Avaliação e Trabalhos de Laboratório de Compiladores - ENPE 2020/2

José de Oliveira Guimarães
Departamento de Computação
UFSCar - Sorocaba, SP
Brasil
e-mail: josedeoliveiraguimaraes@gmail.com

February 22, 2021

Este documento descreve os trabalhos das disciplinas Laboratório de Compiladores, código 1001308 do currículo 2018/1 (4 créditos) e Laboratório de Compiladores, código 480665 do currículo 2011/1. Veja o plano de ensino para saber como é calculada a nota final.

A avaliação de ambas as disciplinas consiste de duas provas (NP1 e NP2) e dois trabalhos (NT1 e NT2). Os trabalhos podem ser individuais ou em grupos de dois. A prova é individual. A data e hora de entrega do primeiro trabalho, para ambas as disciplinas, é dia 26 de abril às 18h. A data e hora para o segundo trabalho é 07 de junho às 18h. A prova correspondente será aplicada na mesma data que a entrega do trabalho.

Os que ficarem com nota final abaixo de 6 terão a oportunidade de fazer uma recuperação entre o dia que for divulgado o resultado da nota da prova do dia 7 de junho e o dia 21 de junho. Esta recuperação consiste do mesmo trabalho mais uma nova prova.

1 Primeiro Trabalho para Alunos da Disciplina 1001308

O primeiro trabalho consiste da construção de:

(a) um analisador sintático e semântico da linguagem Cianeto. Esta linguagem é descrita no texto "The Cianeto Language". Não é necessário implementar campos e métodos *shared*;

(b) um gerador de código de Cianeto para Java utilizando métodos da Árvore de Sintaxe Abstrata (ASA). Deve haver um método

```
public void genJava(PW pw)
```

na classe `Program` da ASA (já está assim no compilador fornecido pelo professor). E métodos com este mesmo nome nas outras classes da ASA. A classe principal do arquivo Java, a única pública, deve ter o mesmo nome que o arquivo. Se o nome do arquivo é "`OK_GER01.ci`", deve haver a seguinte classe no arquivo Java gerado:

```
public class OK_GER01 {
    public static void main(String []args) {
        new Program().run();
```

```
        }
    }
```

Para testar a geração de código em Java de um arquivo

`C:\Dropbox\OK_GER01.ci`

chame o seu compilador com os parâmetros

```
"C:\Dropbox\OK_GER01.ci" -genjava "C:\Dropbox"
```

O arquivo Java produzido será colocado em `C:\Dropbox`, compilado e executado.

Os arquivos que testam se o código foi gerado corretamente tem nomes `OK_GERxx.ci` no qual `xx` é `01`, `02` etc. Estes arquivos foram feitos de tal forma que, ao serem executados, a primeira linha que eles produzem como saída é igual ao restante da saída. A menos de espaços e linhas em branco. Isto é, se a saída for

```
4 1 2  3    4
4
1    2
3      4
```

então isto quer dizer que o compilador produziu código correto para este arquivo. Mas se a saída for

```
4 1 2  3    4
41234
```

então o código produzido está incorreto — a segunda linha não possui os espaços que existem na primeira linha.

Quando o compilador de Cianeto for usado com a opção `-genjava`, ele irá chamar o método `genJava` do programa, gerar um arquivo Java, compilá-lo (com `javac` e interpretá-lo (com o interpretador de Java). E irá, pela saída do programa, verificar se ele está gerando código corretamente. Um relatório chamado "`report.txt`" será produzido no diretório que se segue à opção `-genjava`. Este relatório conterá todos listas com arquivos que não compilaram, que geraram código incorreto etc.

2 Segundo Trabalho para Alunos da Disciplina 1001308

Faça a geração de código de Cianeto para C. Siga precisamente as instruções do texto "Geração de Código em C para Cianeto".

3 Primeiro Trabalho para Alunos da Disciplina 480665

Este trabalho consiste de 60% do item (a) do primeiro trabalho da disciplina 1001308, que é

> um analisador sintático e semântico da linguagem Cianeto. Esta linguagem é descrita no texto "The Cianeto Language". Não é necessário implementar campos e métodos *shared*.

Isto é, espera-se que, ao final deste trabalho, o compilador passe em 60% dos testes relativos à análise semântica. O aluno ou grupo é livre para escolher quais testes serão implementados.

4 Segundo Trabalho para Alunos da Disciplina 480665

Este trabalho consiste em terminar a análise semântica da linguagem (item (a) do primeiro trabalho da disciplina 1001308) e fazer o item (b) do primeiro trabalho da disciplina 1001308, que é

> um gerador de código de Cianeto para Java utilizando métodos da Árvore de Sintaxe Abstrata (ASA). Deve haver um método
>
> ```
> public void genJava(PW pw)
> ```

5 Outras Observações

Para cada trabalho, submeta ao AVA um único arquivo zip cujo nome é Nome1-Nome2.zip, sendo Nome1 e Nome2 os nomes completos dos integrantes do grupo, em ordem alfabética. Não use acentos nos nomes, omita algum sobrenome se achar necessário. O conteúdo do arquivo Nome1-Nome2.zip deve ser um diretório 'src' com o código fonte do compilador. Coloque apenas os arquivos `*.java` nos diretórios apropriados. Em resumo, não envie nenhum arquivo `*.class`, `*.pdf`, `*.txt`, testes, executáveis etc.

Use a classe Comp.java que está no AVA sem **nenhuma** alteração. Esta é a classe principal do compilador, que chama o método `compile` da classe `Compiler`. Para chamar o compilador, digite algo como

```
C:\Dropbox\19-2\LabComp\cianeto\bin>java -cp .
        comp.Comp "C:\Dropbox\19-2\LC\tests"
```

O compilador, quando chamado com um diretório como argumento, compila todos os arquivos `*.ci` do diretório e produz um relatório chamado "report.txt". O compilador também pode ser chamado com um único arquivo.

Até 5% de código do compilador pode ser copiado de outros grupos. Mas é necessário especificar exatamente qual trecho e de qual grupo ele foi copiado em um arquivo "trechos-copiados.txt". Este arquivo deve estar no diretório principal do arquivo zip com o compilador. Estes 5% se referem aos trechos feitos por você, não ao total do compilador, parte dele foi fornecida no AVA. Importante: **os métodos de análise sintática e semântica de expressões NÃO podem ser copiados**. Isto é, os métodos que fazem a análise da regra `Expression` da gramática não podem ser copiados. Obviamente inclui todos os métodos que analisam expressões.

O seu compilador deve ter obrigatoriamente as características descritas abaixo.

(1) O trabalho deve ser feito em Java e a classe principal deve ser a fornecida no AVA. Você pode usar ou não o restante do compilador fornecido pelo professor. Mas é imprescindível emitir os erros usando os métodos que já estão prontos, que usam as classes `Compiler`, `CompilerError` etc.

(2) **Todos os arquivos devem ter um comentário inicial com o nome dos integrantes do grupo**. Todos os arquivos (repetindo). Sem estes comentários, o trabalho não será considerado.

(3) Use codificação Cp1252 para os arquivos, que é o geralmente usado no IDE Eclipse para Windows. Muitos trabalhos feitos nos computadores da Apple não respeitam esta restrição. Então converta os arquivos para Cp1252 antes de comprimi-los e colocá-los no AVA. Se você usar um editor convencional, os arquivos já estarão em Cp1252 ou em uma codificação que não causa erros de compilação.

(4) **NÃO** copie regras da gramática do pdf "The Cianeto Language" e cole-as no seu código fonte (em geral, no arquivo "Compiler.java"). Isto causa erros de compilação quando se usa o compilador `javac` da linha de comando porque acontecem erros na codificação das aspas empregadas no pdf. Você pode perder pontos por fazer isto.

Você pode fazer perguntas por email nesta disciplina. Ou marcar um encontro virtual com o professor.

Apêndices

A O Relatório do Compilador

Chame o compilador com o diretório de testes. Ele produzirá um relatório "`report.txt`" no diretório corrente. Este arquivo inicia-se com um resumo das compilações dos arquivos do diretório. Algo assim:

```
------------------------------------------------------------------------
MI:  25            I: 11             PI: 29            Exc: 9
Dev: 72/130/55%    LE: 15/130/11%    SSE: 24/70/34%

MI = muito importante, I = importante, PI = pouco importante, Exc = exceções
Dev = deveria ter sinalizado, LE = sinalizou linha errada, SSE = sinalizado sem erro
------------------------------------------------------------------------
```

`MI` é o número de testes em que o compilador falhou e que são considerados muito importantes.

`I` é o número de testes em que o compilador falhou e que são considerados importantes.

`PI` é o número de testes em que o compilador falhou e que são considerados pouco importantes (mas são importantes!).

`Dev` é seguido de três números. O primeiro é o número de testes em que o compilador deveria ter sinalizado erro mas não o fez (se o compilador está correto, este número deve ser `0`). O segundo é o número de testes em que há erro; isto é, do diretório que você passou na linha de comando para o compilador, há, por exemplo, NT casos de testes. Destes, há NE com uma anotação `cep` indicando que há um erro naquele arquivo (veja exemplo abaixo) e que o compilador deve sinalizar. Este segundo número é NE (portanto, este número depende dos arquivos com os testes, não depende de você). O terceiro número é a porcentagem do primeiro número em relação ao segundo. Isto é, que porcentagem dos erros que o seu compilador não detectou.

Um exemplo de um arquivo de texto, `ER_SEM13.ci`, é dado abaixo.

```
@annot("check", "typeErrorSearchMethod")
@cep(8, "Não há método readBoolean", "Unknown method 'readBoolean'")


class Program

   func run {
      var Boolean b;

      b = In.readBoolean;
   }

end
```

`LE` é seguido de três números.O primeiro é o número de testes em que o compilador apontou o erro, mas na linha errada. O segundo é o número de testes em que há erro (igual ao número de `Dev`). O terceiro número é a porcentagem do primeiro número em relação ao segundo.

`SSE` é seguido de três números. O primeiro é o número de testes em que o compilador sinalizou erros mas que não possuem erros (em um compilador correto, este primeiro número deve ser `0`). O segundo é o número de testes em que **não** há erro (um arquivo sem erro possui uma anotação `nce` (veja próximo exemplo). Então este é o número de arquivos com anotação `nce`). O terceiro número é a porcentagem do primeiro número em relação ao segundo.

`Exc` é seguido do número de vezes em que o compilador lançou uma exceção — portanto, há um erro no compilador.

Idealmente, a saída deveria ser assim:

```
------------------------------------------------------------------------
MI:  0            I: 0             PI: 0            Exc: 0
Dev: 0/130/0%    LE: 0/130/0%    SSE: 0/70/0%

MI = muito importante, I = importante, PI = pouco importante, Exc = exceções
Dev = deveria ter sinalizado, LE = sinalizou linha errada, SSE = sinalizado sem erro
------------------------------------------------------------------------
```

Um arquivo sem erros é dado a seguir.

```
@annot("check", "openAsIdentifier")
@nce
/*
  @filename   ok-sem11.ci
  @comment    Testa se o compilador aceita metodo com nome 'open',
              que não é uma palavra-chave
  */
class A
   public func open {
   }
end

class Program

   func run {
      var A a;
      a = A.new;
      a.open;
   }

end
```

B Dicas Sobre o Trabalho

A classe que representa uma variável local pode ser, inicialmente,

```
public class Variable {
    private String name;
```

```
    private Type type;
}
```

Como `type` é do tipo `Type`, este campo (variável de instância) pode apontar para objetos de `Type` e suas subclasses, o que inclui `TypeCianetoClass`. Assim, o tipo de uma variável pode ser "`Int`" (objeto de `Type`), "`Boolean`" (objeto de `Type`), "`String`" (objeto de `Type`) ou uma classe (objeto de `TypeCianetoClass`). Naturalmente, `TypeCianetoClass` deve herdar de `Type` para que isto seja possível.

O construtor de `TypeCianetoClass` deve ter um único parâmetro, o nome da classe. Assim, pode-se criar um objeto de `TypeCianetoClass` tão logo saibamos o nome da classe. Isto é necessário pois o objeto que representa a classe deve logo ser inserido na Tabela de Símbolos, pois uma classe pode declarar um objeto dela mesma:

```
class A
   var A x
   ...
end
```

Assim, ao encontar o "`x`", haverá uma busca na tabela de símbolos e lá será encontrado o objeto de `TypeCianetoClass` que representa a classe `A`, que foi inserido lá tão logo o nome da classe se tornou disponível. A mesma observação vale para a classe que representa um método, podemos ter chamadas recursivas.

Ao encontrar um comando

```
x = y;
```

o compilador deve procurar por `x` na tabela de símbolos de tal forma que o objeto de `AssignmentStatement`[1] correspondente a esta atribuição tenha um ponteiro para o objeto `Variable` representando x. Supõe-se que `Variable` é a classe que representa uma variável (local, campo, ou parâmetro, sendo que deve haver subclasses de `Variable` para cada uma destas categorias). Este objeto é o que foi criado na declaração de `x`. O mesmo se aplica a `y`. Você deve fazer algum assim:

```
// lexer.getStringValue() retorna "x"
Variable left = symbolTable.searchVariable( lexer.getStringValue() );

if ( left == null ) { error.show("..."); }

return new AssignmentStatement( left, expr() );
```

Este código assume que existe um método `searchLocalVariable`, que pesquisa por uma variável local (inclusive parâmetros), na tabela de símbolos.

Duas estratégias ERRADAS são dadas abaixo.

1. representar `x` como String. A classe `AssignmentStatement` seria

```
class AssignmentStatement {
   private String    leftSide;
   private Expr      rightSide;
   ...
}
```

[1] Suponha que esta seja a classe que representa uma atribuição. Ela não foi fornecida, crie algo assim, não necessariamente com este nome.

2. representar `x` como uma variável, mas criar esta variável ao encontrar `x`:

```
// lexer.getStringValue() retorna "x"

Variable left = new Variable( lexer.getStringValue() );

return new AssignmentStatement( left, expr() );
```

Os únicos lugares onde deve-se criar objetos representanto variáveis locais, campos (variáveis de instância) e parâmetros é na declaração das variáveis correspondentes. E nunca se deve representar variáveis por strings — utilize objetos de `Variable` (não necessariamente com este nome) e suas subclasses. O mesmo se aplica a `TypeCianetoClass` e ao tipo de variáveis. Isto é, um tipo deve ser um ponteiro para um objeto de `Type` e subclasses, não uma string.

Os compiladores sinalizam uma exceção depois de mostrar um erro. Esta exceção deve ser (é) capturada em um bloco try no método compile da classe Compiler. A impressão da pilha de chamadas (com `e.printStackTrace()`) deve ser utilizada apenas na fase de depuração do compilador e não deve estar presente no compilador entregue ao professor

É quase obrigatório que alguma variável referencie o objeto que representa a classe atualmente sendo compilada. Esta variável deve ser um campo de `Compiler` e será largamente usada. Por exemplo, para inserir métodos e campos na classe corrente de Cianeto. Idem para o método atualmente sendo compilado.

Para gerar código Java de `In.readInt` e `In.readString`, é melhor gerar uma classe com métodos estáticos "`int readInt()`" e "`String readString()`". O nome da classe dever ser diferente de qualquer nome válido em Cyan (como fazer isto?). A geração de código Java para "`Out.print:`" e "`Out.println:`" pode ser feita com "`System.out.println(...)`".

Utilize a classe `PW` para fazer a tabulação do código gerado corretamente. Esta classe está no pacote `ast`. O construtor toma um objeto `PrintWriter` utilizado para saída

```
pw = new PW(out);
```

sendo `out` um objeto de `PrintWriter`. Na verdade, isto já é feito pelo compilador. Você nunca precisará criar um objeto `PW` ou criar um arquivo. Utilize a infraestrutura fornecida com o compilador.

`pw` possui métodos `printIdent` e `printlnIdent` que automaticamente indentam o que você imprime com `pw.printIdent` ou `pw.printlnIdent`. Naturalmente, `printlnIdent` imprime a string e pula para a próxima linha, enquanto que `printIdent` não. Se você quiser aumentar a indentação, utilize "`pw.add()`". Para diminuir, utilize "`pw.sub()`". Teste o seguinte código

```
PrintWriter out = new PrintWriter( ... );
  /* faça isso uma única vez antes do início da geração de código,
     o que já é feito pelo compilador fornecido. */
pw = new PW(out);

pw.add();
pw.printIdent("a = a + 1;");
pw.add();
pw.printIdent("if");
pw.println( "  a > b");
// código sem indentação, pois foi escrito com pw.print
// e não pw.printIdent
```

```
   pw.printIdent("then");
   pw.add();    //  comandos dentro do then devem ser indentados
   pw.printlnIdent("a = b;");
   pw.sub();    // diminui a indentação: acabou o then
   pw.printlnIdent("endif");
   pw.sub();
   pw.println("Texto normal, não indentado");
   pw.printlnIdent("Identado");
   pw.sub();
```

A saída deste código é

```
   a = a + 1;
   if a > b
   then
      a = b;
   endif
Texto normal, não indentado
   Identado
```