastore	82	0x52	Load/Store
aastore	83	0x53	Load/Store
bastore	84	0x54	Load/Store
castore	85	0x55	Load/Store
sastore	86	0x56	Load/Store
pop	87	0x57	Gerenciamento da pilha de operandos
pop2	88	0x58	Gerenciamento da pilha de operandos
dup	89	0x59	Gerenciamento da pilha de operandos
dup_x1	90	0x5a	Gerenciamento da pilha de operandos
dup_x2	91	0x5b	Gerenciamento da pilha de operandos

dup2	92	0x5c	Gerenciamento da pilha de operandos
dup2_x1	93	0x5d	Gerenciamento da pilha de operandos
dup2_x2	94	0x5e	Gerenciamento da pilha de operandos
swap	95	0x5f	Gerenciamento da pilha de operandos
iadd	96	0x60	Aritméticas
ladd	97	0x61	Aritméticas
fadd	98	0x62	Aritméticas
dadd	99	0x63	Aritméticas
isub	100	0x64	Aritméticas
lsub	101	0x65	Aritméticas
fsub	102	0x66	Aritméticas
dsub	103	0x67	Aritméticas
imul	104	0x68	Aritméticas
lmul	105	0x69	Aritméticas
fmul	106	0x6a	Aritméticas
dmul	107	0x6b	Aritméticas
idiv	108	0x6c	Aritméticas
ldiv	109	0x6d	Aritméticas
fdiv	110	0x6e	Aritméticas
ddiv	111	0x6f	Aritméticas
irem	112	0x70	Aritméticas
lrem	113	0x71	Aritméticas
frem	114	0x72	Aritméticas
drem	115	0x73	Aritméticas
.......*ineg*	116	0x74	Aritméticas
lneg	117	0x75	Aritméticas
fneg	118	0x76	Aritméticas
dneg	119	0x77	Aritméticas
ishl	120	0x78	Aritméticas
lshl	121	0x79	Aritméticas
ishr	122	0x7a	Aritméticas
lshr	123	0x7b	Aritméticas
iushr	124	0x7c	Aritméticas
lushr	125	0x7d	Aritméticas
iand	126	0x7e	Aritméticas
land	127	0x7f	Aritméticas
ior	128	0x80	Aritméticas
lor	129	0x81	Aritméticas
ixor	130	0x82	Aritméticas
lxor	131	0x83	Aritméticas
iinc	132	0x84	Aritméticas
i2l	133	0x85	Conversão de tipos
i2f	134	0x86	Conversão de tipos
i2d	135	0x87	Conversão de tipos
l2i	136	0x88	Conversão de tipos
l2f	137	0x89	Conversão de tipos
l2d	138	0x8a	Conversão de tipos
f2i	139	0x8b	Conversão de tipos
f2l	140	0x8c	Conversão de tipos
f2d	141	0x8d	Conversão de tipos
d2i	142	0x8e	Conversão de tipos

d2l	143	0x8f	Conversão de tipos
d2f	144	0x90	Conversão de tipos
i2b	145	0x91	Conversão de tipos
i2c	146	0x92	Conversão de tipos
i2s	147	0x93	Conversão de tipos
lcmp	148	0x94	Aritméticas
fcmpl	149	0x95	Aritméticas
fcmpg	150	0x96	Aritméticas
dcmpl	151	0x97	Aritméticas
dcmpg	152	0x98	Aritméticas
ifeq	153	0x99	Transferência de controle
ifne	154	0x9a	Transferência de controle
iflt	155	0x9b	Transferência de controle
ifge	156	0x9c	Transferência de controle
ifgt	157	0x9d	Transferência de controle
ifle	158	0x9e	Transferência de controle
if_icmpeq	159	0x9f	Transferência de controle
if_icmpne	160	0xa0	Transferência de controle
if_icmplt	161	0xa1	Transferência de controle
if_icmpge	162	0xa2	Transferência de controle
if_icmpgt	163	0xa3	Transferência de controle
if_icmple	164	0xa4	Transferência de controle
if_acmpeq	165	0xa5	Transferência de controle
if_acmpne	166	0xa6	Transferência de controle
goto	167	0xa7	Transferência de controle
jsr	168	0xa8	Transferência de controle
ret	169	0xa9	Transferência de controle
tableswitch	170	0xaa	Transferência de controle
lookupswitch	171	0xab	Transferência de controle
ireturn	172	0xac	Invocação e retorno de métodos
lreturn	173	0xad	Invocação e retorno de métodos
freturn	174	0xae	Invocação e retorno de métodos
dreturn	175	0xaf	Invocação e retorno de métodos
areturn	176	0xb0	Invocação e retorno de métodos
return	177	0xb1	Invocação e retorno de métodos
getstatic	178	0xb2	Invocação e retorno de métodos
putstatic	179	0xb3	Criação e manipulação de objetos
getfield	180	0xb4	Criação e manipulação de objetos
putfield	181	0xb5	Criação e manipulação de objetos
invokevirtual	182	0xb5	Invocação e retorno de métodos
invokespecial	183	0xb7	Invocação e retorno de métodos
invokestatic	184	0xb8	Invocação e retorno de métodos
invokeinterface	185	0xb9	Invocação e retorno de métodos
xxxunusedxxx1	186	0xba	
new	187	0xbb	Criação e manipulação de objetos
newarray	188	0xbc	Criação e manipulação de objetos
anewarray	189	0xbd	Criação e manipulação de objetos
arraylength	190	0xbe	Criação e manipulação de objetos
athrow	191	0xbf	Exceções
checkcast	192	0xc0	Criação e manipulação de objetos
instanceof	193	0xc1	Criação e manipulação de objetos

monitorenter	194	0xc2	Sincronização
monitorexit	195	0xc3	Sincronização
wide	196	0xc4	Load/Store
multianewarray	197	0xc5	Criação e manipulação de objetos
ifnull	198	0xc6	Transferência de controle
ifnonnull	199	0xc7	Transferência de controle
goto_w	200	0xc8	Transferência de controle
jsr_w	201	0xc9	Transferência de controle
breakpoint	202	0xca	
impdep1	254	0xfe	
impdep2	255	0xff	

Anexo B

Código-fonte do Programa de Leitura do *Bytecode*.

```
//ARQUIVO LERBC.JAVA
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class LerBC extends JFrame {
	private DataInputStream arq;
	private JTextArea jta;

	public LerBC() {
		super("LEITOR DE BYTECODE");

		//CONFIGURA JANELA AO FECHAR
		this.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);

		this.addWindowListener(
			new WindowAdapter() {
				public void windowClosing(WindowEvent e) {
					LerBC.this.jta.setText("");
					LerBC.this.close();
					int resultado = JOptionPane.showConfirmDialog(
						LerBC.this, "Deseja ler outro arquivo .class",
						"Ler                    novamente?",
JOptionPane.YES_NO_OPTION);
					if(resultado==JOptionPane.YES_OPTION)
						LerBC.this.open();
					else
						System.exit(0);
```

```
                }
            }
        );

        //CONFIGURA LAYOUT
        Container c = getContentPane();
        jta = new JTextArea();
        jta.setEditable(false);
        c.add(new JScrollPane(jta));
        this.open();
        setSize(350, 500);
        show();
    }

    private void open() {
        try {
            //CONFIGURA O FILECHOOSER
            JFileChooser chooser = new JFileChooser();
            chooser.removeChoosableFileFilter(
                                        chooser.getFileFilter());
            chooser.setFileFilter(
                new javax.swing.filechooser.FileFilter() {
                    public boolean accept(File f) {
                        if(f.getPath().endsWith(".class")
                                || f.isDirectory() )
                            return true;
                        else
                            return false;
                    }

                    public String getDescription() {
                        return "Arquivos .class";
                    }
```

```
                    }
                );
                chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
                int returnVal = chooser.showOpenDialog(this);

                //ABRE O ARQUIVO SE USUÁRIO ESCOLHER SENÃO SAI
                if(returnVal == JFileChooser.CANCEL_OPTION)
                    System.exit(0);
                File file = chooser.getSelectedFile();
                arq = new DataInputStream( new FileInputStream(file));

                //CHAMA MÉTODO PARA LER BYTECODE
                this.imprimaByteCode(file.getName());
            }
            catch(Exception e) {
                System.err.println("Nao foi possivel abrir o "
                                            + "arquivo especificado!");
                System.exit(1);
            }
    }

    private void interface_table() {
            try {
                int count = arq.readUnsignedShort();
                jta.append("\ninterfaces_count: " + count);
                jta.append("\ninterfaces[]:\n");
                for(int i = 1; i <= count; i++)
                    jta.append("\tinterface " + i + ": " +
                        arq.readUnsignedShort() + "\n");
            }
            catch(Exception e) {
                jta.append("\nErro ao ler arquivo!");
            }
```

```
}

private void methods_table() {
        try {
                int count = arq.readUnsignedShort();
                jta.append("\nmethods_count: " + count);
                jta.append("\nmethods[]:");
                for(int i = 1; i <= count; i++) {
                        jta.append("\n  method "+ i + ":");
                        jta.append("\taccess_flags: 0x");
                        this.ler(2, true);
                        jta.append("\n\tname_index: ");
                        this.ler(2, false);
                        jta.append("\n\tdescriptor_index: ");
                        this.ler(2, false);
                        jta.append("\n\tattributes_count: ");

                        int at_count = arq.readUnsignedShort();
                        jta.append(at_count + "\n\tattributtes[]: ");
                        for(int j = 1; j <= at_count; j++) {
                                jta.append("\n\t\tatributte " + j + ": ");
                                jta.append("\n\t\tattribute_name_index: ");
                                this.ler(2, false);
                                int at_length = arq.readInt();
                                jta.append("\n\t\tattribute_length: "
                                                                                +
at_length);
                                jta.append("\n\t\tinfo[]: ");

                                for(int k = 1; k <= at_length; k++) {
                                        this.ler(1, true);
                                }
```

```
                }
                jta.append("\n");
            }
        }
        catch(Exception e) {
            jta.append("\nErro ao ler arquivo!");
        }
    }

    private void attributes_table() {
        try {
            int at_count = arq.readUnsignedShort();
            jta.append("\nattributes_count: " + at_count);
            for(int i = 1; i <= at_count; i++) {
                jta.append("\nattribute_name_index: ");
                this.ler(2, false);
                int at_length = arq.readInt();
                jta.append("\nattribute length: " + at_length);
                jta.append("\ninfo[]: ");
                for(int j = 1; j <= at_length; j++) {
                    this.ler(1, true);
                }
            }
            jta.append("\n");
        }
        catch(Exception e) {
            jta.append("\nErro ao ler arquivo!");
        }
    }

    private void fields_table() {
        try {
            int count = arq.readUnsignedShort();
```

```
                jta.append("\nfields_count: " + count);
                jta.append("\nfields[]:");
                for(int i = 0; i < count; i++) {
                    jta.append("\n  indice "+ i + ":");
                    jta.append("\taccess_flags: 0x");
                    this.ler(2, true);
                    jta.append("\n\tname_index: ");
                    this.ler(2, false);
                    jta.append("\n\tdescriptor_index: ");
                    this.ler(2, false);
                    jta.append("\n\tattributes_count: ");

                    int at_count = arq.readUnsignedShort();
                    jta.append(at_count + "\n\tattributtes[]: ");
                    for(int j = 1; j <= at_count; j++) {
                        jta.append("\n\t\tindice " + j + ": ");
                        jta.append("\n\t\tattribute_name_index: ");
                        this.ler(2, false);
                        int at_length = arq.readInt();
                        jta.append("\n\t\tattribute_length: "
                                                                                +
at_length);
                        jta.append("\n\t\tinfo[]: ");

                        for(int k = 1; k <= at_length; k++) {
                            this.ler(1, true);
                        }

                    }
                    jta.append("\n");
                }
            }
            catch(Exception e) {
```

```
            jta.append("\nErro ao ler arquivo!");
        }
}

private void version() {
        try{
            //minor_version
            int m = arq.readUnsignedShort();
            //major_version
            int M = arq.readUnsignedShort();
            jta.append(M+"."+m);
        }
        catch(Exception e) {
            jta.append("\nErro ao ler arquivo!");
        }
}

private void cp_table() {
        try {
            int count = arq.readUnsignedShort();
            jta.append("\nconstant_pool_count: " + count);
            jta.append("\nconstant_pool[]:\n");
            for(int i = 1; i < count; i++) {
                int tag = arq.readUnsignedByte();
                jta.append("  indice "+ i + ":");
                jta.append("\ttag:" + tag + " (");
                switch(tag) {
                    case 1: //CONSTANT_Utf8
                        jta.append("CONSTANT_utf8)\n\t");
                        jta.append("string: " + arq.readUTF(arq) );
                        break;
                    case 3: //CONSTANT_Integer
                        jta.append("CONSTANT_Integer)\n\t");
```

```
            jta.append("int: " + arq.readInt() );
            break;
    case 4: //CONSTANT_Float
            jta.append("CONSTANT_Float)\n\t");
            jta.append("float: " + arq.readFloat() );
            break;
    case 5: //CONSTANT_Long
            jta.append("CONSTANT_Long)\n\t");
            jta.append("long: " + arq.readLong() );
            ++i;
            break;
    case 6: //CONSTANT_Double
            jta.append("CONSTANT_Double)\n\t");
            jta.append("double: " + arq.readDouble() );
            ++i;
            break;
    case 7: //CONSTANT_Class
            jta.append("CONSTANT_Class)\n\t");
            jta.append("name_index: ");
            ler(2, false);
            break;
    case 8: //CONSTANT_String
            jta.append("CONSTANT_String)\n\t");
            jta.append("string_index: ");
            ler(2, false);
            break;
    case 9: //CONSTANT_Fieldref
            jta.append("CONSTANT_Fieldref)\n\t");
            jta.append("class_index: ");
            ler(2, false);
            jta.append("\n\tname_and_type_index: ");
            ler(2, false);
            break;
```

```
                    case 10: //CONSTANT_Methodref
                        jta.append("CONSTANT_Methodref)\n\t");
                        jta.append("class_index: ");
                        ler(2, false);
                        jta.append("\n\tname_and_type_index: ");
                        ler(2, false);
                        break;
                    case 11: //CONSTANT_InterfaceMethodref
                        jta.append("CONSTANT_InterfaceMethodref)\n\t");
                        jta.append("class_index: ");
                        ler(2, false);
                        jta.append("\n\tname_and_type_index: ");
                        ler(2, false);
                        break;
                    case 12: //CONSTANT_NameAndType
                        jta.append("CONSTANT_NameAndType)\n\t");
                        jta.append("name_index: ");
                        ler(2, false);
                        jta.append("\n\tdescriptor_index: ");
                        ler(2, false);
                        break;
                }
                jta.append("\n");
            }
        }
        catch(Exception e) {
            jta.append("\nErro ao ler arquivo!");
        }
    }

    private void close() {
        try {
            arq.close();
```

```
		}
		catch(Exception e) {
			System.exit(1);
		}
	}

	private void ler(int num, boolean hex) {
		try {
			switch(num) {
				case 1:
					if (hex) jta.append(Integer.toHexString(

	arq.readUnsignedByte()));
					else jta.append(""+arq.readUnsignedByte());

					break;
				case 2:
					if(hex) jta.append(""+Integer.toHexString(

	arq.readUnsignedShort()));
					else jta.append(""+arq.readUnsignedShort());
					break;
				case 4:
					if(hex) jta.append(""+Integer.toHexString(

	arq.readInt()));
					else jta.append(""+arq.readInt());
					break;
			}
		}
		catch(Exception e) {
			jta.append("\nErro ao ler arquivo!");
		}
```

```
	}


/****************************************************
*	ESTRUTURA DO ARQUIVO CLASS
*
* ClassFile {
*
*	u4 magic;
*	u2 minor_version;
*	u2 major_version;
*	u2 constant_pool_count;
*	cp_info constant_pool[constant_pool_count-1];
*	u2 access_flags;
*	u2 this_class;
*	u2 super_class;
*	u2 interfaces_count;
*	u2 interfaces[interfaces_count];
*	u2 fields_count;
*	field_info fields[fields_count];
*	u2 methods_count;
*	method_info methods[methods_count];
*	u2 attributes_count;
*	attribute_info attributes[attributes_count];
*  }
*
****************************************************/
	public void imprimaByteCode(String arquivo) {
		jta.append("Arquivo: "
					+ arquivo + "\n");

		jta.append("\nmagic: 0x");
		this.ler(4, true);
```

```
			jta.append("\nversion: ");
			this.version();
			this.cp_table();
			jta.append("\naccess_flags: 0x");
			this.ler(2, true);
			jta.append("\nthis_class: ");
			this.ler(2, false);
			jta.append("\nsuper_class: ");
			this.ler(2, false);
			this.interface_table();
			this.fields_table();
			this.methods_table();
			this.attributes_table();
		}

		public static void main(String args[]) {
			LerBC leitor = new LerBC();
		}
}
```

Anexo C

Código-fonte do Programa de Otimização.

```
package inter;

import java.io.*;
import javax.swing.*;

public class EscolheArquivo
    extends JFileChooser {

  private DataInputStream arq;

  public EscolheArquivo(JanelaPrincipal jp) {
    try {

      JFileChooser chooser = new JFileChooser();
      this.removeChoosableFileFilter(
          chooser.getFileFilter());
      this.setFileFilter(
          new javax.swing.filechooser.FileFilter() {
        public boolean accept(File f) {
          if (f.getPath().endsWith(".class")
              || f.isDirectory())
            return true;
          else
            return false;
        }

        public String getDescription() {
          return "Arquivos .class";
```

```
      }
     }
     );
     this.setFileSelectionMode(JFileChooser.FILES_ONLY);
     int returnVal = this.showOpenDialog(jp);

     //ABRE O ARQUIVO SE USUÁRIO ESCOLHER SENÃO SAI
     //    if (returnVal == JFileChooser.CANCEL_OPTION)
     //     System.exit(0);
     File file = this.getSelectedFile();
     arq = new DataInputStream(new FileInputStream(file));
    }
    catch (Exception e) {
     System.err.println("Nao foi possivel abrir o "
                  + "arquivo especificado!");
     System.exit(1);
    }
   }

  public DataInputStream getArquivo() {
    return arq;
   }
}

package inter;

import java.io.*;
import javax.swing.*;

public class EscolheArquivo2
    extends JFileChooser {

  DataOutputStream arq;
```

```
  public EscolheArquivo2() {
    try {
      JFileChooser chooser = new JFileChooser();

      this.removeChoosableFileFilter(
          chooser.getFileFilter());
      chooser.setFileFilter(
          new javax.swing.filechooser.FileFilter() {
        public boolean accept(File f) {
          if (f.getPath().endsWith(".class")
              || f.isDirectory())
            return true;
          else
            return false;
        }

        public String getDescription() {
          return "Arquivos .class";
        }
      }
      );
      this.setFileSelectionMode(JFileChooser.FILES_ONLY);
      int returnVal = chooser.showOpenDialog(this);

      //ABRE O ARQUIVO SE USUÁRIO ESCOLHER SENÃO SAI
//      if (returnVal == JFileChooser.CANCEL_OPTION)
//        System.exit(0);
      File file = chooser.getSelectedFile();
      arq = new DataOutputStream(new FileOutputStream(file));
    }

    catch (Exception e) {
      System.err.println("Nao foi possivel abrir o "
```

```
                  + "arquivo especificado!");
      System.exit(1);
    }
  }

  public DataOutputStream getArquivo() {
    return arq;
  }

}

package inter;

import javax.swing.*;
import java.awt.*;
import com.borland.jbcl.layout.*;
import java.awt.event.*;
import java.io.*;
import outros.*;

public class EscolheOtimizacao
    extends JFrame {
  XYLayout xYLayout1 = new XYLayout();
  JButton jButton1 = new JButton();
  JCheckBox jCheckBox3 = new JCheckBox();
  JCheckBox jCheckBox1 = new JCheckBox();
  JCheckBox jCheckBox2 = new JCheckBox();
  JCheckBox jCheckBox5 = new JCheckBox();
  JCheckBox jCheckBox4 = new JCheckBox();
  JLabel jLabel1 = new JLabel();
  DataInputStream arqLer;
  DataOutputStream arqEscrever;
  public EscolheOtimizacao(DataInputStream arq1, DataOutputStream arq2) {
```

```
  try {
    arqLer = arq1;
    arqEscrever = arq2;
    jbInit();
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}

private void jbInit() throws Exception {
  jButton1.setText("Otimizar");
  jButton1.addActionListener(new EscolheOtimizacao_jButton1_actionAdapter(this));
  this.getContentPane().setLayout(xYLayout1);
  jCheckBox3.setSelected(true);
  jCheckBox3.setText("Sub-expressões comuns");
  jCheckBox1.setSelected(true);
  jCheckBox1.setText("Peephole");
  jCheckBox2.setSelected(true);
  jCheckBox2.setText("Propagação de cópias");
  jCheckBox5.setSelected(true);
  jCheckBox5.setText("Variáveis de indução");
  jCheckBox4.setSelected(true);
  jCheckBox4.setText("Movimentação de código");
  xYLayout1.setWidth(204);
  xYLayout1.setHeight(272);
  this.setState(Frame.NORMAL);
  this.setTitle("Otimização de Bytecode");
  jLabel1.setEnabled(true);
  jLabel1.setFont(new java.awt.Font("SansSerif", 1, 13));
  jLabel1.setText("Escolha as otimizações");
  this.getContentPane().add(jCheckBox1, new XYConstraints(31, 40, -1, -1));
  this.getContentPane().add(jCheckBox2, new XYConstraints(31, 75, -1, -1));
```

```
    this.getContentPane().add(jLabel1, new XYConstraints(20, 9, 164, 24));
    this.getContentPane().add(jCheckBox3, new XYConstraints(31, 108, -1, -1));
    this.getContentPane().add(jCheckBox4, new XYConstraints(31, 142, -1, -1));
    this.getContentPane().add(jCheckBox5, new XYConstraints(31, 177, -1, -1));
    this.getContentPane().add(jButton1, new XYConstraints(46, 222, 112, 33));

    this.show();
  }

  void jButton1_actionPerformed(ActionEvent e) {
    boolean qualOtimizacao[] = new boolean[5];
    for (int i = 0; i < qualOtimizacao.length; i++)
      qualOtimizacao[i] = false;

    if (jCheckBox1.isSelected())
      qualOtimizacao[0] = true;
    if (jCheckBox2.isSelected())
      qualOtimizacao[1] = true;
    if (jCheckBox3.isSelected())
      qualOtimizacao[2] = true;
    if (jCheckBox4.isSelected())
      qualOtimizacao[3] = true;
    if (jCheckBox5.isSelected())
      qualOtimizacao[4] = true;

    Otimizacao o = new Otimizacao(arqLer, arqEscrever, qualOtimizacao);
    o.otimiza();
  }

}

class EscolheOtimizacao_jButton1_actionAdapter
    implements java.awt.event.ActionListener {
```

```
  EscolheOtimizacao adaptee;

  EscolheOtimizacao_jButton1_actionAdapter(EscolheOtimizacao adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.jButton1_actionPerformed(e);
  }
}

package outros;

//ARQUIVO ESCRITOR.JAVA
import java.io.*;
import java.util.*;

public class Escritor {

  private Vector byteCode;
  private Vector codes;
  private DataOutputStream arquivo;

  public Escritor(Vector bc, Vector c, DataOutputStream arq) {
    arquivo = arq;
    byteCode = bc;
    codes = c;
  }

  public void escreverBC() {
    try {
      int i = 0;
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue()); //0xCAFEBABE
```

```
arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //minor_version
arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //major_version

int cp_pool_count = ( (Integer) (byteCode.get(i++))).intValue();
arquivo.writeShort(cp_pool_count);
for (int k = 1; k < cp_pool_count; k++) {
  int tag = ( (Integer) (byteCode.get(i++))).intValue();
  arquivo.writeByte(tag);
  switch (tag) {
    case 1: //CONSTANT_Utf8
      arquivo.writeUTF( (String) byteCode.get(i++));
      break;
    case 3: //CONSTANT_Integer
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue());
      break;
    case 4: //CONSTANT_Float
      arquivo.writeFloat( ( (Float) (byteCode.get(i++))).floatValue());
      break;
    case 5: //CONSTANT_Long
      arquivo.writeLong( ( (Long) (byteCode.get(i++))).longValue());
      break;
    case 6: //CONSTANT_Double
      arquivo.writeDouble( ( (Double) (byteCode.get(i++))).doubleValue());
      break;
    case 7: //CONSTANT_Class
      arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());
      break;
    case 8: //CONSTANT_String
      arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());
      break;
    case 9: //CONSTANT_Fieldref
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue());
      break;
```

```
    case 10: //CONSTANT_Methodref
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue());
      break;
    case 11: //CONSTANT_InterfaceMethodref
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue());
      break;
    case 12: //CONSTANT_NameAndType
      arquivo.writeInt( ( (Integer) (byteCode.get(i++))).intValue());
      break;
  }
}

arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //access_flags
arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //this_class
arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //super_class

int interfaces_count = ( (Integer) byteCode.get(i++)).intValue();
arquivo.writeShort(interfaces_count);
for (int k = 1; k <= interfaces_count; k++)
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());

int fields_count = ( (Integer) byteCode.get(i++)).intValue();
arquivo.writeShort(fields_count);
for (int k = 0; k < fields_count; k++) {
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());

  int attributes_count = ( (Integer) byteCode.get(i++)).intValue();
  arquivo.writeShort(attributes_count);
  for (int j = 1; j <= attributes_count; j++) {
    arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());
    int attribute_length = ( (Integer) byteCode.get(i++)).intValue();
```

```
    arquivo.writeInt(attribute_length);
    for (int v = 1; v <= attribute_length; v++) {
      arquivo.writeByte( ( (Integer) (byteCode.get(i++))).intValue());
    }
  }
}

int methods_count = ( (Integer) byteCode.get(i++)).intValue();
arquivo.writeShort(methods_count);
for (int v = 1; v <= methods_count; v++) {
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //access_flags
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //name_index
  arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue()); //descriptor_index

  int attributes_count = ( (Integer) byteCode.get(i++)).intValue();
  arquivo.writeShort(attributes_count);
  for (int j = 1; j <= attributes_count; j++) {
    //byteCode.add(arquivo.readUnsignedShort());//access_flags
    int attribute_name_index = ( (Integer) byteCode.get(i++)).intValue();
    arquivo.writeShort(attribute_name_index);

    boolean ehCode = false;
    if (codes.indexOf(new Integer(attribute_name_index)) != -1)
      ehCode = true; //guarda as instruções

    if (ehCode) {
      int attribute_length = ( (Integer) byteCode.get(i++)).intValue();
      arquivo.writeInt(attribute_length);
      Vector codigo = ( (Vector) byteCode.get(i++));
      for (int k = 1; k <= attribute_length; k++) {
        arquivo.writeByte( ( (Integer) (codigo.get(k - 1))).intValue());
      }
    }
```

```
          else {
            int attribute_length = ( (Integer) byteCode.get(i++)).intValue();
            arquivo.writeInt(attribute_length);
            for (int k = 1; k <= attribute_length; k++) {
              arquivo.writeByte( ( (Integer) (byteCode.get(i++))).intValue());
            }
          }
        }
      }

      int attributes_count = ( (Integer) byteCode.get(i++)).intValue();
      arquivo.writeShort(attributes_count);
      for (int k = 1; k <= attributes_count; k++) {
        arquivo.writeShort( ( (Integer) (byteCode.get(i++))).intValue());

        int attribute_length = ( (Integer) byteCode.get(i++)).intValue();
        arquivo.writeInt(attribute_length);
        for (int j = 1; j <= attribute_length; j++) {
          arquivo.writeByte( ( (Integer) (byteCode.get(i++))).intValue());
        }
      }

      arquivo.flush();
      arquivo.close();
    }
    catch (Exception e) {
      System.err.println("Erro ao escrever no arquivo!");
    }
  }
}

package inter;
```

```
import javax.swing.*;
import java.awt.*;
import com.borland.jbcl.layout.*;
import java.util.*;
import java.io.*;
import outros.*;

public class ExibirGrafo
    extends JFrame {
  XYLayout xYLayout1 = new XYLayout();
  JLabel jLabel1 = new JLabel();
  JScrollPane jScrollPane1 = new JScrollPane();
  JTextArea jTextArea1 = new JTextArea();
  DataInputStream arquivo;
  public ExibirGrafo(DataInputStream dos) {
    try {
      arquivo = dos;
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    jLabel1.setFont(new java.awt.Font("Dialog", 1, 13));
    jLabel1.setToolTipText("");
    jLabel1.setText("GRAFOS DE FLUXO");
    this.getContentPane().setLayout(xYLayout1);
    jTextArea1.setTabSize(8);
    xYLayout1.setWidth(421);
    xYLayout1.setHeight(501);
    this.setTitle("Otimização de Bytecode");
```

```
    this.getContentPane().add(jLabel1, new XYConstraints(140, 7, 139, 39));
    this.getContentPane().add(jScrollPane1, new XYConstraints(0, 44, 420, 455));
    jScrollPane1.getViewport().add(jTextArea1, null);

    this.setSize(421, 501);
    this.show();

    this.imprimirGrafo();
  }

  public void imprimirGrafo() {

    /* int i = 1;
      //loop metodos
      jTextArea1.append("Método " + i + ":
0403000332ab401b803124b6052ab401b8062ab4011271283b8093d1c9a0d2ab401b80aa7073b80
bb100010130001e07000110c01201301302001602401702e01903201a\n\n");

      //loop blocos
      jTextArea1.append("Bloco 1: 0403000332ab401b803124b6052ab401\n");
      jTextArea1.append("Bloco 2: b8062ab4011271283b8093d1c9a0d2ab401b8\n");
     jTextArea1.append("Bloco 3: aa7073b80bb100010130001e07000110c012013013\n");
      jTextArea1.append("Bloco 4: 02001602401702e01903201a\n");
      //fim blocos

      jTextArea1.append("\n\n      XXXXXX\t|      Entrada\t|      Saída\n");

      //loop tabela
     jTextArea1.append("------------------------------------------------------"
                + "--------------------------------------\n");
      jTextArea1.append("     Bloco 1\t|");
      jTextArea1.append("   B4\t|");
      jTextArea1.append("   B2, B3\n");
```

```
 jTextArea1.append("--------------------------------------------------"
              + "--------------------------------------\n");
   jTextArea1.append("     Bloco 2\t|");
   jTextArea1.append("   B1\t|");
   jTextArea1.append("   B4\n");
 jTextArea1.append("--------------------------------------------------"
              + "--------------------------------------\n");
   jTextArea1.append("     Bloco 3\t|");
   jTextArea1.append("   B1\t|");
   jTextArea1.append("   B4\n");
 jTextArea1.append("--------------------------------------------------"
              + "--------------------------------------\n");
   jTextArea1.append("     Bloco 4\t|");
   jTextArea1.append("   B2, B3\t|");
   jTextArea1.append("   B1\n");
   //fim loop tabela
   //fim metodos*/

String grafo = "";
Leitor leitor = new Leitor(arquivo);
Vector bc = leitor.getBC();
Vector instrucoes = leitor.getInstrucoes();

for (int i = 0; i < instrucoes.size(); i++) { //loop métodos
  //imprime metodo
  int indice = ( (Integer) (instrucoes.get(i))).intValue();
  Vector metodo = (Vector) (bc.get(indice));
  grafo += "Método " + (i + 1) + " ";
  for (int j = 0; j < metodo.size(); j++) {
    Integer instrucao = (Integer) metodo.get(j);
    grafo += instrucao.toHexString(instrucao.intValue());
  }
  grafo += "\n\n";
```

```
    //imprime blocos
    Separador separador = new Separador();
    Vector blocos = separador.separa(metodo);
    for (int j = 0; j < blocos.size(); j++) {
      grafo += "Bloco " + (j + 1) + ": ";
      Vector umBloco = (Vector) (blocos.get(j));
      for (int k = 0; k < umBloco.size(); k++) {
        Integer instrucao = ( (Integer) (umBloco.get(k)));
        grafo += instrucao.toHexString(instrucao.intValue());
      }
    }
    grafo += "\n\n      XXXXXX\tl      Entrada\tl      Saída\n";
    grafo += "     Bloco " + (i + 1) + "\tl";
    grafo += this.quaisEntradas(blocos, i) + "\tl";
    grafo += this.quaisSaidas(blocos, i) + "\n";
    grafo += "--------------------------------------------------"
        + "--------------------------------------\n";
  }
  jTextArea1.setText(grafo);
}

private String quaisEntradas(Vector blocos, int indice) {
  int instrucaoInicial = 0;
  for (int i = 0; i < indice; i++) {
    Vector umBloco = ( (Vector) blocos.get(i));
    instrucaoInicial += umBloco.size();
  }
  int instrucaoFinal = instrucaoInicial +
      ( (Vector) blocos.get(indice)).size();
  String entradas = "";
  for (int i = 0; i < blocos.size(); i++) {
    Vector umBloco = ( (Vector) blocos.get(i));
```

```
    for (int j = 0; j < umBloco.size(); j++) {
      int instrucao = ( (Integer) umBloco.get(j)).intValue();
      if (this.ehDesvio(instrucao)) {
        int alvo = this.calculaAlvo(umBloco, j);
        if (alvo >= instrucaoInicial && alvo <= instrucaoFinal) {
          entradas += "B" + (i + 1) + ", ";
        }
      }
    }
  }
  return entradas;
}

private String quaisSaidas(Vector blocos, int indice) {
  String saidas = "";
  Vector umBloco = ( (Vector) blocos.get(indice));
  for (int i = 0; i < umBloco.size(); i++) {
    int instrucao = ( (Integer) umBloco.get(i)).intValue();
    if (this.ehDesvio(instrucao)) {
      int qualBloco = 0;
      int alvo = this.calculaAlvo(umBloco, i);
      for (int j = 0; j < blocos.size(); j++) {
        Vector b = ( (Vector) blocos.get(j));
        qualBloco += (b.size() - 1);
        if (alvo >= qualBloco) {
          saidas += "B" + (j + 1) + ", ";
        }
      }
    }
  }
  return saidas;
}
```

```
private boolean ehDesvio(int instrucao) {
  //153 - 171
  //198 - 201
  if (instrucao >= 153 && instrucao <= 171)
    return true;

  if (instrucao >= 198 && instrucao <= 201)
    return true;

  return false;
}

//retorna o deslocamento(alvo) da instrucao de desvio
private int calculaAlvo(Vector instrucoes, int indice) {
  int instrucao = ( (Integer) instrucoes.get(indice)).intValue();
  int branchbyte1, branchbyte2, branchbyte3, branchbyte4;
  int branchoffset = -1;
  switch (instrucao) {
    case 153: //ifeq
    case 154: //iflt
    case 155: //ifge
    case 156: //ifgt
    case 157: //ifgt
    case 158: //ifle
    case 159: //if_icmpeq
    case 160: //if_icmpne
    case 161: //if_icmplt
    case 162: //if_icmpge
    case 163: //if_icmpgt
    case 164: //if_icmple
    case 165: //if_acmpeq
    case 166: //if_acmpne
    case 167: //goto
```

```
    case 168: //jsr
    case 198: //ifnull
    case 199: //ifnonnull
      branchbyte1 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchbyte2 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchoffset = (branchbyte1 << 8) | branchbyte2;
      break;
    case 200: //goto_w
    case 201: //jsr_w
      branchbyte1 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchbyte2 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchbyte3 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchbyte4 = ( (Integer) instrucoes.get(++indice)).intValue();
      branchoffset = (branchbyte1 << 24) | (branchbyte2 << 16) |
          (branchbyte3 << 8) | branchbyte4;
      break;
    case 169: //ret

      //calcular
      break;
    case 170: //tableswitch
    case 171: //lookupswitch

      //variavel
      break;
    default:
      break;

  }

  return branchoffset;
 }
}
```

```
package inter;

import javax.swing.*;
import java.awt.*;
import com.borland.jbcl.layout.*;
import java.awt.event.*;
import java.io.*;
import outros.LeitorBC;

public class JanelaPrincipal
    extends JFrame {
  private DataInputStream arqLer;
  private DataOutputStream arqEscrever;
  XYLayout xYLayout1 = new XYLayout();
  JButton jButton1 = new JButton();
  JButton jButton3 = new JButton();
  JButton jButton2 = new JButton();
  JLabel jLabel1 = new JLabel();
  JLabel jLabel2 = new JLabel();
  JButton jButton4 = new JButton();
  JScrollPane jScrollPane1 = new JScrollPane();
  JScrollPane jScrollPane2 = new JScrollPane();
  JTextArea jTextArea1 = new JTextArea();
  JTextArea jTextArea2 = new JTextArea();
  JButton jButton5 = new JButton();

  public JanelaPrincipal() {
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
```

```
  }
}

private void jbInit() throws Exception {
  jButton1.setText("Arquivo Origem");
  jButton1.addActionListener(new JanelaPrincipal_jButton1_actionAdapter(this));
  this.setLocale(java.util.Locale.getDefault());
  this.setState(Frame.NORMAL);
  this.setTitle("Otimização de Bytecode");
  this.getContentPane().setLayout(xYLayout1);
  jButton3.setEnabled(false);
  jButton3.setText("Otimização");
  jButton3.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      jButton3_actionPerformed(e);
    }
  });
  jButton2.setEnabled(false);
  jButton2.setText("Arquivo Destino");
  jButton2.addActionListener(new JanelaPrincipal_jButton2_actionAdapter(this));
  jLabel1.setFont(new java.awt.Font("Microsoft Sans Serif", 1, 13));
  jLabel1.setText("Bytecode Original");
  jLabel2.setFont(new java.awt.Font("Dialog", 1, 13));
  jLabel2.setText("Bytecode Otimizado");
  xYLayout1.setWidth(669);
  xYLayout1.setHeight(490);
  jButton4.setText("Grafo de Fluxo");
  jButton4.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
      jButton4_actionPerformed(e);
    }
  });
  jButton4.setEnabled(false);
```

```
    jButton4.setSelected(false);
    jTextArea1.setText("");
    jButton5.setText("Limpar Campos");
    jButton5.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jButton5_actionPerformed(e);
      }
    });
    this.getContentPane().add(jLabel1, new XYConstraints(81, 58, 146, 21));
    this.getContentPane().add(jLabel2, new XYConstraints(460, 58, 132, 21));
    this.getContentPane().add(jButton1, new XYConstraints(75, 10, 122, 28));
    this.getContentPane().add(jButton4, new XYConstraints(459, 10, 122, 28));
    this.getContentPane().add(jButton2, new XYConstraints(203, 10, 122, 28));
    this.getContentPane().add(jButton3, new XYConstraints(331, 10, 122, 28));
    this.getContentPane().add(jScrollPane1, new XYConstraints(0, 91, 332, 399));
    jScrollPane1.getViewport().add(jTextArea1, null);
    this.getContentPane().add(jScrollPane2, new XYConstraints(337, 91, 332, 399));
    this.getContentPane().add(jButton5, new XYConstraints(274, 48, 124, 31));
    jScrollPane2.getViewport().add(jTextArea2, null);

    this.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    this.addWindowListener(
        new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    }
    );

    this.setSize(678, 518);
    this.show();
  }
```

```
public DataInputStream getArqLer() {
  return arqLer;
}

public DataOutputStream getArqEscrever() {
  return arqEscrever;
}

public static void main(String args[]) {
  JanelaPrincipal eo = new JanelaPrincipal();
  try {
    eo.jbInit();
  }
  catch (Exception e) {}
}

void jButton1_actionPerformed(ActionEvent e) {
  EscolheArquivo a1 = new EscolheArquivo(this);
  arqLer = a1.getArquivo();
  LeitorBC lb = new LeitorBC(jTextArea1, arqLer);
  lb.imprimaByteCode();
  jButton2.setEnabled(true);
  jButton4.setEnabled(true);
}

void jButton2_actionPerformed(ActionEvent e) {
  EscolheArquivo2 a2 = new EscolheArquivo2();
  arqEscrever = a2.getArquivo();
  jButton3.setEnabled(true);
}

void jButton3_actionPerformed(ActionEvent e) {
  EscolheOtimizacao eo = new EscolheOtimizacao(arqLer, arqEscrever);
```

```
  }

  void jButton4_actionPerformed(ActionEvent e) {
    ExibirGrafo eg = new ExibirGrafo(arqLer);
  }

  void jButton5_actionPerformed(ActionEvent e) {
    jTextArea1.setText("");
    jTextArea2.setText("");
  }
}

class JanelaPrincipal_jButton1_actionAdapter
    implements java.awt.event.ActionListener {
  JanelaPrincipal adaptee;

  JanelaPrincipal_jButton1_actionAdapter(JanelaPrincipal adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.jButton1_actionPerformed(e);
  }
}

class JanelaPrincipal_jButton2_actionAdapter
    implements java.awt.event.ActionListener {
  JanelaPrincipal adaptee;

  JanelaPrincipal_jButton2_actionAdapter(JanelaPrincipal adaptee) {
    this.adaptee = adaptee;
  }
```

```
  public void actionPerformed(ActionEvent e) {
    adaptee.jButton2_actionPerformed(e);
  }
}

package outros;

import java.util.*;
import java.io.*;

public class Leitor {

  private DataInputStream arquivo;

  //contém todo bytecode
  private Vector byteCode;

  //contém os índices da [] constant_pool
  //com o attribute_name_index == "Code"
  private Vector codes;

  //contém os índices do Vector byteCode
  //com o attribute_name_index == "Code"
  private Vector ponteiros;

  //contém os índices das instruções que serão otimizadas
  private Vector instrucoes;

  public Leitor(DataInputStream arq) {
    arquivo = arq;
    byteCode = new Vector();
    codes = new Vector();
    ponteiros = new Vector();
```

```
  instrucoes = new Vector();
}

public Vector getBC() {
  return byteCode;
}

public Vector getCodes() {
  return codes;
}

public Vector getInstrucoes() {
  return instrucoes;
}

public Vector getPonteiros() {
  return ponteiros;
}

public void lerBC() {
  try {
    byteCode.add(new Integer(arquivo.readInt())); //0xCAFEBABE
    byteCode.add(new Integer(arquivo.readUnsignedShort())); //minor_version
    byteCode.add(new Integer(arquivo.readUnsignedShort())); //major_version

    int cp_pool_count = arquivo.readUnsignedShort();
    byteCode.add(new Integer(cp_pool_count)); //cp_pool_count

    for (int i = 1; i < cp_pool_count; i++) {
      int tag = arquivo.readUnsignedByte();
      byteCode.add(new Integer(tag));
      switch (tag) {
        case 1: //CONSTANT_Utf8
```

```
  String constant = arquivo.readUTF(arquivo);
  if (constant.equals("Code")) {
    codes.add(new Integer(i));
    ponteiros.add(new Integer(byteCode.size() - 1));
  }
  byteCode.add(constant);
  break;
case 3: //CONSTANT_Integer
  byteCode.add(new Integer(arquivo.readInt()));
  break;
case 4: //CONSTANT_Float
  byteCode.add(new Float(arquivo.readFloat()));
  break;
case 5: //CONSTANT_Long
  byteCode.add(new Long(arquivo.readLong()));
  i++;
  break;
case 6: //CONSTANT_Double
  byteCode.add(new Double(arquivo.readDouble()));
  i++;
  break;
case 7: //CONSTANT_Class
  byteCode.add(new Integer(arquivo.readUnsignedShort()));
  break;
case 8: //CONSTANT_String
  byteCode.add(new Integer(arquivo.readUnsignedShort()));
  break;
case 9: //CONSTANT_Fieldref
  byteCode.add(new Integer(arquivo.readInt()));
  break;
case 10: //CONSTANT_Methodref
  byteCode.add(new Integer(arquivo.readInt()));
  break;
```

```
    case 11: //CONSTANT_InterfaceMethodref
      byteCode.add(new Integer(arquivo.readInt()));
      break;
    case 12: //CONSTANT_NameAndType
      byteCode.add(new Integer(arquivo.readInt()));
      break;
  }
}

byteCode.add(new Integer(arquivo.readUnsignedShort())); //access_flags
byteCode.add(new Integer(arquivo.readUnsignedShort())); //this_class
byteCode.add(new Integer(arquivo.readUnsignedShort())); //super_class

int interfaces_count = arquivo.readUnsignedShort();
byteCode.add(new Integer(interfaces_count));
for (int i = 1; i <= interfaces_count; i++)
  byteCode.add(new Integer(arquivo.readUnsignedShort()));

int fields_count = arquivo.readUnsignedShort();
byteCode.add(new Integer(fields_count));
for (int i = 0; i < fields_count; i++) {
  byteCode.add(new Integer(arquivo.readUnsignedShort()));
  byteCode.add(new Integer(arquivo.readUnsignedShort()));
  byteCode.add(new Integer(arquivo.readUnsignedShort()));

  int attributes_count = arquivo.readUnsignedShort();
  byteCode.add(new Integer(attributes_count));
  for (int j = 1; j <= attributes_count; j++) {
    byteCode.add(new Integer(arquivo.readUnsignedShort()));
    int attribute_length = arquivo.readInt();
    byteCode.add(new Integer(attribute_length));
    for (int k = 1; k <= attribute_length; k++) {
      byteCode.add(new Integer(arquivo.readUnsignedByte()));
```

```
    }
  }

}

int methods_count = arquivo.readUnsignedShort();
byteCode.add(new Integer(methods_count));
for (int i = 1; i <= methods_count; i++) {
  byteCode.add(new Integer(arquivo.readUnsignedShort())); //access_flags
  byteCode.add(new Integer(arquivo.readUnsignedShort())); //name_index
  byteCode.add(new Integer(arquivo.readUnsignedShort())); //descriptor_index

  int attributes_count = arquivo.readUnsignedShort();
  byteCode.add(new Integer(attributes_count));
  for (int j = 1; j <= attributes_count; j++) {
    //byteCode.add(arquivo.readUnsignedShort());//access_flags
    int attribute_name_index = arquivo.readUnsignedShort();
    byteCode.add(new Integer(attribute_name_index));

    boolean ehCode = false;
    if (codes.indexOf(new Integer(attribute_name_index)) != -1)
      ehCode = true; //guarda as instruções

    if (ehCode) {
      int attribute_length = arquivo.readInt();
      byteCode.add(new Integer(attribute_length));
      Vector codigo = new Vector();
      for (int k = 1; k <= attribute_length; k++) {
        codigo.add(new Integer(arquivo.readUnsignedByte()));
        System.err.print( ( (Integer) codigo.get(k - 1)).intValue());
      }
      instrucoes.add(new Integer(byteCode.size()));
      byteCode.add(codigo);
```

```
          System.err.print("\n");
        }
        else {
          int attribute_length = arquivo.readInt();
          byteCode.add(new Integer(attribute_length));
          for (int k = 1; k <= attribute_length; k++) {
            byteCode.add(new Integer(arquivo.readUnsignedByte()));
          }
        }
      }

    }

    int attributes_count = arquivo.readUnsignedShort();
    byteCode.add(new Integer(attributes_count));
    for (int i = 1; i <= attributes_count; i++) {
      byteCode.add(new Integer(arquivo.readUnsignedShort())); //attribute_name_index

      int attribute_length = arquivo.readInt();
      byteCode.add(new Integer(attribute_length));
      for (int j = 1; j <= attribute_length; j++) {
        byteCode.add(new Integer(arquivo.readUnsignedByte()));
      }
    }
  }
  catch (Exception e) {
    System.err.println("Erro ao ler arquivo!");
    System.exit(1);
  }
 }
}

package outros;
```

```
import java.io.*;
import javax.swing.*;

public class LeitorBC {

  private DataInputStream arq;
  private JTextArea jta;
  public LeitorBC(JTextArea textArea, DataInputStream a) {
    jta = textArea;
    arq = a;
  }

  private void interface_table() {
    try {
      int count = arq.readUnsignedShort();
      jta.append("\ninterfaces_count: " + count);
      jta.append("\ninterfaces[]:\n");
      for (int i = 1; i <= count; i++)
        jta.append("\tinterface " + i + ": " +
                arq.readUnsignedShort() + "\n");
    }
    catch (Exception e) {
      jta.append("\nErro ao ler arquivo!");
    }
  }

  private void methods_table() {
    try {
      int count = arq.readUnsignedShort();
      jta.append("\nmethods_count: " + count);
      jta.append("\nmethods[]:");
      for (int i = 1; i <= count; i++) {
```

```
      jta.append("\n  method " + i + ":");
      jta.append("\taccess_flags: 0x");
      this.ler(2, true);
      jta.append("\n\tname_index: ");
      this.ler(2, false);
      jta.append("\n\tdescriptor_index: ");
      this.ler(2, false);
      jta.append("\n\tattributes_count: ");
      int at_count = arq.readUnsignedShort();
      jta.append(at_count + "\n\tattributtes[]: ");
      for (int j = 1; j <= at_count; j++) {
        jta.append("\n\t\tatributte " + j + ": ");
        jta.append("\n\t\tattribute_name_index: ");
        this.ler(2, false);
        int at_length = arq.readInt();
        jta.append("\n\t\tattribute_length: "
                + at_length);
        jta.append("\n\t\tinfo[]: ");
        for (int k = 1; k <= at_length; k++) {
          //jta.append("\n");
          this.ler(1, true);


        }


      }
      jta.append("\n");
    }
  }
  catch (Exception e) {
    jta.append("\nErro ao ler arquivo!");
  }
}
```

```
private void attributes_table() {
  try {
    int at_count = arq.readUnsignedShort();
    jta.append("\nattributes_count: " + at_count);
    for (int i = 1; i <= at_count; i++) {
      jta.append("\nattribute_name_index: ");
      this.ler(2, false);
      int at_length = arq.readInt();
      jta.append("\nattribute length: " + at_length);
      jta.append("\ninfo[]: ");
      for (int j = 1; j <= at_length; j++) {
        this.ler(1, true);
      }
    }
    jta.append("\n");
  }
  catch (Exception e) {
    jta.append("\nErro ao ler arquivo!");
  }
}

private void fields_table() {
  try {
    int count = arq.readUnsignedShort();
    jta.append("\nfields_count: " + count);
    jta.append("\nfields[]:");
    for (int i = 0; i < count; i++) {
      jta.append("\n  indice " + i + ":");
      jta.append("\taccess_flags: 0x");
      this.ler(2, true);
      jta.append("\n\tname_index: ");
      this.ler(2, false);
      jta.append("\n\tdescriptor_index: ");
```

```
      this.ler(2, false);
      jta.append("\n\tattributes_count: ");
      int at_count = arq.readUnsignedShort();
      jta.append(at_count + "\n\tattributtes[]: ");
      for (int j = 1; j <= at_count; j++) {
        jta.append("\n\t\tindice " + j + ": ");
        jta.append("\n\t\tattribute_name_index: ");
        this.ler(2, false);
        int at_length = arq.readInt();
        jta.append("\n\t\tattribute_length: "
                + at_length);
        jta.append("\n\t\tinfo[]: ");
        for (int k = 1; k <= at_length; k++) {
          this.ler(1, true);
        }

      }
      jta.append("\n");
    }
  }
  catch (Exception e) {
    jta.append("\nErro ao ler arquivo!");
  }
}

private void version() {
  try {
    //minor_version
    int m = arq.readUnsignedShort();
    //major_version
    int M = arq.readUnsignedShort();
    jta.append(M + "." + m);
  }
```

```
  catch (Exception e) {
    jta.append("\nErro ao ler arquivo!");
  }
}

private void cp_table() {
  try {
    int count = arq.readUnsignedShort();
    jta.append("\nconstant_pool_count: " + count);
    jta.append("\nconstant_pool[]:\n");
    for (int i = 1; i < count; i++) {
      int tag = arq.readUnsignedByte();
      jta.append("  indice " + i + ":");
      jta.append("\ttag:" + tag + " (");
      switch (tag) {
        case 1: //CONSTANT_Utf8
          jta.append("CONSTANT_utf8)\n\t");
          jta.append("string: " + arq.readUTF(arq));
          break;
        case 3: //CONSTANT_Integer
          jta.append("CONSTANT_Integer)\n\t");
          jta.append("int: " + arq.readInt());
          break;
        case 4: //CONSTANT_Float
          jta.append("CONSTANT_Float)\n\t");
          jta.append("float: " + arq.readFloat());
          break;
        case 5: //CONSTANT_Long
          jta.append("CONSTANT_Long)\n\t");
          jta.append("long: " + arq.readLong());
          ++i;
          break;
        case 6: //CONSTANT_Double
```

```
  jta.append("CONSTANT_Double)\n\t");
  jta.append("double: " + arq.readDouble());
  ++i;
  break;
case 7: //CONSTANT_Class
  jta.append("CONSTANT_Class)\n\t");
  jta.append("name_index: ");
  ler(2, false);
  break;
case 8: //CONSTANT_String
  jta.append("CONSTANT_String)\n\t");
  jta.append("string_index: ");
  ler(2, false);
  break;
case 9: //CONSTANT_Fieldref
  jta.append("CONSTANT_Fieldref)\n\t");
  jta.append("class_index: ");
  ler(2, false);
  jta.append("\n\tname_and_type_index: ");
  ler(2, false);
  break;
case 10: //CONSTANT_Methodref
  jta.append("CONSTANT_Methodref)\n\t");
  jta.append("class_index: ");
  ler(2, false);
  jta.append("\n\tname_and_type_index: ");
  ler(2, false);
  break;
case 11: //CONSTANT_InterfaceMethodref
  jta.append("CONSTANT_InterfaceMethodref)\n\t");
  jta.append("class_index: ");
  ler(2, false);
  jta.append("\n\tname_and_type_index: ");
```

```
        ler(2, false);
        break;
      case 12: //CONSTANT_NameAndType
        jta.append("CONSTANT_NameAndType)\n\t");
        jta.append("name_index: ");
        ler(2, false);
        jta.append("\n\tdescriptor_index: ");
        ler(2, false);
        break;
     }
     jta.append("\n");
    }
  }
  catch (Exception e) {
   jta.append("\nErro ao ler arquivo!");
  }
}

private void ler(int num, boolean hex) {
  try {
   switch (num) {
     case 1:
      if (hex) jta.append(Integer.toHexString(
          arq.readUnsignedByte()));
      else jta.append("" + arq.readUnsignedByte());
      break;
     case 2:
      if (hex) jta.append("" + Integer.toHexString(
          arq.readUnsignedShort()));
      else jta.append("" + arq.readUnsignedShort());
      break;
     case 4:
      if (hex) jta.append("" + Integer.toHexString(
```

```
        arq.readInt()));
      else jta.append("" + arq.readInt());
      break;
    }
  }
  catch (Exception e) {
    jta.append("\nErro ao ler arquivo!");
  }
}

/*******************************************************
 *      ESTRUTURA DO ARQUIVO CLASS
 *
 * ClassFile {
 *
 *      u4 magic;
 *      u2 minor_version;
 *      u2 major_version;
 *      u2 constant_pool_count;
 *      cp_info constant_pool[constant_pool_count-1];
 *      u2 access_flags;
 *      u2 this_class;
 *      u2 super_class;
 *      u2 interfaces_count;
 *      u2 interfaces[interfaces_count];
 *      u2 fields_count;
 *      field_info fields[fields_count];
 *      u2 methods_count;
 *      method_info methods[methods_count];
 *      u2 attributes_count;
 *      attribute_info attributes[attributes_count];
 *  }
 *
```

```
  *****************************************************/
  public void imprimaByteCode() {
//    jta.append("Arquivo: "
 //              + arq.getClass() + "\n");

    jta.append("\nmagic: 0x");
    this.ler(4, true);
    jta.append("\nversion: ");
    this.version();
    this.cp_table();
    jta.append("\naccess_flags: 0x");
    this.ler(2, true);
    jta.append("\nthis_class: ");
    this.ler(2, false);
    jta.append("\nsuper_class: ");
    this.ler(2, false);
    this.interface_table();
    this.fields_table();
    this.methods_table();
    this.attributes_table();
  }

}

package outros;

//ARQUIVO MONTAATRIBUTO.JAVA
/*
 Code_attribute {

        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
```

```
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {
            u2 start_pc;
           u2 end_pc;
          u2  handler_pc;
          u2  catch_type;
    }     exception_table[exception_table_length];
          u2 attributes_count;
          attribute_info attributes[attributes_count];
 }

 */
import java.util.*;

public class MontaAtributo {

  private Vector byteCode;

  public MontaAtributo(Vector bc) {
    byteCode = bc;
  }

  //recebe o índice do atributo code no Vector
  public Vector monta(Vector instrucoes, int indice) {

    //substitui o tamanho do atributo inteiro
    int atributte_length =
        ( (Integer) byteCode.get(indice + 1)).intValue();
    int code_length =
        ( (Integer) byteCode.get(indice + 4)).intValue();
```

```
    int resto =
        atributte_length - code_length;
    int novo_tamanho = instrucoes.size() + resto;

    byteCode.setElementAt(
        new Integer(novo_tamanho), indice + 1);

    //substitui o tamanho das instrucoes
    byteCode.setElementAt(
        new Integer(instrucoes.size()), indice + 4);

    //substitui as instrucoes
    byteCode.setElementAt(instrucoes, indice + 5);

    return byteCode;
  }
}

package outros;

import java.io.*;
import java.util.*;

public class Otimizacao {
  boolean[] quaisOtimizacoes;
  DataInputStream arqLer;
  DataOutputStream arqEscrever;
  public Otimizacao(DataInputStream dis, DataOutputStream dos, boolean[] quais) {
    quaisOtimizacoes = quais;
    arqLer = dis;
    arqEscrever = dos;
  }
```

```
  public Vector otimiza() {
    Leitor leitor = new Leitor(arqLer);
    leitor.lerBC();
    Vector byteCode = leitor.getBC();
    Vector codes = leitor.getCodes();
    Vector ponteiros = leitor.getPonteiros();
    Vector instrucoes = leitor.getInstrucoes();

    Separador separador = new Separador();
    Otimizador otimizador = new Otimizador();

    Vector metodoNovo = new Vector();
    for (int i = 0; i < instrucoes.size(); i++) {
      int aux = ( (Integer) instrucoes.get(i)).intValue();
      Vector umMetodo = (Vector) byteCode.get(aux);
      Vector blocos = separador.separa(umMetodo);

      for (int j = 0; j < blocos.size(); j++) {
        Vector umBloco = (Vector) blocos.get(j);
        Vector blocoOtimizado = otimizador.otimiza(quaisOtimizacoes, umBloco);
      }
    }
    //montar novo método
    return metodoNovo;
  }
}

package outros;

import java.util.*;

public class Otimizador {
```

```
Vector instrucoes;
public Otimizador() {
}

public Vector otimiza(boolean[] quaisOtimizacoes, Vector inst) {
  instrucoes = inst;
  //chamar as outras funções de otimização
  if (quaisOtimizacoes[0])
    this.peepHole();
  if (quaisOtimizacoes[1])
    this.copias();
  if (quaisOtimizacoes[2])
    this.subexpressoes();
  if (quaisOtimizacoes[3])
    this.movimentacao();
  if (quaisOtimizacoes[4])
    this.inducao();

  return instrucoes;
}

public void copias() {
}

public void subexpressoes() {
}

public void movimentacao() {
}

public void inducao() {
}
```

```
public void peepHole() {
 /* inicio simplificação algébrica */
 for (int i = 0; i < instrucoes.size(); i++) {
  int instrucao = ( (Integer) instrucoes.get(i)).intValue();

  //verifica se carrega constante 1 ou 0
  // ou se não é a última instrução
  if ( (instrucao == 3 || instrucao == 4) && (i + 1 != instrucoes.size())) {
   int proxima = ( (Integer) instrucoes.get(i + 1)).intValue();
   if (instrucao == 3) {
    if (proxima == 96 || proxima == 100) { //iadd ou isub
     instrucoes.setElementAt(new Integer(0), i);
     instrucoes.setElementAt(new Integer(0), i + 1);
    }
   }
   else {
    if (proxima == 104 || proxima == 108) { //imul ou idiv
     instrucoes.setElementAt(new Integer(0), i);
     instrucoes.setElementAt(new Integer(0), i + 1);
    }
   }
  }
  else i += pulaQuantos(instrucao, i);
 }
 /* fim simplificação algébrica */

 /*-------------------------------------------------------------*/

 /* desvios para desvios */
 for (int i = 0; i < instrucoes.size(); i++) {

  int instrucao = ( (Integer) instrucoes.get(i)).intValue();
  int alvo = -1;
```

```
//incondicional pra incondicional
if (instrucao == 167 || instrucao == 200) {
  if (instrucao == 167) {
    int branchbyte1 = ( (Integer) instrucoes.get(i + 1)).intValue();
    int branchbyte2 = ( (Integer) instrucoes.get(i + 2)).intValue();
    alvo = (branchbyte1 << 8) | branchbyte2;

    int instrucao_alvo = ( (Integer) instrucoes.get(alvo)).intValue();
    if (instrucao_alvo == 167) {
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 1), i + 1);
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 2), i + 2);
    }
    i += 2;
  }
  else {
    int branchbyte1 = ( (Integer) instrucoes.get(i + 1)).intValue();
    int branchbyte2 = ( (Integer) instrucoes.get(i + 2)).intValue();
    int branchbyte3 = ( (Integer) instrucoes.get(i + 3)).intValue();
    int branchbyte4 = ( (Integer) instrucoes.get(i + 4)).intValue();
    alvo = (branchbyte1 << 24) | (branchbyte2 << 16)
        | (branchbyte3 << 8) | branchbyte4;
    int instrucao_alvo = ( (Integer) instrucoes.get(alvo)).intValue();
    if (instrucao_alvo == 200) {
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 1), i + 1);
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 2), i + 2);
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 3), i + 3);
      instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 4), i + 4);
    }
    i += 4;
  }
}
else {
  //condicional pra incondicional
```

```
      if ( (instrucao >= 153 && instrucao <= 166)
          || (instrucao == 198 || instrucao == 199)) {
        int branchbyte1 = ( (Integer) instrucoes.get(i + 1)).intValue();
        int branchbyte2 = ( (Integer) instrucoes.get(i + 2)).intValue();
        alvo = (branchbyte1 << 8) | branchbyte2;

        int instrucao_alvo = ( (Integer) instrucoes.get(alvo)).intValue();
        if (instrucao_alvo == 167) {
          instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 1), i + 1);
          instrucoes.setElementAt( (Integer) instrucoes.get(alvo + 2), i + 2);
        }
        i += 2;
      }
      else i += pulaQuantos(instrucao, i);
    }
  }
  /* fim desvios para desvios */

  /*------------------------------------------------------------*/
}

//retorna quantos bytes a instrucao tem como branchbyte
private int pulaQuantos(int instrucao, int indice) {

  if (instrucao >= 0 && instrucao <= 15)
    return 0;
  else if (instrucao == 16)
    return 1;
  else if (instrucao == 17)
    return 2;
  else if (instrucao == 18)
    return 1;
  else if (instrucao == 19 || instrucao == 20)
```

```
  return 2;
else if (instrucao >= 21 && instrucao <= 25)
  return 1;
else if (instrucao >= 26 && instrucao <= 53)
  return 0;
else if (instrucao == 54 || instrucao == 55)
  return 1;
else if (instrucao == 56)
  return 0;
else if (instrucao == 57 || instrucao == 58)
  return 1;
else if (instrucao >= 59 && instrucao <= 131)
  return 0;
else if (instrucao == 132)
  return 2;
else if (instrucao >= 133 && instrucao <= 152)
  return 0;
else if (instrucao >= 153 && instrucao <= 168)
  return 2;
else if (instrucao == 169)
  return 1;
//170-171
else if (instrucao >= 172 && instrucao <= 177)
  return 0;
else if (instrucao >= 178 && instrucao <= 184)
  return 2;
else if (instrucao == 185)
  return 4;
//186 unused
else if (instrucao == 187)
  return 2;
else if (instrucao == 188)
  return 1;
```

```
    else if (instrucao == 189)
      return 2;
    else if (instrucao == 190 || instrucao == 191)
      return 0;
    else if (instrucao == 192 || instrucao == 193)
      return 2;
    else if (instrucao == 194 || instrucao == 195)
      return 0;
    //196 3 ou 5
    else if (instrucao == 197)
      return 3;
    else if (instrucao == 198 || instrucao == 199)
      return 2;
    else if (instrucao == 200 || instrucao == 201)
      return 4;

    else if (instrucao == 196) {
      if ( ( (Integer) instrucoes.get(indice + 1)
            ).intValue() == 132)
        return 5;
      else return 3;
    }
    else if (instrucao == 170)
      return -1;
    else if (instrucao == 171)
      return -1;
    else return -1; //???
  }
}

package outros;

import java.util.*;
```

```
public class Separador {
 /*
  Algoritmo de separação em blocos básicos:

  1. Determinamos primeiro o conjunto de líderes, os primeiros enunciados
  dos blocos básicos. As regras que usamos são as seguintes:

  i)   O primeiro enunciado é um líder.
  ii)  Qualquer enunciado que seja objeto de um desvio condicional ou
     incondicional é um líder.
  iii) Qualquer enunciado que siga imediatamente um enunciado de desvio
     condicional ou incondicional é um líder.

  2. Para cada líder, seu bloco básico consiste no líder e em todos os
  enunciados até, mas não incluindo o próximo líder ou o final do programa.
  */

 private Vector lideres;

 public Separador() {
  lideres = new Vector();
 }

 //separa as instruções em blocos básicos
 public Vector separa(Vector instrucoes) {

  this.determinaLideres(instrucoes);
  this.organizaLideres(instrucoes);

  //System.err.println("n. instrucoes: " + instrucoes.size() + "\n\n\n\n");
  Vector blocos = new Vector();
  int liderAtual = 0;
```

```
  for (int i = 1; i < lideres.size(); i++) {
    int proximoLider = ( (Integer) lideres.get(i)).intValue();
    //System.out.println("iteracao: " + i);
    //System.out.println("\natual lider: " + liderAtual);
    //System.out.println("\nproximo lider: " + proximoLider);
    Vector umBloco = new Vector();
    //System.out.println("\nj: ");
    for (int j = liderAtual; j < proximoLider; j++) {
      //System.out.print(j + " ");
      Integer inst = (Integer) instrucoes.get(j);
      umBloco.add(inst);
    }
    liderAtual = proximoLider;
    blocos.add(umBloco);
  }
  Vector ultimoBloco = new Vector();
  for (int i = liderAtual; i < instrucoes.size(); i++) {
    ultimoBloco.add(instrucoes.get(i));
  }
  blocos.add(ultimoBloco);

  //retorna um vector com as instruçoes separadas
  //em blocos básicos
  return blocos;
}

//verifica os lideres e armazena os
//respectivos indices no Vector lideres
private void determinaLideres(Vector instrucoes) {
  //i) o primeiro enunciado é líder
  lideres.add(new Integer(0));
  for (int i = 0; i < instrucoes.size(); i++) {
    int instrucao = ( (Integer) instrucoes.get(i)).intValue();
```

```
    if (this.ehDesvio(instrucao)) {
      //ii) Qualquer enunciado que seja objeto de um
      //    desvio condicional ou incondicional é um líder.
      int naoInstrucoes = pulaQuantos(instrucao, i, instrucoes);
      int alvo = this.calculaAlvo(instrucoes, i);
      lideres.add(new Integer(alvo));
      //iii) Qualquer enunciado que siga imediatamente
      //     um enunciado de desvio condicional ou
      //     incondicional é um líder.
      if (i + 1 < instrucoes.size())
        lideres.add(new Integer(i + naoInstrucoes + 1));

    }

    i += this.pulaQuantos(instrucao, i, instrucoes);
  }
}

//arruma o Vector de lideres em ordem crescente
private void organizaLideres(Vector instrucoes) {

  Vector organizado = new Vector();
  organizado.add(new Integer(0));
  for (int i = 1; i < lideres.size(); i++) {
    int menor = instrucoes.size();
    //((Integer)lideres.get(i)).intValue();
    for (int j = 1; j < lideres.size(); j++) {
      int outro = ( (Integer) lideres.get(j)).intValue();
      int ultimo = ( (Integer) organizado.get(i - 1)).intValue();
      if (menor > outro && ultimo < outro)
        menor = outro;
    }
    organizado.add(new Integer(menor));
```

```
  }

  lideres = organizado;
}

private boolean ehDesvio(int instrucao) {
  //153 - 171
  //198 - 201
  if (instrucao >= 153 && instrucao <= 171)
    return true;

  if (instrucao >= 198 && instrucao <= 201)
    return true;

  return false;
}

//retorna quantos bytes a instrucao tem como branchbyte
private int pulaQuantos(int instrucao, int indice, Vector instrucoes) {
  if (instrucao >= 0 && instrucao <= 15)
    return 0;
  else if (instrucao == 16)
    return 1;
  else if (instrucao == 17)
    return 2;
  else if (instrucao == 18)
    return 1;
  else if (instrucao == 19 || instrucao == 20)
    return 2;
  else if (instrucao >= 21 && instrucao <= 25)
    return 1;
  else if (instrucao >= 26 && instrucao <= 53)
    return 0;
```

```
else if (instrucao == 54 || instrucao == 55)
  return 1;
else if (instrucao == 56)
  return 0;
else if (instrucao == 57 || instrucao == 58)
  return 1;
else if (instrucao >= 59 && instrucao <= 131)
  return 0;
else if (instrucao == 132)
  return 2;
else if (instrucao >= 133 && instrucao <= 152)
  return 0;
else if (instrucao >= 153 && instrucao <= 168)
  return 2;
else if (instrucao == 169)
  return 1;
//170-171
else if (instrucao >= 172 && instrucao <= 177)
  return 0;
else if (instrucao >= 178 && instrucao <= 184)
  return 2;
else if (instrucao == 185)
  return 4;
//186 unused
else if (instrucao == 187)
  return 2;
else if (instrucao == 188)
  return 1;
else if (instrucao == 189)
  return 2;
else if (instrucao == 190 || instrucao == 191)
  return 0;
else if (instrucao == 192 || instrucao == 193)
```

```
    return 2;
  else if (instrucao == 194 || instrucao == 195)
    return 0;
  //196 3 ou 5
  else if (instrucao == 197)
    return 3;
  else if (instrucao == 198 || instrucao == 199)
    return 2;
  else if (instrucao == 200 || instrucao == 201)
    return 4;

  else if (instrucao == 196) {
    if ( ( (Integer) instrucoes.get(indice + 1)
        ).intValue() == 132)
      return 5;
    else return 3;
  }
  else if (instrucao == 170)
    return -1;
  else if (instrucao == 171)
    return -1;
  else return -1; //???
}

//retorna o deslocamento(alvo) da instrucao de desvio
private int calculaAlvo(Vector instrucoes, int indice) {
  int instrucao = ( (Integer) instrucoes.get(indice)).intValue();
  int branchbyte1, branchbyte2, branchbyte3, branchbyte4;
  int branchoffset = -1;
  switch (instrucao) {
    case 153: //ifeq
    case 154: //iflt
    case 155: //ifge
```

```
case 156: //ifgt
case 157: //ifgt
case 158: //ifle
case 159: //if_icmpeq
case 160: //if_icmpne
case 161: //if_icmplt
case 162: //if_icmpge
case 163: //if_icmpgt
case 164: //if_icmple
case 165: //if_acmpeq
case 166: //if_acmpne
case 167: //goto
case 168: //jsr
case 198: //ifnull
case 199: //ifnonnull
  branchbyte1 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchbyte2 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchoffset = (branchbyte1 << 8) | branchbyte2;
  break;
case 200: //goto_w
case 201: //jsr_w
  branchbyte1 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchbyte2 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchbyte3 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchbyte4 = ( (Integer) instrucoes.get(++indice)).intValue();
  branchoffset = (branchbyte1 << 24) | (branchbyte2 << 16) |
      (branchbyte3 << 8) | branchbyte4;
  break;
case 169: //ret

  //calcular
  break;
case 170: //tableswitch
```

```
      case 171: //lookupswitch

        //variavel
        break;
      default:
        break;

    }

    return branchoffset;
  }
}
```