Trabalho Prático IV

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

20 de Novembro de 2017

1 Introdução

O trabalho em questão tem como objetivo apresentar o compilador integrado, desenvolvido ao longo dos trabalhos anteriores (*front-end* e o tradutor). Para auxiliar a implementação(do tradutor) foram utilizadas as seguintes ferramentas:

- **Jasmin**[2]: sintaxe de bytecodes extendida.
- **Krakatau**: decompiler, disassembler e assembler para arquivos `.class` Java.

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`.

Esse relatório não visa explicar em detalhes cada componente do compilador, uma vez que isso já foi feito nos relatórios passados. Com isso, descreveremos como utilizar o compilador e rodar os programas gerados.

2 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.

2.1 Utilizando o compilador

Para facilitar a utilização do **compilador** foram criados três scripts *bash* que condensam as tarefas de compilar o código do *front-end*, executar o *front-end* e traduzir o código intermediário.

- `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`, gerando código intermediário.
- `translate.sh`: reponsável por traduzir os códigos intermediários obtidos após o rodar `execute.sh`, gerando os bytecodes (em formato `.j` e em formato `.class`).

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*.

```
// 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 `Main.class` dentro das pastas `nome_teste-bin/`), a partir dos códigos gerados pelo tradutor.

2.2 Rodando o código compilado

Para verificar o funcionamento de um programa compilado, basta invocar o comando `java` no terminal, passando como parâmetro o arquivo binário do programa (`.class`). Tal arquivo é gerado a partir da ferramenta `assembler` do **Krakatau**. Por exemplo:

```
// mude para o diretorio contendo binario de algum teste
cd smallL/outputs/translated/test1-bin/
// rode o binario (arquivo Main.class)
java Main
```

2.3 Comando *print*

Para facilitar a visualização do funcionamento do programa compilado e executado, adicionamos o comando print no conjunto de instruções do código intermediário. O comando print gera um código que imprime o nome da variável, seguido de dois pontos e o seu valor naquele momento.

Para utilizá-la, basta adicionar no código intermediário um novo comando com a diretiva *print* e um nome de variável (já utilizada anteriormente no código). Os exemplos a seguir mostram a utilização do comando.

Código 1: Codigo na linguagem SmallL.

```
{
    float i; float j;
    i = 1;
    j = 10;

    while (i < j) {
        i = i + 1;
    }
}
```

O código intermediário do exemplo acima é então gerado e o comando print é adicionado ("à mão", ou seja, o arquivo `.txt` é editado) abaixo dos *labels* L1 e L5:

Código 2: Código intermediário com adição do comando print.

```
L1: i = 1
    print i
L3: j = 10
L4: iffalse i < j goto L2
L5: i = i + 1
    print i
    goto L4
L2:
```

O código traduzido para bytecodes (Jasmin) contém o trecho relativo à chamada da função print:

Código 3: Trecho do código traduzido.

```
...

L2:    getstatic Field java/lang/System out Ljava/io/
    ↪ PrintStream;
       ldc 'i: '
       invokevirtual Method java/io/PrintStream print (Ljava/
           ↪ lang/String;)V
       getstatic Field java/lang/System out Ljava/io/
           ↪ PrintStream;
       dload 1
       invokevirtual Method java/io/PrintStream println (D)V
       return
```

O resultado após rodar o binário é o seguinte:

```
i: 1.0
i: 2.0
i: 3.0
i: 4.0
i: 5.0
i: 6.0
i: 7.0
i: 8.0
i: 9.0
i: 10.0
```

Ou seja, mostra corretamente o valor da variável i (ao longo da execução do programa).

O comando print também é capaz de reproduzir na saída o valor de alguma posição de array. Exemplo `print a [ 1 ]` ou `print a [ t1 ]` (com os espaços inclusos).

3 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.
4. saída do programa após rodar o binário gerado.

3.1 Teste 1

O teste abaixo visa testar as funcionalidades básicas da linguagem. Os prints adicionados no código intermediário visam obter o valor da variável i ao longo do primeiro `do-while`, bem como o valor da terceira posição do vetor a criado (contagem começando de zero) e o valor da quarta posição do vetor a ao longo do `while`.

Código 4: **Arquivo do teste 1 em linguagem SmallL**

```
{
	int i; int j; float[4] a;

	i = 1;
	j = 10;
	a[0] = 1;
	a[1] = 5;
	a[3] = 0;

	do i = i+1; while( i < j);

	a[2] = a[0] + a[1];

	i = 0;
```

```
        while ( i < j){
                i = i+1;
                a[3] = a[3] + 1;
        }

}
```

Código 5: **Arquivo com código intermediário para o teste 1 produzido pelo front-end (adicionado de comandos print "na mão")**

```
L1: i = 1
L3: j = 10
L4: t1 = 0 * 8
    a [ t1 ] = 1
L5: t2 = 1 * 8
    a [ t2 ] = 5
L6: t3 = 3 * 8
    a [ t3 ] = 0
L7: i = i + 1
    print i
L9: if i < j goto L7
L8: t4 = 2 * 8
    t5 = 0 * 8
    t6 = a [ t5 ]
    t7 = 1 * 8
    t8 = a [ t7 ]
    t9 = t6 + t8
    a [ t4 ] = t9
    print a [ t4 ]
L10: i = 0
L11: iffalse i < j goto L2
L12: i = i + 1
L13: t10 = 3 * 8
    t11 = 3 * 8
    t12 = a [ t11 ]
    t13 = t12 + 1
    a [ t10 ] = t13
    print a [ t10 ]
    goto L11
```

```
L2:
```

Código 6: **Código para o teste 1 em sintaxe Jasmin produzido pelo tradutor implementado**

```
.version 50 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 33

L1:           ldc2_w 1.0
              dstore 1
L3:           ldc2_w 10.0
              dstore 3
L4:           ldc2_w 0.0
              ldc2_w 8.0
              dmul
              dstore 5
              sipush 1000
              newarray double
              astore 7
              dload 5
              ldc2_w 8.0
              ddiv
              dstore 5
              aload 7
              dload 5
              d2i
              ldc2_w 1.0
```

```
                dastore
L5:             ldc2_w 1.0
                ldc2_w 8.0
                dmul
                dstore 9
                dload 9
                ldc2_w 8.0
                ddiv
                dstore 9
                aload 7
                dload 9
                d2i
                ldc2_w 5.0
                dastore
L6:             ldc2_w 3.0
                ldc2_w 8.0
                dmul
                dstore 11
                dload 11
                ldc2_w 8.0
                ddiv
                dstore 11
                aload 7
                dload 11
                d2i
                ldc2_w 0.0
                dastore
L7:             dload 1
                ldc2_w 1.0
                dadd
                dstore 1
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                ldc 'i: '
                invokevirtual Method java/io/PrintStream print (
                    ↪ Ljava/lang/String;)V
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                dload 1
                invokevirtual Method java/io/PrintStream println
```

```
                    ↪ (D)V
L9:             dload 1
                dload 3
                dcmpg
                iflt L7
L8:             ldc2_w 2.0
                ldc2_w 8.0
                dmul
                dstore 13
                ldc2_w 0.0
                ldc2_w 8.0
                dmul
                dstore 15
                dload 15
                ldc2_w 8.0
                ddiv
                dstore 15
                aload 7
                dload 15
                d2i
                daload
                dstore 17
                ldc2_w 1.0
                ldc2_w 8.0
                dmul
                dstore 19
                dload 19
                ldc2_w 8.0
                ddiv
                dstore 19
                aload 7
                dload 19
                d2i
                daload
                dstore 21
                dload 17
                dload 21
                dadd
                dstore 23
                dload 13
```

```
                ldc2_w 8.0
                ddiv
                dstore 13
                aload 7
                dload 13
                d2i
                dload 23
                dastore
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                ldc 'a[t4]: '
                invokevirtual Method java/io/PrintStream print (
                    ↪ Ljava/lang/String;)V
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                aload 7
                dload 13
                d2i
                daload
                invokevirtual Method java/io/PrintStream println
                    ↪ (D)V
L10:            ldc2_w 0.0
                dstore 1
L11:            dload 1
                dload 3
                dcmpg
                ifge L2
L12:            dload 1
                ldc2_w 1.0
                dadd
                dstore 1
L13:            ldc2_w 3.0
                ldc2_w 8.0
                dmul
                dstore 25
                ldc2_w 3.0
                ldc2_w 8.0
                dmul
                dstore 27
                dload 27
```

```
                ldc2_w 8.0
                ddiv
                dstore 27
                aload 7
                dload 27
                d2i
                daload
                dstore 29
                dload 29
                ldc2_w 1.0
                dadd
                dstore 31
                dload 25
                ldc2_w 8.0
                ddiv
                dstore 25
                aload 7
                dload 25
                d2i
                dload 31
                dastore
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                ldc 'a[t10]: '
                invokevirtual Method java/io/PrintStream print (
                    ↪ Ljava/lang/String;)V
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                aload 7
                dload 25
                d2i
                daload
                invokevirtual Method java/io/PrintStream println
                    ↪ (D)V
                goto L11
L2:             return

        .end code
.end method
.sourcefile 'Main.java'
```

```
.end class
```

Código 7: **Saída após rodar o binário**

```
i: 2.0
i: 3.0
i: 4.0
i: 5.0
i: 6.0
i: 7.0
i: 8.0
i: 9.0
i: 10.0
a[t4]: 6.0
a[t10]: 1.0
a[t10]: 2.0
a[t10]: 3.0
a[t10]: 4.0
a[t10]: 5.0
a[t10]: 6.0
a[t10]: 7.0
a[t10]: 8.0
a[t10]: 9.0
a[t10]: 10.0
```

Como podemos observar, os valores relacionados estão corretos. Em relação ao print da posição do vetor, é possível identificar variáveis temporárias (t4 e t10) indexando os vetores. Tais variáveis foram criadas pelo tradutor em tempo de tradução, mas que indicam respectivamente a teceira e a quarta posição do vetor a.

Vale ressaltar aqui que, nesse caso, o valor das variáveis temporárias havia sido multiplicado por 8 para indexar no código intermediário. No entanto, isso não funcionaria na *JVM*, e portanto, quando usado para indexar, o tradutor atualiza essa variável dividindo-na 8. Sendo assim, como o comando `print a [ t4 ]` vem após a variável temporária já ter sido usada como índice, ele é equivalente a `print a [ 2 ]` e não `print a [ 16 ]`. Portanto, para printarmos na tela as posições do array, devemos indexar normalmente, e não utilizando múltiplos de 8.

3.2 Teste 2

O teste a seguir visa abordar mais algumas funcionalidades básicas da linguagem. Os prints adicionados ao código intermediário tem o intuito de obter o valor de i e y ao longo do `while` e do valor da variável x antes do término do `while` (ao entrar na condição do if) e após o término.

Código 8: **Arquivo para o teste 2 em linguagem SmallL**

```
{
       int i; int k; float x; float y;

       i = 0;
       k = 10;
       x = 2;
       y = 0;

       while (i < k){

              if (i >= 5){
                     x = x + 1;
                     break;
              }

              i = i + 1;
              y = y + 1;
       }

       x = x + y;

}
```

Código 9: **Arquivo com código intermediário para o teste 2 produzido pelo front-end (adicionado de comandos print "na mão")**

```
L1: i = 0
L3: k = 10
L4: x = 2
L5: y = 0
L6: iffalse i < k goto L7
L8: iffalse i >= 5 goto L9
L10: x = x + 1
```

```
    print x
L11: goto L7
L9: i = i + 1
   print i
L12: y = y + 1
    print y
    goto L6
L7: x = x + y
   print x
L2:
```

Código 10: **Código em sintaxe Jasmin para o teste 2 produzido pelo tradutor implementado**

```
.version 50 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 9

L1:           ldc2_w 0.0
              dstore 1
L3:           ldc2_w 10.0
              dstore 3
L4:           ldc2_w 2.0
              dstore 5
L5:           ldc2_w 0.0
              dstore 7
L6:           dload 1
              dload 3
```

```
                dcmpg
                ifge L7
L8:             dload 1
                ldc2_w 5.0
                dcmpl
                iflt L9
L10:             dload 5
                ldc2_w 1.0
                dadd
                dstore 5
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                ldc 'x: '
                invokevirtual Method java/io/PrintStream print (
                    ↪ Ljava/lang/String;)V
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                dload 5
                invokevirtual Method java/io/PrintStream println
                    ↪ (D)V
L11:             goto L7
L9:             dload 1
                ldc2_w 1.0
                dadd
                dstore 1
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                ldc 'i: '
                invokevirtual Method java/io/PrintStream print (
                    ↪ Ljava/lang/String;)V
                getstatic Field java/lang/System out Ljava/io/
                    ↪ PrintStream;
                dload 1
                invokevirtual Method java/io/PrintStream println
                    ↪ (D)V
L12:             dload 7
                ldc2_w 1.0
                dadd
                dstore 7
                getstatic Field java/lang/System out Ljava/io/
```

```
                ↪ PrintStream;
               ldc 'y: '
               invokevirtual Method java/io/PrintStream print (
                ↪ Ljava/lang/String;)V
               getstatic Field java/lang/System out Ljava/io/
                ↪ PrintStream;
               dload 7
               invokevirtual Method java/io/PrintStream println
                ↪ (D)V
               goto L6
L7:            dload 5
               dload 7
               dadd
               dstore 5
               getstatic Field java/lang/System out Ljava/io/
                ↪ PrintStream;
               ldc 'x: '
               invokevirtual Method java/io/PrintStream print (
                ↪ Ljava/lang/String;)V
               getstatic Field java/lang/System out Ljava/io/
                ↪ PrintStream;
               dload 5
               invokevirtual Method java/io/PrintStream println
                ↪ (D)V
L2:            return

       .end code
.end method
.sourcefile 'Main.java'
.end class
```

Código 11: **Saída após rodar o binário**

```
i: 1.0
y: 1.0
i: 2.0
y: 2.0
i: 3.0
y: 3.0
i: 4.0
```

```
y: 4.0
i: 5.0
y: 5.0
x: 3.0
x: 8.0
```

Como podemos observar, os valores estão de acordo com o esperado.

4 Conclusão

A partir do desenvolvimento dos trabalhos anteriores e da finalização por meio deste, foi possível compreender como funciona um compilador em todas as possíveis partes internas.

Conseguimos implementar tanto o *front-end* [1] quanto uma versão simples do *back-end*(tradutor) do compilador de forma prática e eficaz, para uma gramática relativamente parecida com a de uma linguagem de programação amplamente utilizada (C ou Java).

Pudemos aprender sobre o funcionamento das análises léxica e sintática, geração de código intermediário e sobre a geração de *bytecodes* via tradução, a partir do código intermediário.

Por fim, foi possível integrar todas as partes desenvolvidas com sucesso e também disponibilizar o código para qualquer pessoas que deseja utilizar essa implementação.

Referências

[1] A. V. Aho, *Compilers: principles, techniques, and tools*, 2007, vol. 2.

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