Trabalho Prático III

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Compiladores I

João Francisco B. S. Martins, Pedro D. V. Chaves
{joaofbsm, pedrodallav}@dcc.ufmg.br

12 de Novembro de 2017

1 Introdução

O trabalho em questão tem como objetivo implementar um **tradutor** da linguagem intermediária gerada pelo *front-end* da linguagem **SmallL** para Java **bytecode**. Para codificação dos componentes do tradutor foi utilizada a linguagem de programação **Java** (*v.8*) e a linguagem **Python** (*v.3*).

Para auxiliar a implementação foram utilizadas as seguintes ferramentas:

- **Jasmin**: *assembler* para Java que recebe descrição textual de classes Java, numa sintaxe de bytecodes extendida, e as converte para arquivos binários no formato `.class`
- **Krakatau**: *decompiler*, *disassembler* e *assembler* para arquivos `.class` Java.

A ferramenta Jasmin foi desenvolvida para acompanhar o livro *Java Virtual Machine*[1], e, apesar de já ter sido descontinuada há 9 anos, sua sintaxe é tão mais límpida e fácil de editar e entender, do que a gerada pelas próprias ferramentas Java, que ela continua sendo usada até hoje pelos assemblers e disassemblers *third-party*, como o **ASM**, o **Soot** e o próprio **Krakatau**. Portanto a única coisa utilizada de **Jasmin** nesse trabalho foi sua sintaxe, e não a ferramenta de montagem propriamente dita.

O trabalho foi desenvolvido utilizando a ferramenta de versionamento Git juntamente com a plataforma de desenvolvimento remoto GitHub. O repositório do projeto contém não só o código fonte, mas também os scripts

auxiliares desenvolvidos e os arquivos de teste, podendo ser acessado no endereço `https://github.com/joaofbsm/smallL`. Mais especificamente, o código do tradutor descrito neste trabalho se encontra na subpasta `https://github.com/joaofbsm/smallL/tree/master/code/translator`.

2 Conceitos e Ferramentas

A seguir serão apresentados conceitos e definições para auxiliar o entendimento do trabalho implementado. Também será apresentado como as ferramentas escolhidas auxiliaram no desenvolvimento do **tradutor**.

2.1 Bytecode Java e JVM

O código de um programa de computador escrito na linguagem Java é compilado para uma forma intermediária de código denominada **bytecode**, que é interpretada pelas Máquinas Virtuais Java (JVMs) e independe da arquitetura da máquina que a gerou. A JVM é o programa que carrega e executa os aplicativos Java, convertendo os bytecodes em código executável de máquina. Os *bytecodes* são portanto o conjunto de instruções da JVM.

A interpretação processa os *bytecodes* um por um, promovendo modificações no estado da máquina virtual.

2.2 Jasmin e Krakatau

Para promover o *disassemble* de um arquivo `.class`, utiliza-se o comando `javap -c`, gerando como saída uma forma legível por humanos das instruções que compõem os *bytecodes* Java.

Porém, não é possivel voltar da saída do comando anterior para o *bytecode* original (instruções em hexadecimal), pois ocorrem perdas de informação durante o processo de *disassemble*.

Para contornar tal problema, utilizou-se a sintaxe do **Jasmin** com auxílio das ferramentas de montagem e desmontagem do **Krakatau**. Ou seja, a saída do gerador de código intermedíario (quadruplas) é parseada e traduzida, gerando instruções no formato **Jasmin** (sintaxe no formato de *assembler* usando o conjunto de instruções da JVM). O *assembler* do **Krakatau** é então utilizado para gerar *bytecodes* binários que podem rodar na JVM.

3 Desenvolvimento

3.1 Visão Geral

O **tradutor** em questão teve sua implementação feita através da linguagem **Python** pela maior facilidade presente para manipulação de cadeias de caracteres (passo essencial para a tradução). Além disso, Python foi usada para tornar a comunicação com a ferramenta **Krakatau** mais fácil e eficiente.

O *front-end* implementado no trabalho anterior tem como saída um código intermediário que tem por base 7 tipos de operações possíveis:

- **atribuição direta**: `x = y`
- **atribuição com expressão aritmética**: `x = y arith_op z`
- **atribuição para posição de vetor**: `x[p] = y`
- **atribuição de posição de vetor**: `x = y[p]`
- **if com comparação**: `if x logic_op y goto L`
- **iffalse com comparação**: `iffalse x logic_op y goto L`
- **desvio**: `goto L`

A partir dos formatos de operações descritos acima, foi possível desenvolver um **tradutor** que indentificasse cada tipo de formato no arquivo de saída do gerador de código intermediário e gerasse código na sintaxe **Jasmin**, que por sua vez seria passado para o *assembler* do **Krakatau** para gerar *bytecodes* binários.

3.2 Implementação

O **tradutor** é implementado através dos 4 arquivos abaixo (além da integração com o **Krakatau**):

- `translator.py`: parseia e traduz o arquivo de código intermediário para *bytecodes* Java utilizando sintaxe **Jasmin**.
- `opbuilder.py`: constrói operações baseadas na sintaxe de *bytecodes* **Jasmin**.
- `operation.py`: classe auxiliar para estruturar a representação de uma **operação**.

- `variable.py`: classe auxiliar para estruturar a representação de uma **variável**.

O arquivo `translator.py` parseia o arquivo de entrada, gerando dicionários para os *labels* (um dicionário para guardar os diferentes *labels* e suas respectivas linhas e outro para guardar referências para *labels* já existentes).

O bytecode não permite que uma mesma linha possua mais de um *label*, como pode ocorrer no código intermediário. Sendo assim, uma forma de criar equivalências entre *labels* foi implementada utilizando um dicionário, que é preenchido nessa primeira passada sob o arquivo.

Uma vez identificados os *labels*, são geradas uma ou mais operações equivalentes na sintaxe **Jasmin**. Isso é feito através do módulo `opbuilder.py`, que mapeia cada uma dos formatos das quádruplas elucidadas acima para uma operação equivalente em **Jasmin**.

3.3 Operações em bytecode

A seguir será mostrado como cada tipo de operação que pode estar presente no código intermediário é mapeada para uma possível sequência de códigos na sintaxe **Jasmin**.

3.3.1 Atribuição direta: `x = y`

Uma atribuição simples é baseada em duas ações: carregar um operando (`y`) da memória (checando a tabela de símbolos) e salvar esse valor no operando do lado esquerdo `x`. No código abaixo, os valores de `x` e `y` já haviam sido inicializados (com valores nos endereços 1 e 3, respectivamente)

Código 1: Operação de atribuição em código intermediário.

```
L1: x = y
```

Código 2: Operação de atribuição simples em Jasmin.

```
L1:  dload 3
     dstore 1
```

3.3.2 Atribuição com expressão aritmética: `x = y arith_op z`

Uma atribuição com expressão é baseada em 3 operandos (`x` (op1), `y` (op2) e `z` (op3)) e um operador aritmético (`arith_op`). A atribuição segue as seguintes ações: primeiro carregam-se os operandos 2 e 3 da memória. Depois, os

operandos 2 e 3 são somados através do comando `dload` (ações baseadas em pilha). Por final, o valor somado é salvo no operando 1 (os operandos 1, 2 e 3 estavam salvos nos endereços 1, 3 e 5, respectivamente). A sequência de passos pode ser vista no código abaixo, podendo-se substituir os operandos 2 e 3 por constantes caso algum desses fosse uma constante (utilizando o comando `ldc2_w 1.0`, por exemplo).

Código 3: Operação de atribuição com expressão aritmética em código intermediário.

```
L2: x = y + z
```

Código 4: Operação de atribuição com expressão aritmética em Jasmin.

```
L2: dload 3
    dload 5
    dadd
    dstore 1
```

3.3.3 Atribuição para posição de vetor: `a[x] = z`

Uma atribuição para posição de vetor requer uma sequência maior de passos. Explicaremos o passo a passo de cada trecho de código presente no Código 6:

```
dload 1
ldc2_w 8.0
dmul
dstore 7
```

A JVM não requer a multiplicação do índice por 8 nos vetores, ou seja, sempre que é recebida uma variável que vai ser o índice de um array, ela por *default* já é multiplicada por 8.

O código imediatamente acima carrega o valor que vai ser o índice (salvo no endereço 1), carrega a constante 8 na pilha e multiplica. Os próximos passos são indicados abaixo:

```
sipush 1000
newarray double
astore 9
```

Cria um *array* de double de 1000 posições (por *default*) e salva na memória na posição 9.

```
dload 7
ldc2_w 8.0
ddiv
dstore 7
```

O tradutor identifica que o endereço 7 vai ser usado como o índice do array, carrega esse endereço na pilha e também carrega a constante 8. Após isso, divide ambos e salva novamente no endereço 7.

```
aload 9
dload 7
d2i
dload 5
dastore
```

Por último, carrega o ponteiro do *array* (endereço 9) e carrega a variável temporária que representa o índice (endereço 7). Pega o valor mais alto na pilha e converte de `double` para `int` (comando `d2i`), para que indexação seja possível. Finaliza carregando o valor a ser atribuído para a posição do vetor e salva esse valor na posição.

Abaixo podemos ver a sequência de passos completa.

Código 5: Atribuição para posição de vetor em código intermediário.

```
L3: t1 = x * 8
    a [ t1 ] = z
```

Código 6: Atribuição para posição de vetor em Jasmin.

```
L3: dload 1
    ldc2_w 8.0
    dmul
    dstore 7
    sipush 1000
    newarray double
    astore 9
    dload 7
    ldc2_w 8.0
    ddiv
    dstore 7
    aload 9
    dload 7
```

```
    d2i
    dload 5
    dastore
```

3.3.4 Atribuição de posição de vetor: `y = a[p]`

A atribuição de posição de vetor segue um raciocínio bem semelhante ao elucidado anteriormente na atribuição para posição de vetor, mudando apenas a ordem de alguns *loads* e *stores*. No exemplo abaixo, o valor de `p` é uma constante (igual a 1, indicando o índice 1).

Código 7: Atribuição de variável utilizando valor na posição de vetor em código intermediário.

```
L4: t3 = 1 * 8
  y = a [ t3 ]
```

Código 8: Atribuição de variável utilizando valor na posição de vetor em Jasmin.

```
L4:  ldc2_w 1.0
   ldc2_w 8.0
   dmul
   dstore 13
   dload 13
   ldc2_w 8.0
   ddiv
   dstore 13
   aload 9
   dload 13
   d2i
   daload
  dstore 3
```

3.3.5 Condicional comparativo: `if/iffalse x logic_op y goto L`

A comparação iffalse é feita da seguinte forma: primeiro são carregados os operandos (x e y) a serem comparados (na pilha) e em seguida o comando `dcmpl` compara os dois números, retornando como resultado 0 se forem iguais, 1 se y maior que x e -1 caso contrário. O último comando (`iflt`) verifica o resultado e, se x menor que y, desvia para L2.

Código 9: iffalse com comparação em código intermediário

```
L6: iffalse x >= y goto L2
```

Código 10: iffalse com comparação em Jasmin

```
L6: dload 1
    dload 3
    dcmpl
    iflt L2
```

A diferença de `iffalse` e `if` é baseada no fato de iffalse ser utilizado com operador de maior ou igual (`>=`) e if ser utilizado com operador de menor (`<`).

Com isso, uma tradução de uma operação com if, geraria a seguinte sequência:

Código 11: if com comparação em Jasmin

```
L6: dload 1
    dload 3
    dcmpl
    ifgt L2
```

Ou seja, com a lógica "inversa"do iffalse, pois utiliza o comando `ifgt`, verificando o resultado de `dcmpl` de forma invertida.

3.3.6 Desvio: `goto L`

O comando de desvio funciona da mesma forma (e com a mesma sintaxe) em ambos os códigos (intermediário e na sintaxe Jasmin).

4 Código e Utilização

Por ser muito extenso, preferimos não descrever todo o código do **tradutor** neste documento, disponibilizando-o no repositório mencionado na seção de introdução.

Para obter o código, basta clonar o repositório utilizando o comando:

```
git clone https://github.com/joaofbsm/smallL.git
```

ou baixar o `.zip` disponibilizado ao clicar em **"Clone or download"** e depois em **"Download ZIP"** (na página do repositório).

Caso tenha optado pela segunda opção, basta descompactar e entrar na pasta descompactada.

4.1 Traduzindo

Para facilitar a utilização do **tradutor** foram criados dois scripts *bash* que condensam as tarefas de compilar o código do *front-end* e executar o *front-end* (necessário para geração das quadruplas).

- `compile.sh`: responsável por compilar as classes Java necessárias para o funcionamento do *front end*.
- `execute.sh`: reponsável por testar todas as entradas de teste (código na linguagem **SmallL**) disponibilizadas no diretório `tests`.

Para traduzir os códigos intermediários gerados, basta executar o script `translate.sh` (após ter executado os dois scripts citados acima).

Para criar um caso de teste, basta adicionar um arquivo `.txt`, contendo o teste desejado (em linguagem SmallL), no diretório `tests` presente no diretório raiz do *front-end*.

Caso deseje rodar um teste em específico, cuja entrada já está em formato de código intermediário, basta rodar:

```
// mude para o diretorio raiz do front end
cd /caminho_para_diretorio_raiz/smallL
cd code/translator
python3 translator.py codigo_intermediario
```

Tal comando produzirá na saída padrão o código traduzido na sintaxe Jasmin. Para rodar a sequência de scripts completa, siga os seguintes passos:

```
// mude para o diretorio raiz do front end
cd /caminho_para_diretorio_raiz/smallL
// compila
./compile.sh
// executa front-end
./execute.sh
// traduz codigo intermediario
./translate.sh
```

A saída dos testes se encontra na pasta `outputs`. Os `.txt` gerados são referentes ao código intermediário gerado pelo *front-end*. Dentro do diretório `outputs` há uma pasta denominada `translated` que contém os códigos em Jasmin (`.j`) gerados pelo **tradutor** e os binários gerados pelo **Krakatau** (arquivos `.class` dentro das pastas `nome_teste-bin/`), a partir dos códigos gerados pelo tradutor.

Para uma melhor formatação da saída, é aconselhável rodar o `disassembler` do **Krakatau** nos arquivos `.class` gerados. Para tal, rode a seguinte linha de comando:

```
\\ entre no diretorio com a saida binaria
cd caminho_para_smallL/outputs/translated/nome_teste-bin/
\\ rode o disassembler
python2.7 ../../../tools/Krakatau/disassemble.py Main.class
```

Tal comando produzirá um arquivo `Main.j` em sintaxe Jasmin, contendo o código produzido pelo tradutor em uma formatação mais clara e organizada.

5 Testes

A seguir são apresentados alguns testes que tentam englobar todas as possíveis instruções geradas em sintaxe Jasmin. A sequência de arquivos para cada teste é a seguinte:

1. arquivo em linguagem **SmallL**
2. arquivo com código intermediário gerado a partir de 1.
3. arquivo com código traduzido a partir de 2.

5.1 Teste 1

Código 12: Arquivo para o teste 1 em linguagem SmallL

```
{
    int x; int y; int z; float d; float e; float[3] a;

    x = 1;
    y = 10;
    z = 5;

    x = y;
    x = y + z;

    a[x] = z;
    a[2] = 1.5;
```

```
    y = a[1];

    if( x >= y ) x = 1;
}
```

Código 13: Arquivo com código intermediário para o teste 1 produzido pelo front-end

```
L1: x = 1
L3: y = 10
L4: z = 5
L5: x = y
L6: x = y + z
L7: t1 = x * 8
    a [ t1 ] = z
L8: t2 = 2 * 8
    a [ t2 ] = 1.5
L9: t3 = 1 * 8
    y = a [ t3 ]
L10: iffalse x >= y goto L2
L11: x = 1
L2:
```

Código 14: **Código em sintaxe Jasmin para o teste 1 produzido pelo tradutor implementado**

```
.version 52 0
.class public super Main
.super java/lang/Object

.method public <init> : ()V
       .code stack 1 locals 1
L0:  aload_0
L1:  invokespecial Method java/lang/Object <init> ()V
L4:  return
L5:
       .end code
.end method

.method public static main : ([Ljava/lang/String;)V
       .code stack 4 locals 50
```

```
L1:           ldc2_w 1.0
              dstore 1
L3:           ldc2_w 10.0
              dstore 3
L4:           ldc2_w 5.0
              dstore 5
L5:           dload 3
              dstore 1
L6:  dload 3
              dload 5
              dadd
              dstore 1
L7:           dload 1
              ldc2_w 8.0
              dmul
              dstore 7
              sipush 1000
              newarray double
              astore 9
              dload 7
              ldc2_w 8.0
              ddiv
              dstore 7
              aload 9
              dload 7
              d2i
              dload 5
              dastore
L8:           ldc2_w 2.0
              ldc2_w 8.0
              dmul
              dstore 11
              dload 11
              ldc2_w 8.0
              ddiv
              dstore 11
              aload 9
              dload 11
              d2i
```

```
                ldc2_w 1.5
                dastore
L9:             ldc2_w 1.0
                ldc2_w 8.0
                dmul
                dstore 13
                dload 13
                ldc2_w 8.0
                ddiv
                dstore 13
                aload 9
                dload 13
                d2i
                daload
                dstore 3
L10:             dload 1
                dload 3
                dcmpl
                iflt L2
L11:             ldc2_w 1.0
                dstore 1
L2:             return
        .end code
.end method
.sourcefile 'Main.java'
.end class
```

5.2 Teste 2

Código 15: Arquivo do teste 2 em linguagem SmallL

```
{
    int i; int j; float v; float x; float[3] a;
    i = 1;
    j = 10;
    v = 2;
    x = 6;
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
```

```
    while( true ) {
        do i = i+1; while( a[i] < v);
        do j = j-1; while( a[j] > v);
        if( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

Código 16: Arquivo com código intermediário para o teste 2 produzido pelo front-end

```
L1: i = 1
L3: j = 10
L4: v = 2
L5: x = 6
L6: t1 = 0 * 8
    a [ t1 ] = 1
L7: t2 = 1 * 8
    a [ t2 ] = 2
L8: t3 = 2 * 8
    a [ t3 ] = 3
L9:L10: i = i + 1
L12: t4 = i * 8
     t5 = a [ t4 ]
     if t5 < v goto L10
L11: j = j - 1
L14: t6 = j * 8
     t7 = a [ t6 ]
     if t7 > v goto L11
L13: iffalse i >= j goto L15
L16: goto L2
L15: t8 = i * 8
     x = a [ t8 ]
L17: t9 = i * 8
     t10 = j * 8
     t11 = a [ t10 ]
     a [ t9 ] = t11
L18: t12 = j * 8
     a [ t12 ] = x
```

```
     goto L9
L2:
```

Código 17: **Código para o teste 2 em sintaxe Jasmin produzido pelo tradutor implementado**

```
.version 52 0
.class public super Main
.super java/lang/Object

.method public <init> : ()V
       .code stack 1 locals 1
L0:  aload_0
L1:  invokespecial Method java/lang/Object <init> ()V
L4:  return
L5:
       .end code
.end method

.method public static main : ([Ljava/lang/String;)V
       .code stack 4 locals 50

L1:            ldc2_w 1.0
               dstore 1
L3:            ldc2_w 10.0
               dstore 3
L4:            ldc2_w 2.0
               dstore 5
L5:            ldc2_w 6.0
               dstore 7
L6:            ldc2_w 0.0
               ldc2_w 8.0
               dmul
               dstore 9
               sipush 1000
               newarray double
               astore 11
               dload 9
               ldc2_w 8.0
               ddiv
```

```
                dstore 9
                aload 11
                dload 9
                d2i
                ldc2_w 1.0
                dastore
L7:             ldc2_w 1.0
                ldc2_w 8.0
                dmul
                dstore 13
                dload 13
                ldc2_w 8.0
                ddiv
                dstore 13
                aload 11
                dload 13
                d2i
                ldc2_w 2.0
                dastore
L8:             ldc2_w 2.0
                ldc2_w 8.0
                dmul
                dstore 15
                dload 15
                ldc2_w 8.0
                ddiv
                dstore 15
                aload 11
                dload 15
                d2i
                ldc2_w 3.0
                dastore
L9:             dload 1
                ldc2_w 1.0
                dadd
                dstore 1
L12:            dload 1
                ldc2_w 8.0
                dmul
                dstore 17
```

```
                dload 17
                ldc2_w 8.0
                ddiv
                dstore 17
                aload 11
                dload 17
                d2i
                daload
                dstore 19
                dload 19
                dload 5
                dcmpg
                iflt L9
L11:            dload 3
                ldc2_w 1.0
                dsub
                dstore 3
L14:            dload 3
                ldc2_w 8.0
                dmul
                dstore 21
                dload 21
                ldc2_w 8.0
                ddiv
                dstore 21
                aload 11
                dload 21
                d2i
                daload
                dstore 23
                dload 23
                dload 5
                dcmpl
                ifgt L11
L13:            dload 1
                dload 3
                dcmpl
                iflt L15
L16:            goto L2
L15:            dload 1
```

```
                ldc2_w 8.0
                dmul
                dstore 25
                dload 25
                ldc2_w 8.0
                ddiv
                dstore 25
                aload 11
                dload 25
                d2i
                daload
                dstore 7
L17:            dload 1
                ldc2_w 8.0
                dmul
                dstore 27
                dload 3
                ldc2_w 8.0
                dmul
                dstore 29
                dload 29
                ldc2_w 8.0
                ddiv
                dstore 29
                aload 11
                dload 29
                d2i
                daload
                dstore 31
                dload 27
                ldc2_w 8.0
                ddiv
                dstore 27
                aload 11
                dload 27
                d2i
                dload 31
                dastore
L18:            dload 3
                ldc2_w 8.0
```

```
                dmul
                dstore 33
                dload 33
                ldc2_w 8.0
                ddiv
                dstore 33
                aload 11
                dload 33
                d2i
                dload 7
                dastore
                goto L9
L2:             return
        .end code
.end method
.sourcefile 'Main.java'
.end class
```

6 Conclusão

Com o trabalho em questão foi possível entender e implementar um **tradutor** da linguagem **SmallL** para bytecodes **Java**(com sintaxe **Jasmin**).

No próximo trabalho, iremos promover a junção de todas as etapas construídas nos TPs 1,2 e 3. Com o auxílio da ferramenta de `assembler` do **Krakatau**, geraremos *bytecodes* binários que sejam passíveis de execução em uma JVM.

Referências

[1] J. Meyer and T. Downing, *Java virtual machine*. Cambridge, [Mass.] : O'Reilly, 1997, includes index.