Geração de Código
Parte II - Roteiro para atividades

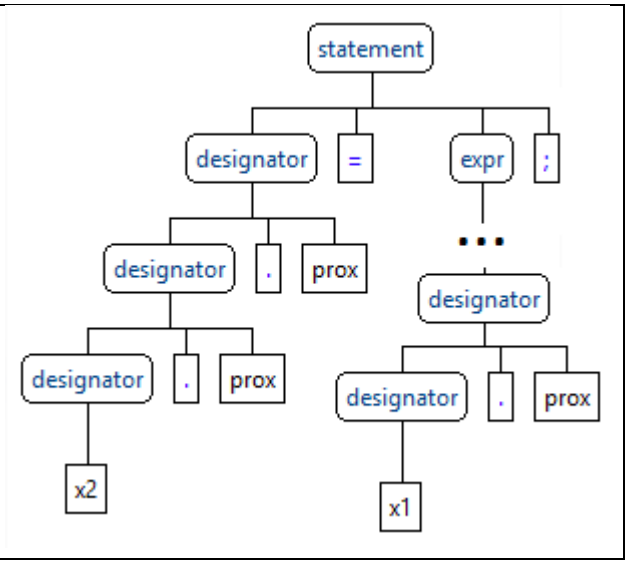
Executando este roteiro, todas as tarefas para geração de código do compilador estarão completadas.

Implementaremos a geração de código como um *listener*, usando a ferramenta ANTLR4. Como sabemos, os métodos de um listener são chamados automaticamente quando os nós da árvore de sintaxe são visitados. No caso da geração de código, cada método deverá depositar em uma pilha global o resultado de sua execução, que deverá ser uma String representando o código *assembly* associado ao nó da árvore de sintaxe. Para isso, cada método poderá assumir que a visita dos nós das subárvores associadas deverá deixar na pilha os resultados intermediários da sua geração de código. O método então deverá desempilhar esses resultados intermediários, combiná-los e empilhar uma nova String que represente o código que deve gerar. Veja o exemplo a seguir.

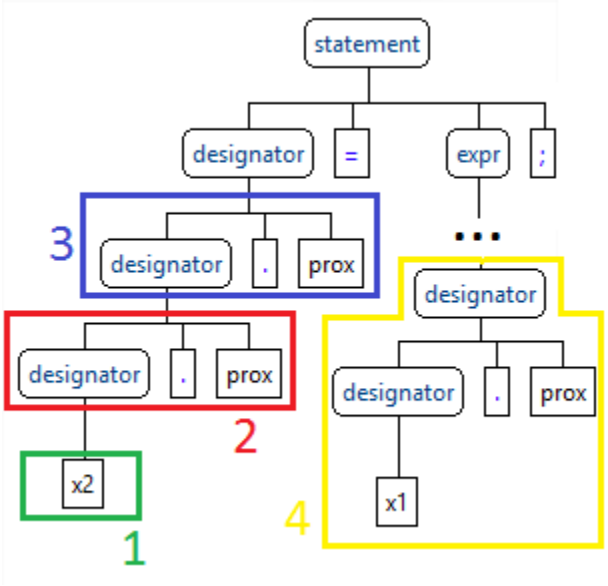
No lado esquerdo da figura abaixo, é apresentado um trecho de código MicroJava com um comando de atribuição destacado. Usando a gramática que construímos em aulas anteriores, é apresentada, no lado direito da mesma figura, uma árvore de sintaxe que representa esse comando de atribuição.

```
class C {
  int v;
  C prox;
}

...
void f()
C x1, x2;
{
...
  x2.prox.prox = x1.prox;
```



Para gerar código para o comando de atribuição, um listener irá visitar recursivamente, em profundidade, cada subárvore da estrutura. Como explicado acima, cada método deverá deixar na pilha o resultado da geração de código da árvore associada, na forma de uma String. Veja a seguir os passos esperados, para o exemplo acima.



O listener "desce" seguidamente pela estrutura até encontrar o nó marcado como **1** na figura. Ao visitar esse nó, que constitui um identificador simples, o listener irá empilhar código para ler o valor de x2:

```
load 1
```

Retornando ao nível acima, ao visitar o nó marcado como **2** na figura, o listener espera que o código gerado para o designador (subárvore esquerda) esteja na pilha. Sua tarefa então é desempilhar esse código, adicionar instrução para obter o campo `prox` e empilhar o código resultante. A configuração da pilha será então:

```
load 1
getfield 1
```

OBS: A pilha representada acima tem apenas 1 elemento, que é um código com 2 instruções. Para representar isso em forma de String, iremos usar "fim de linha" para separar as instruções ("`load 1\ngetfield 1`").

Quando for terminar de visitar então o nó identificado como **3** na figura, o listener irá verificar uma situação diferente: o pai desse nó é um comando de atribuição. Isso significa que a operação em **3** é a última a ser executada do lado esquerdo de uma atribuição. Quando estudamos o código assembly MicroJava, aprendemos que essa instrução deverá ser gerada **após** o código associado ao lado direito da atribuição. Assim, o listener irá empilhar o código associado ao nó **3**, sem ainda combiná-lo com o que já está na pilha:

```
putfield 1
```
```
load 1
getfield 1
```

Em seguida, o listener irá visitar o lado direito da atribuição, representado na figura por 4. Essa visita deverá deixar mais um trecho de código na pilha:

```
load 0
getfield 1
```
```
putfield 1
```
```
load 1
getfield 1
```

Finalmente, quando o listener visita o nó na raiz da árvore que representa o comando de atribuição, ele deve esperar que no topo da pilha estejam 3 trechos de código correspondentes aos seus subcomponentes. A sua tarefa então será desempilhar esses códigos e empilhar um resultado com a ordem correta:

```
load 1
getfield 1
load 0
getfield 1
putfield 1
```

Com esse exemplo, deve ficar claro todo o trabalho que iremos fazer ao implementar o listener para geração de código:

- cada método deverá desempilhar os trechos de código correspondentes aos subcomponentes da árvore sendo visitada; em seguida,
- combinar esses trechos de acordo com o tipo de comando sendo processado; e finalmente,
- empilhar um resultado que represente o código a ser gerado para toda aquela árvore.

Neste tutorial, iremos ainda utilizar mais uma ferramenta para auxiliar na geração de código: a ferramenta ***StringTemplate***.

StringTemplate permite gerar textos a partir de esquemas (*templates*) pré-definidos. Os esquemas podem ser parametrizados e definem o layout desejado para o texto final.

Por exemplo, observe um esquema para gerar código para comando de atribuição, como o que vimos no exemplo discutido neste documento:

```
assign(read, write, exp) ::= <<
<read>
<exp>
<write>
>>
```

O nome do esquema é `assign` - esse nome serve para diferenciar cada esquema em um arquivo com vários esquemas. Os parâmetros são `read`, `write` e `exp`. O texto final deverá

ter o formato que é apresentado entre `<< ... >>`, que pode combinar os parâmetros (definidos entre `<...>`) com quaisquer outros caracteres, incluindo formatação. O exemplo acima indica que o texto `read` deverá vir primeiro, o texto `exp` virá na linha seguinte, e mais uma linha é adicionada antes de se completar o texto com `write`.

Código em alto nível para o método do listener que gera código para comando de atribuição:

```
void exitAssign(...) {
  // desempilha os códigos dos subcomponentes:
  String s1 = desempilha();
  String s2 = desempilha();
  String s3 = desempilha();
  // define os parâmetros do template:
  Template t = carrega template "assign";
  t.add("read", s3);  // define que o parâmetro read terá valor s3
  t.add("write", s2); // define que o parâmetro write terá valor s2
  t.add("exp", s1);   // define que o parâmetro exp terá valor s1
  // empilha o resultado do texto construído pelo template:
  empilha( t.render() );
}
```

Vantagens de se usar StringTemplate:

- O formato do texto associado ao código de cada comando fica mais claro, observando-se a especificação do template.
- O formato dos textos associados aos códigos gerados fica separado do programa que controla a geração de código. Isso pode permitir alterar o tipo de código gerado, sem ter que alterar o programa (listener), bastando carregar outro arquivo com especificações diferentes para os templates.

Veja outro exemplo a seguir.
O código MJ assembly para um método tem o seguinte formato geral:

```
label <nome do método>
enter <número de parâmetros>, <número total de variáveis>
<... instruções associadas ao código do método>
exit
return
```

Esse formato geral pode ser representado por um template de StringTemplate com a forma:

```
method(name, params, vars, instructions) ::= <<

label <name>
enter <params>, <vars>

<instructions>

exit
return
>>
```

Observe que o template tem nome *method* e parâmetros *name, params, vars, instructions*.

StringTemplate permite que seja carregado um grupo de vários templates a partir de um arquivo. Essa abordagem facilita o procedimento que mencionamos anteriormente, de poder trocar um template (na realidade todo o grupo de templates que trabalha em conjunto) e assim gerar um código diferente, sem ter que alterar o programa que está realizando a geração de código.

A figura a seguir apresenta código Java completo com uso de um template. Suponha que o arquivo "MJ.stg" esteja disponível e contenha o template "method" especificado como acima.

```
// carrega grupo de templates
STGroup group  = new STGroupFile("templates/MJ.stg");

ST st = group.getInstanceOf("method"); // carrega template de nome method

st.add("name", "func1"); // define parâmetro name com valor func1
st.add("params", 2);     // define parâmetro params com valor 2
st.add("vars", 3);       // define parâmetro vars com valor 3
// define parãmetro instructions com instruções em linhas separadas
st.add("instructions",
            "load0\n"
            + "load1\n"
            + "sub\n"
            + "store2\n"
            + "load2");

System.out.println(st.render()); // imprime string construída
```

O código acima irá imprimir:

```
label func1
enter 2, 3

load0
load1
sub
store2
load2

exit
return
```

INSTRUÇÕES :

Para executar as tarefas deste tutorial:

1. Obtenha e instale o projeto Eclipse GeracaoCodigo-roteiro.
2. Abra o arquivo Piloto.java e observe:
    a. No método *main*, define-se que o arquivo de entrada é "input/teste01.txt".
    b. No método *testinput*, são definidos e aplicados ao arquivo de entrada um analisador léxico/sintático, um listener para análise semântica e finalmente um listener para geração de código.
3. Abra *input/teste01.txt* e observe o programa que está sendo usado como teste.
4. Execute as instruções apresentadas no arquivo ***CodeListenerGenerator.java***.
5. Entregue pelo sistema do Google Classroom o arquivo ***CodeListenerGenerator.java*** e o arquivo de *templates* (***MJ.stg***) com as modificações solicitadas.