More Old Exam Questions

1.**[5 pts.]** The language TeenyJ is defined like TinyJ except that the syntax of <expr1> is given by:

```
<expr1> ::=   UNSIGNEDINT   |  new int '[' <expr3> ']' { '[' ']' }
```

Suppose the Parser class you completed for TinyJ Assignment 1 is to be modified so that it will parse TeenyJ programs (instead of TinyJ programs). Show how you would complete the following parsing method for <expr1>. (No code generation is expected.) **3 solutions are given on pp. 13-14.**

```
private static void expr1() throws SourceFileErrorException
{
  TJ.output.printSymbol(NTexpr1);
  TJ.output.incTreeDepth();
```

Solution to Problem 2:

```
0:      PUSHSTATADDR      0
1:      PUSHNUM           315
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM        1
5:      PUSHLOCADDR       1
6:      PUSHNUM           19
7:      PASSPARAM
8:      CALLSTATMETHOD    26
9:      SAVETOADDR
10:     PUSHSTATADDR      0
11:     LOADFROMADDR
12:     PUSHNUM           271
13:     ADDTOPTR
14:     PUSHLOCADDR       1
15:     LOADFROMADDR
16:     SAVETOADDR
17:     WRITESTRING       1     9
18:     PUSHSTATADDR      0
19:     LOADFROMADDR
20:     PUSHNUM           271
21:     ADDTOPTR
22:     LOADFROMADDR
23:     WRITEINT
24:     WRITELNOP
25:     STOP
26:     INITSTKFRM        0
27:     PUSHLOCADDR       -2
28:     LOADFROMADDR
29:     PUSHNUM           3
30:     LT
31:     JUMPONFALSE       35
32:     PUSHNUM           0
33:     RETURN            1
34:     JUMP              45
35:     PUSHLOCADDR       -2
36:     LOADFROMADDR
37:     PUSHLOCADDR       -2
38:     LOADFROMADDR
39:     PUSHNUM           1
40:     SUB
41:     PASSPARAM
42:     CALLSTATMETHOD    26
43:     SUB
44:     RETURN            1
```

```
  TJ.output.decTreeDepth();
}
```

2.**[10 pts.]** Complete the table below the following program to show the TinyJ virtual machine instructions that should be generated by TJasn.TJ (after completion of TinyJ Assignment 2) for this TinyJ program.

The solution is given on p. 1.

```
class ExamQ {
  static int b[] = new int[315];

  public static void main (String args[])
  {
    int i = g(19);
    b[271] = i;
    System.out.print("g(19) is ");
    System.out.println(b[271]);
  }

  static int g(int m)
  {
    if (m < 3) return 0;
    else return m-g(m-1);
  }
}
```

0: PUSHSTATADDR 0	16:___________________	32:___________________
1: PUSHNUM 315	17:___________________	33:___________________
2: ___________________	18:___________________	34: JUMP 45
3: SAVETOADDR	19:___________________	35:___________________
4: ___________________	20:___________________	36:___________________
5: ___________________	21:___________________	37:___________________
6: ___________________	22:___________________	38:___________________
7: ___________________	23:___________________	39:___________________
8: ___________________	24:___________________	40:___________________
9: ___________________	25:___________________	41:___________________
10:___________________	26:___________________	42:___________________
11:___________________	27:___________________	43:___________________
12:___________________	28:___________________	44: RETURN 1
13:___________________	29:___________________	
14:___________________	30:___________________	
15:___________________	31:___________________	

Hint: Among the 40 instructions you are asked to write, there are 7 LOADFROMADDR instructions, 6 PUSHNUM instructions, 5 PUSHLOCADDR instructions, 2 each of the ADDTOPTR, CALLSTATMETHOD, INITSTKFRM, PASSPARAM, PUSHSTATADDR**,** SAVETOADDR, and SUB instructions, and 1 each of the HEAPALLOC, JUMPONFALSE, LT, RETURN, STOP, WRITEINT, WRITELNOP, and WRITESTRING instructions.

While reading this page and the next, you should refer back when necessary to the pages of https://euclid.cs.qc.cuny.edu/316/Memory-allocation-VM-instruction-set-and-hints-for-asn-2.pdf that specify the effects of executing each VM instruction.

Comments on Problem 2 Regarding the Translation of the Statements `b[271] = i;` and `System.out.println(b[271]);`

Note: The **EXPRSTACK** column on the right shows the items on the expression evaluation stack immediately ***after*** each VM instruction has been executed. The stack grows downwards—when more than one item is on the stack the first line below the word **EXPRSTACK** refers to the ***bottom*** item on the stack.

`b[271] = i;` is translated into the seven VM instructions that are shown on the left below. These instructions are put into code memory at addresses 10 – 16, as indicated on p. 1.

`PUSHSTATADDR 0`	Pushes **pointer to `b`**.	**EXPRSTACK** **ptr to `b`**
`LOADFROMADDR`	Pops **pointer to `b`**. Pushes the **pointer to `b[0]` that is stored in `b`'s location**.	**EXPRSTACK** **ptr to `b[0]`**
`PUSHNUM 271`	Pushes the **integer `271`**.	**EXPRSTACK** **ptr to `b[0]`** **`271`**
`ADDTOPTR`	Pops **`271`** and **pointer to `b[0]`**. Pushes **(pointer to `b[0]`) + `271`** (i.e., **pointer to `b[271]`**).	**EXPRSTACK** **ptr to `b[271]`**
`PUSHLOCADDR 1`	Pushes **pointer to `i`**.	**EXPRSTACK** **ptr to `b[271]`** **ptr to `i`**
`LOADFROMADDR`	Pops **pointer to `i`**. Pushes the **value stored in `i`'s location** (i.e., the **value of `i`**).	**EXPRSTACK** **ptr to `b[271]`** **value of `i`**
`SAVETOADDR`	Pops **value of `i`** and **pointer to `b[271]`**. Saves **value of `i`** into the location of **`b[271]`**.	**EXPRSTACK** *is empty*

`System.out.println(b[271]);` is translated into the seven VM instructions that are shown on the left below. These instructions are put into code memory at addresses 18 – 24, as indicated on p. 1.

`PUSHSTATADDR 0`	Pushes **pointer to `b`**.	**EXPRSTACK** **ptr to `b`**
`LOADFROMADDR`	Pops **pointer to `b`**. Pushes the **pointer to `b[0]` that is stored in `b`'s location**.	**EXPRSTACK** **ptr to `b[0]`**
`PUSHNUM 271`	Pushes the **integer `271`**	**EXPRSTACK** **ptr to `b[0]`** **`271`**
`ADDTOPTR`	Pops **`271`** and **pointer to `b[0]`**. Pushes **(pointer to `b[0]`) + `271`** (i.e., **pointer to `b[271]`**).	**EXPRSTACK** **ptr to `b[271]`**
`LOADFROMADDR`	Pops **pointer to `b[271]`**. Pushes the **value stored in `b[271]`'s location** (i.e., the **value of `b[271]`**).	**EXPRSTACK** **value of `b[271]`**
`WRITEINT`	Pops **value of `b[271]`**. Writes the value on the screen.	**EXPRSTACK** *is empty*
`WRITELNOP`	Writes a newline to the screen.	**EXPRSTACK** *is empty*

Further problems to test your understanding:

3. Suppose we delete the line `static int b[] = new int[315];` from the TinyJ program of problem 2 but insert a line `int b[] = new int[536];` at the beginning of the body of `main`. (Thus `b` would become the first local variable of `main`, and `i` would become the second local variable of `main` rather than the first local variable.) How would the 14 instructions shown on the previous page change?
   `Answer:` `PUSHLOCADDR 1` would be changed to `PUSHLOCADDR 2`.
   Each occurrence of `PUSHSTATADDR 0` would be changed to `PUSHLOCADDR 1`.

4. Suppose that the first variable declaration in a certain TinyJ program is `static int b[][][];` Suppose also that this variable `b` is used in the following statement later in the program: `System.out.print(b[7][29][5]);` What TinyJ VM instructions would the TinyJ compiler translate the latter statement into? [**Note**: Although Exam 2 may have questions relating to arrays, Exam 2 will ***not*** have any question such as this one that involves an indexed variable with more than one actual index. However, there may be a question on the ***Final Exam*** that involves an indexed variable with more than one index.]

Answer to problem 4, and explanation of the generated intructions:

`PUSHSTATADDR 0`	Pushes **pointer to `b`**.	**EXPRSTACK** **ptr to `b`**
`LOADFROMADDR`	Pops **pointer to `b`**. Pushes the **pointer to `b[0]` that is stored in `b`'s location**.	**EXPRSTACK** **ptr to `b[0]`**
`PUSHNUM 7`	Pushes the **integer `7`**.	**EXPRSTACK** **ptr to `b[0]`** **`7`**
`ADDTOPTR`	Pops **`7`** and **pointer to `b[0]`**. Pushes **(pointer to `b[0]`) + `7`** (i.e., **pointer to `b[7]`**).	**EXPRSTACK** **ptr to `b[7]`**
`LOADFROMADDR`	Pops **pointer to `b[7]`**. Pushes the **pointer to `b[7][0]` that is stored in `b[7]`'s location**.	**EXPRSTACK** **ptr to `b[7][0]`**
`PUSHNUM 29`	Pushes the **integer `29`**.	**EXPRSTACK** **ptr to `b[7][0]`** **`29`**
`ADDTOPTR`	Pops **`29`** and **pointer to `b[7][0]`**. Pushes **(pointer to `b[7][0]`) + `29`** (i.e., **pointer to `b[7][29]`**).	**EXPRSTACK** **ptr to `b[7][29]`**
`LOADFROMADDR`	Pops **pointer to `b[7][29]`**. Pushes the **pointer to `b[7][29][0]` that is stored in `b[7][29]`'s location**.	**EXPRSTACK** **ptr to `b[7][29][0]`**
`PUSHNUM 5`	Pushes the **integer `5`**.	**EXPRSTACK** **ptr to `b[7][29][0]`** **`5`**
`ADDTOPTR`	Pops **`5`** and **pointer to `b[7][29][0]`**. Pushes **(pointer to `b[7][29][0]`) + 5** (i.e., **pointer to `b[7][29][5]`**).	**EXPRSTACK** **ptr to `b[7][29][5]`**
`LOADFROMADDR`	Pops **pointer to `b[7][29][5]`**. Pushes **value stored in `b[7][29][5]`'s location** (i.e., **value of `b[7][29][5]`**).	**EXPRSTACK** **value of `b[7][29][5]`**
`WRITEINT`	Pops **value of `b[7][29][5]`**. Writes this value to the screen.	**EXPRSTACK** *is empty*

Some Properties of the Code That is Generated by the TinyJ Compiler

When answering certain exam questions, it may be useful to remember that the code generated by the TinyJ compiler when it translates a TinyJ program has the following properties:

1. The code generated for any assignment statement consists of code whose execution will push a pointer to the target variable's location, followed by code whose execution will push what we want to store, followed by a SAVETOADDR instruction.
2. If the target variable of an assignment statement is ***not*** an indexed variable, then the assignment statement is translated into code that begins with a PUSHSTATADDR or PUSHLOCADDR instruction that is ***not*** immediately followed by a LOADFROMADDR instruction.
3. With the exceptions of the PUSHSTATADDR and PUSHLOCADDR instructions referred to in item 2, every PUSHSTATADDR or PUSHLOCADDR instruction is immediately followed by a LOADFROMADDR instruction.
4. The code generated to push the value of an indexed variable onto EXPRSTACK consists of code whose execution will push a pointer to the indexed variable's location and an additional LOADFROMADDR instruction at the end.
5. The code generated for any method call includes a PASSPARAM instruction for each argument of the call, and includes a CALLSTATMETHOD instruction; if the call has arguments, then the CALLSTATMETHOD instruction is the next instruction after the last PASSPARAM.

Much further information about the generated code is provided in the document https://euclid.cs.qc.cuny.edu/316/Slides/Memory-allocation-VM-instruction-set-and-hints-for-asn-2.pdf that has been discussed in class—especially in the section **Code Generation Rules Used by the TinyJ Compiler** and the sections relating to `whileStmt()` and `ifStmt()`.

More Hand-Translation Examples

[Note: You can also make up your own hand-translation examples: If X.java is any valid TinyJ program, then the correct solution to the problem of translating X.java can be obtained by running my solution to TinyJ Assignment 2 with X.java as the input file.]

(a) Suppose Instruction.getNextCodeAddress() == 35 when a correct solution to TinyJ Assignment 2 begins to translate the following two methods. What code is generated?

```
static void m()
{
  int x = 12, y = 9;
  System.out.print(p(17, y, x+5));
}

static int p (int a, int b, int c)
{
  int u = a - b;
  return c - u;
}
```

SOLUTION:

```
35:     INITSTKFRM        2
36:     PUSHLOCADDR       1
37:     PUSHNUM           12
38:     SAVETOADDR
39:     PUSHLOCADDR       2
40:     PUSHNUM           9
41:     SAVETOADDR
42:     PUSHNUM           17
43:     PASSPARAM
44:     PUSHLOCADDR       2
45:     LOADFROMADDR
46:     PASSPARAM
47:     PUSHLOCADDR       1
48:     LOADFROMADDR
49:     PUSHNUM           5
50:     ADD
51:     PASSPARAM
52:     CALLSTATMETHOD    55
53:     WRITEINT
54:     RETURN            0
55:     INITSTKFRM        1
56:     PUSHLOCADDR       1
57:     PUSHLOCADDR       -4
58:     LOADFROMADDR
59:     PUSHLOCADDR       -3
60:     LOADFROMADDR
61:     SUB
62:     SAVETOADDR
63:     PUSHLOCADDR       -2
64:     LOADFROMADDR
65:     PUSHLOCADDR       1
66:     LOADFROMADDR
67:     SUB
68:     RETURN            3
```

(b) An example involving arrays:

```
class ArrayTest {
  static int b[] = new int[10];

  public static void main (String args[])
  {
    int a = 1;
    b[3] = a;
    System.out.println(b[3]+a);

    b = new int[5];

    int c[][] = new int [7][];
    c[4] = b;
  }
}
```

What would a correct solution to TinyJ Assignment 2 translate this into?

SOLUTION:

```
0:      PUSHSTATADDR      0
1:      PUSHNUM           10
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM        2
5:      PUSHLOCADDR       1
6:      PUSHNUM           1
7:      SAVETOADDR
8:      PUSHSTATADDR      0
9:      LOADFROMADDR
10:     PUSHNUM           3
11:     ADDTOPTR
12:     PUSHLOCADDR       1
13:     LOADFROMADDR
14:     SAVETOADDR
15:     PUSHSTATADDR      0
16:     LOADFROMADDR
17:     PUSHNUM           3
18:     ADDTOPTR
19:     LOADFROMADDR
20:     PUSHLOCADDR       1
21:     LOADFROMADDR
22:     ADD
23:     WRITEINT
24:     WRITELNOP
25:     PUSHSTATADDR      0
26:     PUSHNUM           5
27:     HEAPALLOC
28:     SAVETOADDR
29:     PUSHLOCADDR       2
30:     PUSHNUM           7
31:     HEAPALLOC
32:     SAVETOADDR
33:     PUSHLOCADDR       2
34:     LOADFROMADDR
35:     PUSHNUM           4
36:     ADDTOPTR
37:     PUSHSTATADDR      0
38:     LOADFROMADDR
39:     SAVETOADDR
40:     STOP
```

(c) Example involving a while loop:

```
class Fall02a {
  static int a[] = new int[10];

  public static void main (String args[])
  {
       int x = 100;

       while (x > 10)
         x = f(x, 2);
  }

  static int f(int m, int n)
  {
       a[3] = m / n;
       return a[3];
  }
}
```

What would a correct solution to TinyJ Assignment 2 translate this into?

SOLUTION:

```
0:      PUSHSTATADDR      0
1:      PUSHNUM           10
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM        1
5:      PUSHLOCADDR       1
6:      PUSHNUM           100
7:      SAVETOADDR
8:      PUSHLOCADDR       1
9:      LOADFROMADDR
10:     PUSHNUM           10
11:     GT
12:     JUMPONFALSE       22
13:     PUSHLOCADDR       1
14:     PUSHLOCADDR       1
15:     LOADFROMADDR
16:     PASSPARAM
17:     PUSHNUM           2
18:     PASSPARAM
19:     CALLSTATMETHOD    23
20:     SAVETOADDR
21:     JUMP              8
22:     STOP
23:     INITSTKFRM        0
24:     PUSHSTATADDR      0
25:     LOADFROMADDR
26:     PUSHNUM           3
27:     ADDTOPTR
28:     PUSHLOCADDR       -3
29:     LOADFROMADDR
30:     PUSHLOCADDR       -2
31:     LOADFROMADDR
32:     DIV
33:     SAVETOADDR
34:     PUSHSTATADDR      0
35:     LOADFROMADDR
36:     PUSHNUM           3
37:     ADDTOPTR
38:     LOADFROMADDR
39:     RETURN            2
```

(d) Another example involving a while loop:

```
class Fall02b {

  static int a[] = new int [25];

  public static void main (String args[])
  {
       a[5] = 900;
       System.out.print(g(7));
  }

  static int g(int m)
  {
       int i = a[5];

       while (i > 30)
         i = i / m;

       return i;
  }
}
```

What would a correct solution to TinyJ Assignment 2 translate this into?

SOLUTION:

```
0:      PUSHSTATADDR      0
1:      PUSHNUM           25
2:      HEAPALLOC
3:      SAVETOADDR
4:      INITSTKFRM        0
5:      PUSHSTATADDR      0
6:      LOADFROMADDR
7:      PUSHNUM           5
8:      ADDTOPTR
9:      PUSHNUM           900
10:     SAVETOADDR
11:     PUSHNUM           7
12:     PASSPARAM
13:     CALLSTATMETHOD    16
14:     WRITEINT
15:     STOP
16:     INITSTKFRM        1
17:     PUSHLOCADDR       1
18:     PUSHSTATADDR      0
19:     LOADFROMADDR
20:     PUSHNUM           5
21:     ADDTOPTR
22:     LOADFROMADDR
23:     SAVETOADDR
24:     PUSHLOCADDR       1
25:     LOADFROMADDR
26:     PUSHNUM           30
27:     GT
28:     JUMPONFALSE       37
29:     PUSHLOCADDR       1
30:     PUSHLOCADDR       1
31:     LOADFROMADDR
32:     PUSHLOCADDR       -2
33:     LOADFROMADDR
34:     DIV
35:     SAVETOADDR
36:     JUMP              24
```

```
37:     PUSHLOCADDR       1
38:     LOADFROMADDR
39:     RETURN            1
```

(e) The next example involves if as well as while:

```
import java.util.Scanner;

class Spring99 {

  static Scanner input = new Scanner(System.in);
  static int x;

  public static void main (String args[])
  {
    int a;

    x = input.nextInt();
    if (x > 1  &  x < 20000) {
      while (x <= 20000) {
        int b = 2;
        x = x * 3;
      }

      int c = x;
      System.out.print(c);
    }
  }
}
```

What would a correct solution to TinyJ Assignment 2 translate this into?

SOLUTION:

Note that the local variables b and c both have a stackframe offset of 2. At the point where c is declared, b no longer exists--b's scope is confined to the body of the while loop. Thus stackframe offset 2 can be reallocated to c.

```
0:      INITSTKFRM        2
1:      PUSHSTATADDR      0
2:      READINT
3:      SAVETOADDR
4:      PUSHSTATADDR      0
5:      LOADFROMADDR
6:      PUSHNUM           1
7:      GT
8:      PUSHSTATADDR      0
9:      LOADFROMADDR
10:     PUSHNUM           20000
11:     LT
12:     AND
13:     JUMPONFALSE       36
14:     PUSHSTATADDR      0
15:     LOADFROMADDR
16:     PUSHNUM           20000
17:     LE
18:     JUMPONFALSE       29
19:     PUSHLOCADDR       2
20:     PUSHNUM           2
21:     SAVETOADDR
22:     PUSHSTATADDR      0
23:     PUSHSTATADDR      0
```

```
24:     LOADFROMADDR
25:     PUSHNUM           3
26:     MUL
27:     SAVETOADDR
28:     JUMP              14
29:     PUSHLOCADDR       2
30:     PUSHSTATADDR      0
31:     LOADFROMADDR
32:     SAVETOADDR
33:     PUSHLOCADDR       2
34:     LOADFROMADDR
35:     WRITEINT
36:     STOP
```

(f) Suppose Instruction.getNextCodeAddress() == 45 when a correct solution to TinyJ Assignment 2 begins to translate the following method, and suppose z is a static variable with address 2. What TinyJ VM instructions would this method be translated into?

```
static int p(int x)
{
  int y = 3, w;

  x = z + y;

  if (x < 10) z = x;
  else z = y;

  while (z <= 100) {
    System.out.println(z);
    z = z + y;
  }

  return z - x;
}
```

SOLUTION:

```
45:     INITSTKFRM        2
46:     PUSHLOCADDR       1
47:     PUSHNUM           3
48:     SAVETOADDR
49:     PUSHLOCADDR       -2
50:     PUSHSTATADDR      2
51:     LOADFROMADDR
52:     PUSHLOCADDR       1
53:     LOADFROMADDR
54:     ADD
55:     SAVETOADDR
56:     PUSHLOCADDR       -2
57:     LOADFROMADDR
58:     PUSHNUM           10
59:     LT
60:     JUMPONFALSE       66
61:     PUSHSTATADDR      2
62:     PUSHLOCADDR       -2
63:     LOADFROMADDR
64:     SAVETOADDR
65:     JUMP              70
66:     PUSHSTATADDR      2
67:     PUSHLOCADDR       1
68:     LOADFROMADDR
69:     SAVETOADDR
70:     PUSHSTATADDR      2
```

```
71:     LOADFROMADDR
72:     PUSHNUM          100
73:     LE
74:     JUMPONFALSE      87
75:     PUSHSTATADDR     2
76:     LOADFROMADDR
77:     WRITEINT
78:     WRITELNOP
79:     PUSHSTATADDR     2
80:     PUSHSTATADDR     2
81:     LOADFROMADDR
82:     PUSHLOCADDR      1
83:     LOADFROMADDR
84:     ADD
85:     SAVETOADDR
86:     JUMP             70
87:     PUSHSTATADDR     2
88:     LOADFROMADDR
89:     PUSHLOCADDR      -2
90:     LOADFROMADDR
91:     SUB
92:     RETURN           1
```

Three Solutions to the Recursive Descent Parsing Problem on Page 1

First Solution:

```
private void expr1() throws SourceFileErrorException
{
    TJ.output.printSymbol(NTexpr1);
    TJ.output.incTreeDepth();

    switch (getCurrentToken()) {
        case UNSIGNEDINT:
            nextToken();
            break;

        case NEW:
            nextToken();
            accept(INT);
            accept(LBRACKET);
            expr3();
            accept(RBRACKET);
            while (getCurrentToken() == LBRACKET) {
                nextToken();
                accept(RBRACKET);
            }
            break;

        default:
            throw new SourceFileErrorException("Expected UNSIGNEDINT or new");
    }

    TJ.output.decTreeDepth();
}
```

Second Solution:

```
private void expr1() throws SourceFileErrorException
{
    TJ.output.printSymbol(NTexpr1);
    TJ.output.incTreeDepth();

    if (getCurrentToken() == UNSIGNEDINT) {
        nextToken();
    }
    else if (getCurrentToken() == NEW) {
        nextToken();
        accept(INT);
        accept(LBRACKET);
        expr3();
        accept(RBRACKET);
        while (getCurrentToken() == LBRACKET) {
            nextToken();
            accept(RBRACKET);
        }
    }
    else throw new SourceFileErrorException("Expected UNSIGNEDINT or new");

    TJ.output.decTreeDepth();
}
```

Third Solution:

```
private void expr1() throws SourceFileErrorException
{
    TJ.output.printSymbol(NTexpr1);
    TJ.output.incTreeDepth();

    if (getCurrentToken() == UNSIGNEDINT) {
        nextToken();
    }
    else {
        accept(NEW);
        accept(INT);
        accept(LBRACKET);
        expr3();
        accept(RBRACKET);
        while (getCurrentToken() == LBRACKET) {
            nextToken();
            accept(RBRACKET);
        }
    }

    TJ.output.decTreeDepth();
}
```

COMMENT: The third solution is slightly more concise than the first two solutions, but it gives a slightly less informative error message if expr1 is called when currentToken is neither UNSIGNEDINT nor NEW.

A Mistake to Avoid When Writing Recursive Descent Parsing Code

A common mistake in writing recursive descent parsing code is to write

`getCurrentToken() ==` *X*

or `accept(`*X*`)` [which performs a `getCurrentToken() ==` *X* test]

using a `Symbols` constant *X* that represents a ***non***terminal. This is wrong, as `getCurrentToken()` returns a `Symbols` constant that represents a ***token***. Here are two examples of this kind of mistake.

1. In TinyJ Assignment 1, the method `argumentList()` should be based on the following EBNF rule:

   ```
   <argumentList>    ::=   '('[<expr3>{,<expr3>}]')'
   ```

   When writing this method it would be ***wrong*** to write:

   ```
   accept(LPAREN);
   if (getCurrentToken() == NTexpr3) /* INCORRECT! */ {
      expr3();
       ...      // a while loop that deals with {,<expr3>}
   }
   accept(RPAREN);
   ```

   Here it would be correct to write code of the following form:

   ```
   accept(LPAREN);
   if (getCurrentToken() != RPAREN) /* CORRECT */ {
      expr3();
       ...      // a while loop that deals with {,<expr3>}
   }
   accept(RPAREN);
   ```

2. When writing the method `expr1()` for TinyJ Assignment 1, one case that needs to be dealt with relates to the following part of the TinyJ EBNF rule that defines `<expr1>`:

   ```
   IDENTIFIER ( . nextInt '(' ')' | [<argumentList>]{'[' <expr3> ']'} )
   ```

   Here it would be wrong to write something like:

   ```
   case IDENT:
     nextToken();
     if (getCurrentToken() != DOT) {
       if (getCurrentToken() == NTargumentList /* INCORRECT! */ ) argumentList();
       ... // a while loop that deals with {'[' <expr3> ']'}
     }
     else {
       ... // code to deal with  . nextInt '(' ')'
     }
     break;
   ```

   Instead, you can write something like:

   ```
   case IDENT:
     nextToken();
     if (getCurrentToken() != DOT) {
       if (getCurrentToken() == LPAREN /* CORRECT */ ) argumentList();
       ... // a while loop that deals with {'[' <expr3> ']'}
     }
     else {
       ... // code to deal with  . nextInt '(' ')'
     }
     break;
   ```

   The use of `LPAREN` in the above code is correct because the first token of any instance of `<argumentList>` must be a left parenthesis, as we see from the EBNF rule

   ```
   <argumentList>    ::=   '('[<expr3>{,<expr3>}]')'
   ```