INF 441 - Exercícios Preparatórios para Prova 2

Para associar ações a uma gramática, duas estratégias foram utilizadas na disciplina INF441:

- **Esquemas de tradução**: as ações são trechos de código inseridos entre os símbolos do lado direito de uma gramática.
- **Uso de padrões de projeto *Visitor* ou *Listener***: percorrem a árvore de sintaxe e executam ações antes ou depois de visitar nós específicos da árvore.

O uso de padrões como Visitor e Listener tem como vantagem, quando comparado a esquemas de tradução, o fato de não "poluir" a gramática original. A gramática é a especificação da sintaxe da linguagem, e é desejável que essa sintaxe possa ser visualizada de forma mais clara possível. Nesse sentido, ações semânticas inseridas entre os símbolos podem comprometer essa clareza. Além disso, se for necessário processamento em mais de um passo (por exemplo, análise semântica e depois geração de código), é necessário ter mais de uma versão da gramática com ações diferentes. E ter mais de uma versão da gramática pode favorecer ocorrência de inconsistências.

Uma desvantagem do uso de Listener quando comparado a esquemas de tradução é a restrição de se inserir ações apenas antes ou depois de visitar um nó da árvore de sintaxe. Em esquemas de tradução, ações podem ser inseridas em qualquer posição entre os símbolos do lado direito de uma gramática.

Neste exercício, vamos utilizar uma estratégia baseada no padrão Listener para executar ações sobre uma gramática. Para que os métodos do Listener possam comunicar-se trocando informações, será utilizada uma pilha global. Cada método consulta na pilha as informações deixadas por outros métodos, e depois empilha um resultado do seu próprio processamento.

A gramática de uma linguagem denominada SIMPLE é apresentada na Figura 1, abaixo. O símbolo NUM representa uma constante inteira e ID representa um identificador. A Figura 2 apresenta um exemplo de programa escrito na linguagem SIMPLE. São permitidas apenas variáveis globais de tipo inteiro, e não há funções ou outros tipos de subprogramas.

```
p -> v b
v -> 'var' ID+
b -> '{' s* '}'
s -> a ';' | w ';' | i | b
a -> ID '=' e
w -> 'print' e
i -> 'if' '(' c ')' s
c -> e x e
x ->  '>=' | '<='
e -> f | t
f -> e y t
y -> '+' | '-'
t -> t1 | t2
t1 -> NUM
t2 -> ID
```

Fig.1: Gramática da linguagem SIMPLE.

```
var x y z
{
  x = 1;
  if (x >= 0)
    y = 2;
  z = k;
  print z + 1;
}
```

Fig.2: Exemplo de programa.

Vamos apresentar, em seguida, um Listener que implementa tradução de programas da linguagem SIMPLE para uma forma simplificada da linguagem assembly MicroJava estudada na disciplina INF441 (as instruções são quase as mesmas, assume-se são conhecidas). Para auxiliar na tradução, vamos usar uma estrutura simples para representar a tabela de símbolos, permitindo mapeamento entre nomes de variáveis e posição na memória, e oferecendo serviços para geração de rótulos.

```
class TabSimb {

  private Hashtable<String, Integer> map =
    new Hashtable<String, Integer>();

  private int contVar = 0;

  private int contLabel = 0;

  public void addSymbol(String s) { map.put(s, contVar); ++contVar; }

  public int getSymbol(String s) {
    Integer r = map.get(s); if (r == null) return -1; else return r; }

  public String genLabel() { ++contLabel; return "L" + contLabel; }

}
```

Um Listener para fazer tradução de programas da linguagem Simple é apresentado a seguir, executando análise semântica e geração de código em um único passo. É usada a convenção adotada pela ferramenta ANTLR: os métodos têm nomes no formato enter<NAME> ou exit<NAME>, onde <NAME> é o nome de um símbolo não terminal da gramática. Árvores de sintaxe são visitadas automaticamente por Listeners, em profundidade, da esquerda para a direita. Um método enter<NAME> é chamado assim que o caminhamento na árvore chega a um nó associado ao símbolo <NAME>. Um método exit<NAME> é chamado após o caminhamento ter visitado todos os subcomponentes de um nó associado ao símbolo <NAME>.

O código do Listener procura uma independência com relação a detalhes de implementação da árvore de sintaxe. Isso pode ser visto em diversos pontos, como por exemplo, ao obter a lista com os nomes das variáveis no método “exitV”, ou ao obter o nome da variável no método “exitA”. Nos exercícios a seguir, quando for necessário estender ou modificar o Listener, você pode usar linguagem algorítmica nos casos em que precisar acessar dados da árvore de sintaxe.

```
public class SimpleListener ... {

      private TabSimb tab; // tabela de símbolos
      private StringStack stack; // pilha para comunicação de valores

      public SimpleListener() {
            tab = new TabSimb();
            stack = new StringStack();
      }

      private void error(String s) {
            System.out.println(s);
      }

...
```

```
public void exitP(...) {
	String b = stack.popString();
	String v = stack.popString();
	System.out.println(v + "\n" + b);
}

public void exitV(...) {
	List list = lista com os nomes das variáveis;
	for (String n: list) {
		if (tab.getSymbol(n) != -1) {
			error("Identificador repetido: " + n);
		} else {
			tab.addSymbol(n);
		}
	}
	stack.pushString("data " + list.size());
}

public void enterB(...) {
	empilha uma "marca" para identificar começo de bloco;
}

public void exitB(...) {
	String s = desempilha todas as strings até encontrar a marca
		e concatena essas strings, separadas por "\n";
	stack.pushString(s);
}

public void exitA(...) {
	String n = nome da variável ID;
	int i = tab.getSymbol(n);
	if (i == -1) {
		error("Identificador não declarado: " + n);
	}
	String e = stack.popString();
	e += "\nstore " + i;
	stack.pushString(e);
}

public void exitW(...) {
	String e = stack.popString();
	e += "\nprint";
	stack.pushString(e);
}

public void exitT1(...) {
	stack.pushString("const " + texto associado a NUM);
}

public void exitT2(...) {
	String n = nome da variável ID;
	int i = tab.getSymbol(n);
	if (i == -1) {
		error("Identificador não declarado: " + n);
	}
	stack.pushString("load " + i);
}
```

```
	public void enterI(...) {
		// Ao entrar no IF, gera os 2 labels necessários,
		// que serão usados no código da condição e depois
		// no código do próprio IF
		String lab1 = tab.genLabel(); // início do comando s do IF
		String lab2 = tab.genLabel(); // depois do comando s do IF
		stack.pushString(lab1);
		stack.pushString(lab2);//
	}


	public void exitC(...) { // expressão condicional
		// desempilha as duas sub-expressões da expressão condicional
		String e2 = stack.popString();
		String e1 = stack.popString();
		String s = texto associado ao símbolo x;
		String op;
		if (s.equals("<=")) op = "jleq"; else op = "jgeq";
		// desempilha as dois labels da condição
		String labF = stack.popString();
		String labT = stack.popString();
		// constrói código
		String code = e1 + "\n" + e2 + "\n"
				+ op + labT + "\njump " + labF;
		// retorna os labels da condição para a pilha
		stack.pushString(labT);
		stack.pushString(labF);
		// empilha código construído
		stack.pushString(code);
	}

	public void exitI(...) { // comando IF
		String s = stack.popString(); // comando s, dentro do IF
		String c = stack.popString(); // condição
		String lab2 = stack.popString(); // depois do comando s
		String lab1 = stack.popString(); // início do comando s
		String i = c  + "\nlabel " + lab1 + "\n"
				+ s + "\n" + "label " + lab2;
		stack.pushString(i);
	}

	public void exitF(...) {
		String e2 = stack.popString();
		String e1 = stack.popString();
		String s = texto associado ao símbolo y;
		String op;
		if (s.equals("+")) {
			op = "add";
		} else {
			op = "sub";
		}
		stack.pushString(e1 + "\n" + e2 + "\n" + op);
	}

}
```

Primeira Questão

As figuras 1 e 2, com a gramática definida e um programa de exemplo, são repetidas abaixo.

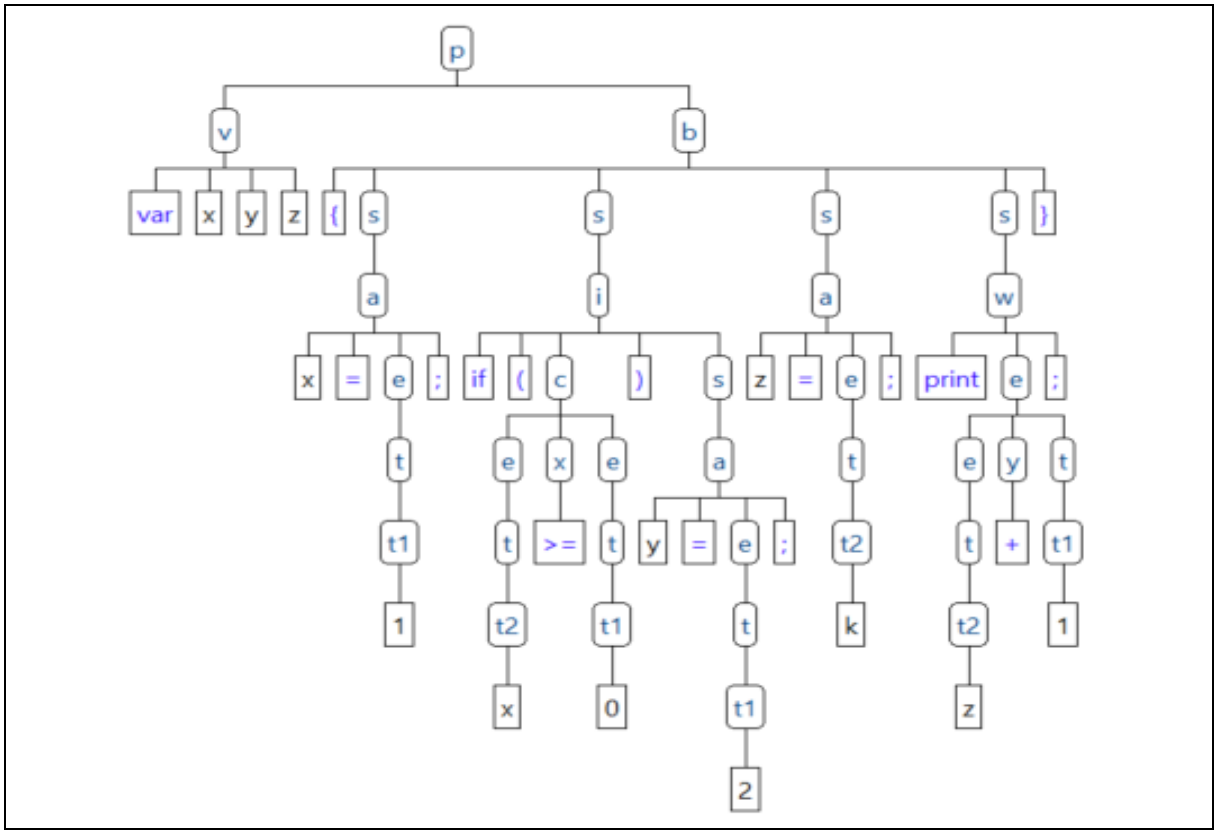
```
p -> v b
v -> 'var' ID+
b -> '{' s* '}'
s -> a ';' | w ';' | i | b
a -> ID '=' e
w -> 'print' e
i -> 'if' '(' c ')' s
c -> e x e
x ->  '>=' | '<='
e -> f | t
f -> e y t
y -> '+' | '-'
t -> t1 | t2
t1 -> NUM
t2 -> ID
```

Fig.1: Gramática da linguagem SIMPLE.

```
var x y z
{
  x = 1;
  if (x >= 0)
    y = 2;
  z = k;
  print z + 1;
}
```

Fig.2: Exemplo de programa.

Uma árvore de sintaxe para o programa da Fig. 2 é exibida abaixo. A única imprecisão é o não uso de "f" (a árvore está apresentada como se a produção fosse `e -> e y t | t`).



Execute o caminhamento na árvore simulando o comportamento do Listener apesentado. Apresente o estado da pilha após cada alteração e finalmente, a saída produzida pelo Listener.

Segunda Questão

O Listener fornecido apresenta mensagens de erro a cada vez que uma variável não declarada é encontrada. Se uma variável não declarada é usada mais de uma vez, será emitida mais de uma mensagem de erro, como no exemplo abaixo.

```
var x y
{
  x = z;
  z = y;
}
```

->

```
Identificador não declarado: z
Identificador não declarado: z
data 2
load -1
store 0
load 1
store -1
```

Altere o Listener de modo a apresentar uma mensagem de erro apenas na primeira vez que uma variável não declarada é encontrada. Faça isso **em todos os casos** em que uma variável não declarada é identificada. Como se trata de uma situação de erro, não importa o código que for gerado, pois não se deve usar o código gerado por programa com erro.

Terceira Questão

Adicione um comando "while" na linguagem SIMPLE, com sintaxe similar à de MicroJava. Defina **as produções para a gramática** e **o(s) método(s) necessário(s) no Listener** para gerar o código apropriado. Pode tomar como base a tradução do comando "if". Observe um exemplo de uso do comando proposto, e um possível código adequado a ser gerado:

```
var x
{
  x = 2;
  while (x <= 10) {
    x = x + 3;
  }
  print x;
}
```

->

```
data 1
const 2
store 0
label L1
load 0
const 10
jleq L2
jump L3
label L2
load 0
const 3
add
store 0
jump L1
label L3
load 0
print
```

Dicas:

- A condição usada no comando WHILE pode ser a mesma do IF. Neste caso, o processamento da condição espera que os 2 últimos rótulos na pilha sejam os pontos para desvio se a condição for falsa e se for verdadeira (ordem é importante).
- O comando WHILE precisa de 3 rótulos para ser especificado, enquanto que o IF simples precisava apenas de 2 rótulos. Empilhe 3 rótulos na entrada do comando WHILE, em uma ordem que satisfaça seu desempilhamento ao processar a condição.

Quarta Questão

Suponha que o assembly ofereça uma instrução "`loadA` *`base`*" com a seguinte semântica: a instrução espera que no topo da pilha esteja armazenado um valor inteiro `k`; esse valor é desempilhado e o conteúdo do endereço *`base`*`+k` é empilhado. Suponha que exista também uma instrução "`storeA` *`base`*" que espera que no topo da pilha estejam armazenados dois valores `k` e `v`; esses valores são desempilhados e o valor `v` é armazenado na posição de memória *`base`*`+k`. Essas instruções podem ajudar na tradução de comandos com arrays.

Altere **a gramática** e insira **novos métodos no Listener** para permitir declaração e uso de arrays. Se for conveniente, pode alterar também a **tabela de símbolos**. O tamanho de um array deverá ser declarado usando constante inteira. A expressão `a[i]` significa "a `i`-ésima posição contada a partir da variável `a`", mesmo se `a` não tiver sido declarada como array.

Observe o exemplo abaixo. A instrução "data" no programa assembly indica a alocação total de espaço, levando em conta o tamanho do array. As variáveis "x" e "y" são associadas aos offsets 12 e 13 respectivamente, pois "a" ocupa 10 posições. A variável "x" é usada como array no programa, mesmo sem ter sido declarada como tal.

```
var i j a[10] x y
{
  x = a[i];
  a[j+3] = y;
  x[1] = 5;
}
```

->

```
data 14
load 0
loadA 2
store 12
load 1
const 3
add
load 13
storeA 2
const 1
const 5
storeA 12
```