

CSCI 316 (Kong): TinyJ Assignment 1

To be submitted **no later than**: Thursday, December 4. [Note: If mars fails to operate normally or becomes inaccessible at any time after **6 p.m.**, on this due date, the submission deadline will **not** be extended. Try to submit no later than noon that day, and **on an earlier day if possible**. TinyJ Assignment 2 will be provided to you on or before **Wednesday, November 26**.] This assignment counts **1.5%** towards your grade if the grade is computed using rule A.

The TinyJ language is an extremely small subset of Java. Every valid TinyJ program is a valid Java program, and has the same semantics whether it is regarded as a TinyJ or a Java program. The syntax of TinyJ is given by the EBNF specification that is shown below. *In this EBNF specification each terminal is a token of TinyJ, and each nonterminal <X> denotes the set of all sequences of tokens that are **syntactically valid** for the TinyJ construct X.* In particular, a piece of source code is a *syntactically valid* TinyJ program if and only if its sequence of tokens belongs to the language generated by this EBNF specification. A piece of source code is a valid TinyJ program if and only if it is *both* a syntactically valid TinyJ program *and* a valid Java 8 program, with a few exceptions: TinyJ does **not** allow non-decimal (i.e., hexadecimal, octal, or binary) or long integer literals, underscores in integer literals, method name overloading, program arguments, printing of Boolean values, “return;” statements within the **main()** method, escape sequences other than \n, \\, and \", and ints that are $\geq 2^{31} - 2^{16} = 2,147,418,112$.

Reserved words of TinyJ are shown in boldface in this EBNF specification. Some names used by Java library packages, classes, and their methods (e.g., **java**, **Scanner**, and **nextInt**) are reserved words of TinyJ, as is **main**. Otherwise, IDENTIFIER here means any Java identifier consisting of ASCII characters.

→ indicates a nonterminal whose method you must complete in Parser.java--other methods have already been written for you!

```
<program> ::= [<importStmt>] class IDENTIFIER '{' {<dataFieldDecl>}
                                     <mainDecl> {<methodDecl>} '}'

<importStmt> ::= import java . util . Scanner ;
<dataFieldDecl> ::= static <varDecl>
<varDecl> ::= int <singleVarDecl> { , <singleVarDecl> } ;
               | Scanner IDENTIFIER = new Scanner '(' System . in ')' ;
→ <singleVarDecl> ::= IDENTIFIER { '[' ']' } [ = <expr3> ]
→ <mainDecl> ::= public static void main '(' String IDENTIFIER '[' ']' ')'
               <compoundStmt>
→ <methodDecl> ::= static ( void | int '[' ']' ) IDENTIFIER
               '(' <parameterDeclList> ')' <compoundStmt>

<parameterDeclList> ::= [<parameterDecl> { , <parameterDecl> } ]
<parameterDecl> ::= int IDENTIFIER '[' ']'
→ <compoundStmt> ::= '{' { <statement> } '}'
→ <statement> ::= ; | return [<expr3>] ; | <varDecl> | <assignmentOrInvoc>
               | <compoundStmt> | <ifStmt> | <whileStmt> | <outputStmt>
→ <assignmentOrInvoc> ::= IDENTIFIER ( { '[' <expr3> ']' } = <expr3> ; | <argumentList> ; )
→ <argumentList> ::= '(' [<expr3> { , <expr3> } ] ')'
→ <ifStmt> ::= if '(' <expr7> ')' <statement> [else <statement>]
→ <whileStmt> ::= while '(' <expr7> ')' <statement>
→ <outputStmt> ::= System . out . ( print '(' <printArgument> ')' ;
               | println '(' [ <printArgument> ] ')' ;
               )

→ <printArgument> ::= CHARSTRING | <expr3>
→ <expr7> ::= <expr6> { '|' <expr6> }
→ <expr6> ::= <expr5> { & <expr5> }
→ <expr5> ::= <expr4> { (== | !=) <expr4> }
→ <expr4> ::= <expr3> [ (> | < | >= | <=) <expr3> ]
→ <expr3> ::= <expr2> { (+ | -) <expr2> }
→ <expr2> ::= <expr1> { (* | / | %) <expr1> }
→ <expr1> ::= '(' <expr7> ')' | (+|-|!) <expr1> | UNSIGNEDINT | null
               | new int '[' <expr3> ']' { '[' ']' }
               | IDENTIFIER ( . nextInt '(' ') | [<argumentList>] { '[' <expr3> ']' }
```

This is the first of three TinyJ assignments. After completing all three assignments you will have a program that can compile any TinyJ program into a simple virtual machine code, and then execute the virtual machine code it has generated. (Execution should produce the same run-time behavior as you would get if you compiled the same TinyJ program using `javac` into a `.class` file and then executed that `.class` file using a Java VM.) **There will be exam questions relating to the TinyJ assignments.**

TinyJ Assignment 1 will not deal with compilation of TinyJ programs, nor with execution of virtual machine code, but only with *syntax analysis* of TinyJ programs. The goal of TinyJ Assignment 1 is to complete a program that will:

- (a) determine if the sequence of tokens of its input file belongs to `<program>` (as defined by the above EBNF rules), and
- (b) output a parse tree of the sequence of tokens of its input file, if that sequence belongs to `<program>`.

Regarding (a), note that the sequence of tokens of the input file belongs to `<program>` if, and only if, the input file is a *syntactically* valid TinyJ program. However, a syntactically valid TinyJ program may still contain errors like “undeclared variable” or “array index out of range”. A “sideways” representation of ordered trees, described below, will be used for (b).

A Sideways Representation of an Ordered Rooted Tree T

If T has just one node, then
Otherwise,

representation of T = the unique node of T
representation of T = the root of T

representation of the 1st subtree of the root of T
representation of the 2nd subtree of the root of T
...
representation of the last subtree of the root of T
... node has no more children

Note: The vertical red lines on the left and below are NOT part of the sideways representation: Those red lines were added to help you understand the tree.

In this sideways representation, sibling nodes always have the *same* indentation, but each non-root node is further indented than its parent; *the indentation of a node is proportional to the depth of that node in the tree*. Here are the “ordinary” and the “sideways” representations of a tree:

```

      <expr4>
      |
      <expr3>
    /  |  \
<expr2> + <expr2>
 /  \  /  \
<expr1> <expr1> * <expr1>
 |       |       |
UNSIGNEDINT IDENTIFIER UNSIGNEDINT

```

```

<expr4>
<expr3>
<expr2>
<expr1>
|
UNSIGNEDINT
|
... node has no more children
+
<expr2>
<expr1>
|
IDENTIFIER
|
... node has no more children
*
<expr1>
|
UNSIGNEDINT
|
... node has no more children
... node has no more children
... node has no more children
... node has no more children

```

Indentation level
= **depth of node** in the tree

Indentation levels of consecutive lines are equal or differ by just 1:

If previous line is a *nonterminal*, then the current line's indentation level is **higher by 1**.

If previous line is a *token/terminal*, then the current line's indentation level is **the same**.

If previous line is ... node has no more children, the current line's indentation level is **lower by 1**.

NOTE: If a sideways representation of a parse tree appears as part of an exam question, it probably will not be accompanied by vertical red lines such as those you see above!

How to Install the TinyJ Assignment 1 Files on mars, and (optionally) Your PC / Mac

Do 1 and 2, and optionally 3 – 8, **before our class on Wednesday, November 26 (and preferably before our class on Monday, November 24)**. Remember that Unix/Linux file and command names are case-sensitive when following the instructions below!

1. Login to your **xxxxx_yyyy316 mars** account and enter the following command at the **xxxxx_yyyy316@mars:~\$** prompt:
/home/faculty/ykong/TJ1setup

IMPORTANT: The 1 in TJ1setup is the digit 1, not the letter l.

2. Wait for “TJ1setup done” to appear on the screen. Then enter the following at the **xxxxx_yyyy316@mars:~\$** prompt:
java -cp TJ1solclasses:. TJ1asn.TJ CS316ex12.java 12.sol

Note the period after the *colon* in this command. *This command executes my solution to this assignment with CS316ex12.java as the input file and 12.sol as the output file.* A listing of CS316ex12.java should be displayed on the screen, and 12.sol should contain a sideways representation of the program’s parse tree afterwards. **There should not be any error message**. To view the parse tree, you can use **less 12.sol** or just open 12.sol in an editor.

A Correct TinyJ Input File and the Corresponding Parse Tree That is Written to the Output File by a Solution to TinyJ Assignment 1

Input File (a TinyJ program):

```
import java.util.Scanner;

class Simple2 {

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

    public static void main(String args[])
    {
        int x = input.nextInt();
        x = x % 3;
        System.out.println(x + 2);
    }
}
```

Output File (the above program's parse tree, in sideways representation):

```
<program>
<importStmt>
  Reserved Word: import
  Reserved Word: java
  .
  Reserved Word: util
  .
  Reserved Word: Scanner
  ;
  ... node has no more children
Reserved Word: class
IDENTIFIER: Simple2
{
<dataFieldDecl>
  Reserved Word: static
<varDecl>
  Reserved Word: Scanner
  IDENTIFIER: input
  =
  Reserved Word: new
  Reserved Word: Scanner
  (
  Reserved Word: System
  .
  Reserved Word: in
  )
  ;
  ... node has no more children
  ... node has no more children
<mainDecl>
  Reserved Word: public
  Reserved Word: static
  Reserved Word: void
  Reserved Word: main
  (
  Reserved Word: String
  IDENTIFIER: args
  [
  ]
  )
<compoundStmt>
{
  <statement>
  <varDecl>
    Reserved Word: int
```

IMPORTANT

This output is **NOT** automatically displayed on the screen! To see it, you must view the output file (e.g., using the **less** command on mars or using an editor).

```
<singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
  <expr2>
  <expr1>
    IDENTIFIER: input
    .
    Reserved Word: nextInt
    (
    )
    ... node has no more children
    ... node has no more children
    ... node has no more children
    ... node has no more children
  ;
  ... node has no more children
  ... node has no more children
<statement>
<assignmentOrInvoc>
  IDENTIFIER: x
  =
  <expr3>
  <expr2>
  <expr1>
    IDENTIFIER: x
    ... node has no more children
  %
  <expr1>
    UNSIGNED INTEGER LITERAL: 3
    ... node has no more children
    ... node has no more children
    ... node has no more children
  ;
  ... node has no more children
  ... node has no more children
<statement>
<outputStmt>
  Reserved Word: System
  .
  Reserved Word: out
  .
  Reserved Word: println
  (
  <printArgument>
  <expr3>
  <expr2>
  <expr1>
    IDENTIFIER: x
    ... node has no more children
    ... node has no more children
  +
  <expr2>
  <expr1>
    UNSIGNED INTEGER LITERAL: 2
    ... node has no more children
    ... node has no more children
    ... node has no more children
    ... node has no more children
  )
  ;
  ... node has no more children
  ... node has no more children
}
  ... node has no more children
  ... node has no more children
}
```

A TinyJ Input File with a Syntax Error, and the Output That is Generated by a Solution to TinyJ Assignment 1

Input File (the error is that `public` should be preceded by a semicolon):

```
import java.util.Scanner;

class Simple2 {

    static Scanner input = new Scanner(System.in)

    public static void main(String args[])
    {
        int x = input.nextInt();
        x = x % 3;
        System.out.println(x + 2);
    }
}
```

Wrong token: Symbols.SEMICOLON expected, not Symbols.PUBLIC

Output on the Screen (shows the tokens that are read before the error is detected):

```
1:  import java.util.Scanner;
2:
3:  class Simple2 {
4:
5:      static Scanner input = new Scanner(System.in)
6:
7:      public
```

```
ERROR!  Something's wrong--maybe the following token is missing: ;
input.currentChar = ' '
LexicalAnalyzer.currentToken = Reserved Word: public
```

Output File (an incomplete parse tree whose leaves are the tokens that appear in the input file before the syntax error):

```
<program>
<importStmt>
  Reserved Word: import
  Reserved Word: java
  .
  Reserved Word: util
  .
  Reserved Word: Scanner
  ;
  ... node has no more children
Reserved Word: class
IDENTIFIER: Simple2
{
<dataFieldDecl>
  Reserved Word: static
<varDecl>
  Reserved Word: Scanner
  IDENTIFIER: input
  =
  Reserved Word: new
  Reserved Word: Scanner
  (
  Reserved Word: System
  .
  Reserved Word: in
  )
```

Last correct token: Symbols.RPAREN

```

1  package Tjlexer;
2
3  import java.util.EnumSet;
4
5  public enum Symbols {
6
7      // TinyJ reserved words
8      INT("Reserved Word: int", "int"),
9      VOID("Reserved Word: void", "void"),
10     STATIC("Reserved Word: static", "static"),
11     IF("Reserved Word: if", "if"),
12     WHILE("Reserved Word: while", "while"),
13     ELSE("Reserved Word: else", "else"),
14     NEW("Reserved Word: new", "new"),
15     OUT("Reserved Word: out", "out"),
16     PRINT("Reserved Word: print", "print"),
17     SYSTEM("Reserved Word: System", "System"),
18     PRINTLN("Reserved Word: println", "println"),
19     RETURN("Reserved Word: return", "return"),
20     IN("Reserved Word: in", "in"),
21     NULL("Reserved Word: null", "null"),
22     NEXTINT("Reserved Word: nextInt", "nextInt"),
23     MAIN("Reserved Word: main", "main"),
24     JAVA("Reserved Word: java", "java"),
25     UTIL("Reserved Word: util", "util"),
26     CLASS("Reserved Word: class", "class"),
27     STRING("Reserved Word: String", "String"),
28     PUBLIC("Reserved Word: public", "public"),
29     IMPORT("Reserved Word: import", "import"),
30     SCANNER("Reserved Word: Scanner", "Scanner"),
31     // End of TinyJ reserved words. (See the definition of the reservedWords EnumSet below.)
32
33     // Other TinyJ tokens that have just one instance
34     LBRACE("{"),
35     RBRACE("}"),
36     COMMA(","),
37     SEMICOLON(";"),
38     BECOMES("="),
39     LPAREN("("),
40     RPAREN(")"),
41     LBRACKET("["),
42     RBRACKET("]"),
43     DOT("."),
44     OR("|"),
45     AND("&"),
46     NOT("!"),
47     EQ("=="),

```

There is one **Symbols.X** enum constant object for each token and each nonterminal of the EBNF specification of TinyJ:

For each *token*, the name **X** will be **all uppercase**. **Examples:**

Symbols.WHILE (line 12) Symbols.BECOMES (line 38)
 Symbols.EQ (line 47) Symbols.IDENT (line 60)

For each *nonterminal*, the name **X** will be **NT followed by the name of the nonterminal**. **Examples:**

Symbols.NTprogram (line 72) Symbols.NTwhileStmt (line 86)

For each **Symbols.X** enum constant object, the field

X.symbolRepresentationForOutputFile (see line 100)

is set to the string passed as 1st argument of the constructor call for **Symbols.X**---see line 106.

A call **TJ.output.printSymbol(Symbols.X);** or a call **TJ.output.printSymbol(Symbols.X, null);** will print the string **X.symbolRepresentationForOutputFile** to the **output file** as a node of the sideways parse tree (with the correct indentation), unless **X** is **NONE**.

The calls **TJ.output.printSymbol(Symbols.X, 6);** and **TJ.output.printSymbol(Symbols.X, "mouse");** will respectively print the strings

X.symbolRepresentationForOutputFile + ": 6" and **X.symbolRepresentationForOutputFile + ": mouse"** to the **output file** as a parse tree node.

```
48     NE("!="),
49     GT(">"),
50     LT("<"),
51     GE(">="),
52     LE("<="),
53     TIMES("*"),
54     DIV("/"),
55     MOD("%"),
56     PLUS("+"),
57     MINUS("-"),
58
59     // TinyJ tokens that have more than one instance
60     IDENT("IDENTIFIER"),
61     UNSIGNEDINT("UNSIGNED INTEGER LITERAL"),
62     CHARSTRING("CHARACTER STRING LITERAL"),
63
64     // Fictitious tokens
65     ENDOFINPUT("EOF"),
66     BADTOKEN("????????? BAD TOKEN"),
67     EMPTY("<empty>"),
68     NONE(""),
69
70
71     // Nonterminals
72     NTprogram("<program>"),
73     NTimport("<importStmt>"),
74     NTdataFieldDecl("<dataFieldDecl>"),
75     NTvarDecl("<varDecl>"),
76     NTsingleVarDecl("<singleVarDecl>"),
77     NTmainDecl("<mainDecl>"),
78     NTmethodDecl("<methodDecl>"),
79     NTparameterDeclList("<parameterDeclList>"),
80     NTparameterDecl("<parameterDecl>"),
81     NTcompoundStmt("<compoundStmt>"),
82     NTstatement("<statement>"),
83     NTassignmentOrInvoc("<assignmentOrInvoc>"),
84     NTargumentList("<argumentList>"),
85     NTifStmt("<ifStmt>"),
86     NTwhileStmt("<whileStmt>"),
87     NToutputStmt("<outputStmt>"),
88     NTprintArgument("<printArgument>"),
89     NTexpr7("<expr7>"),
90     NTexpr6("<expr6>"),
91     NTexpr5("<expr5>"),
92     NTexpr4("<expr4>"),
93     NTexpr3("<expr3>"),
94     NTexpr2("<expr2>"),
```

```
95     NExpr1("<expr1>");
96
97
98     static final EnumSet<Symbols> reservedWords = EnumSet.range(INT, SCANNER);
99
100    public final String symbolRepresentationForOutputFile;
101
102    final String reservedWordSpelling;
103
104    Symbols(String symbolRepresentationForOutputFile, String reservedWordSpelling)
105    {
106        this.symbolRepresentationForOutputFile = symbolRepresentationForOutputFile;
107        this.reservedWordSpelling = reservedWordSpelling;
108    }
109
110    Symbols(String symbolRepresentationForOutputFile)
111    { this(symbolRepresentationForOutputFile, null); }
112 }
```

Input File (a TinyJ program):

```

<singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
    <expr2>
      <expr1>
        IDENTIFIER: input
        .
        Reserved Word: nextInt
        (
        )
        ... node has no more children
        ... node has no more children
        ... node has no more children
        ... node has no more children
      ;
    ... node has no more children
    ... node has no more children
  <statement>
  <assignmentOrInvoc>
    IDENTIFIER: x
    =
    <expr3>
      <expr2>
        <expr1>
          IDENTIFIER: x
          ... node has no more children
        null) %
          printed by TJ.output.printSymbol(Symbols.UNSIGNEDINT, 3)
          <expr1>
            UNSIGNED INTEGER LITERAL: 3
            ... node has no more children
            ... node has no more children
            pth() ... node has no more children
          ;
        ... node has no more children
        ... node has no more children
      <statement>
      <outputStmt>
        Reserved Word: System
        .
        Reserved Word: out
        III) .
        Reserved Word: println
        (
        <printArgument>
        <expr3>
        <expr2>
        <expr1>
          IDENTIFIER: x
          ... node has no more children
          pth() ... node has no more children
        +
        <expr2>
        <expr1>
          UNSIGNED INTEGER LITERAL: 2
          ... node has no more children
          ... node has no more children
          ... node has no more children
          ... node has no more children
        )
      ;
    ... node has no more children
    ... node has no more children
  }
  ... node has no more children
  ... node has no more children
  ... node has no more children

```

```
treeDepth = 0 <program> ← printed by TJ.output.printSymbol(Symbols.NTprogram)
treeDepth = 1 <importStmt> ← printed by TJ.output.printSymbol(Symbols.NTimport)
treeDepth = 2   Reserved Word: import ←
                Reserved Word: java
                . ← printed by TJ.output.printSymbol(Symbols.IMPORT, null)
                Reserved Word: util
                .
                Reserved Word: Scanner
                ; ← printed by TJ.output.printSymbol(Symbols.SEMICOLON, null)
                ... node has no more children printed by TJ.output.decTreeDepth()
treeDepth = 1   Reserved Word: class
                IDENTIFIER: Simple2
                { ← printed by TJ.output.printSymbol(Symbols.LBRACE, null)
                <dataFieldDecl>
treeDepth = 2   Reserved Word: static
                <varDecl> ← printed by TJ.output.printSymbol(Symbols.NTvarDecl)
treeDepth = 3   Reserved Word: Scanner
                IDENTIFIER: input
                = ← printed by TJ.output.printSymbol(Symbols.NEW, null)
                Reserved Word: new
                Reserved Word: Scanner
                (
                Reserved Word: System
                .
                Reserved Word: in
                )
                ;
                ... node has no more children printed by TJ.output.decTreeDepth()
treeDepth = 2   ... node has no more children printed by TJ.output.decTreeDepth()
treeDepth = 1 <mainDecl>
treeDepth = 2   Reserved Word: public
                Reserved Word: static
                Reserved Word: void
                Reserved Word: main
                (
                Reserved Word: String
                IDENTIFIER: args
                [
                ]
                )
                <compoundStmt>
treeDepth = 3   {
                <statement>
                <varDecl>
treeDepth = 4   Reserved Word: int
treeDepth = 5
```


The following 6 steps are needed *only if* you are interested in the possibility of doing TinyJ assignments on your PC or Mac rather than **mars**. (**Important:** Regardless of where you do the assignments, **you must test your code in your xxxxx_yyyy316 mars account and submit each assignment on mars using that account.**) Many students in previous semesters were able to do the TinyJ assignments on a PC or Mac, but I do not guarantee that you will be able to do so: *Students who try to do the assignments on a PC or Mac must be prepared to switch to working in their xxxxx_yyyy316 accounts on mars if they run into difficulties.*

- Open a powershell / terminal window on your PC / Mac and enter the following at its prompt: **javac -version**
If you get an error message after entering **javac -version**, or if the major version number that is printed is older than **11**, install a new version of the Java JDK—e.g., JDK 25 from <https://www.oracle.com/technetwork/java/javase/downloads/index.html>. (After installing the JDK on a PC, update the PC's System PATH environment variable so its first directory is the directory that contains the JDK's **jar.exe** application; for a typical installation of JDK 25, **c:\program files\java\jdk-25\bin** is the directory that should be added to your PC's System PATH. See, e.g., <https://www.computerhope.com/issues/ch000549.htm> if you don't know how to edit your PC's System PATH.)
- In the powershell / terminal window, enter the following: **mkdir ~/316java**
- Make **~/316java** your working directory by entering the following in the powershell / terminal window: **cd ~/316java**
- This step assumes your PC / Mac is connected to the qwifi-secured wireless network or connected to the Queens College VPN.** Use an scp or sftp client to copy **TJ1asn.jar** from the home directory of your **xxxxx_yyyy316** account on **mars** into the **~/316java** folder. If **~/316java** is your working directory in the powershell / terminal window (see step 5), you can do this by entering the following in that window: **scp xxxxx_yyyy316@mars.cs.qc.cuny.edu:TJ1asn.jar .** Here **xxxxx_yyyy316** means your **mars** username. *Note the space followed by a period at the end of this command!*
- Enter the following *two* commands in the powershell / terminal window:


```
jar xvf TJ1asn.jar
javac -cp . TJ1asn/TJ.java
```
- Enter the appropriate one of the following commands in the powershell / terminal window:


```
On a PC:      java -cp "TJ1solclasses;." TJ1asn.TJ CS316ex12.java 12.sol
On a Mac:     java -cp TJ1solclasses:. TJ1asn.TJ CS316ex12.java 12.sol
```

*This command executes my solution to this assignment with **CS316ex12.java** as the input file and **12.sol** as the output file. A listing of **CS316ex12.java** should be displayed on the screen, and **12.sol** should contain a sideways representation of the program's parse tree afterwards. There should not be any error message. To view the parse tree, you can enter the command **more 12.sol** on a PC or **less 12.sol** on a Mac.*

Important Files That will be Available to You After You Have Done Steps 1 and 2 Above

From your **TJ1asn** directory on **mars**:

OutputFileHandler.java.txt **Parser.java.txt** **SourceFileErrorException.java.txt** **TJ.java.txt**

From your **TJ1exer** directory on **mars** (the **l** in **TJ1exer** is the letter **l**, not the digit **1**):

LexicalAnalyzer.java.txt **SourceHandler.java.txt** **Symbols.java.txt**

These are the source files of the program, with **line numbers added**. (The actual source files (without line numbers) are in the same directories and have the same names, **but their extension is .java.**) The files can be viewed on **mars** using the **less** file viewer—e.g., enter the command **less TJ1asn/Parser.java.txt** to view **Parser.java.txt**, and enter the command **less TJ1exer/Symbols.java.txt** to view **Symbols.java.txt**.

If you have done steps 3 – 8 above, the same files will be in **~/316java/TJ1asn** and **~/316java/TJ1exer** on your PC or Mac; they can be viewed using **less** on a Mac (e.g., enter **less ~/316java/TJ1asn/Parser.java.txt** in a terminal window on a Mac to view **Parser.java.txt**) and can be viewed using, e.g., Notepad++ or VS Code on a PC.

How to Execute My Solution to This Assignment

Recall that the *output file* will contain the parse tree!

Step 1 put 16 files named **CS316exk.java** (**k = 0 – 15**) into the home directory of your **xxxxx_yyyy316** account on **mars**. These are all valid TinyJ source files. If you did step 7, it will have put copies of the same 16 files on your PC or Mac.

You should be able to execute *my* solution to this assignment on **mars** by entering the following command:

java -cp TJ1solclasses:. TJ1asn.TJ TinyJ-source-file-name output-file-name

[Your current working directory has to be your home directory for this to work.] **args[0]** **args[1]**

If you have done steps 3 – 8 on a Mac, then the above command should also work in a terminal window on your Mac if your working directory is **~/316java** (see step 5). If you have done steps 3 – 8 on a PC, then the following similar command (which has **;.** instead of **:.**) should work in a powershell window on your PC if **~/316java** is your working directory:

java -cp "TJ1solclasses;." TJ1asn.TJ TinyJ-source-file-name output-file-name

See steps 2 and 8 above for concrete examples of these commands!

EXAMPLES:

java -cp TJ1solclasses:. TJ1asn.TJ CS316ex12.java 12.sol (on mars or a Mac)
java -cp "TJ1solclasses;." TJ1asn.TJ CS316ex12.java 12.sol (on a PC)

How to Do TinyJ Assignment 1

The file `TJ1asn/Parser.java` is incomplete. It was produced by taking a complete version of that file and replacing parts of the code with comments of the following two forms:

```
/* ???????? */           or (in two places)    /* ????????
                                     default: throw ...
*/
```

To complete this assignment, replace every such comment in `TJ1asn/Parser.java` with appropriate code, and recompile the file. On **mars**, you can use the **nano**, **vim**, or **emacs** editor to edit the file; **nano** or **vim** could also be used on a Mac in a terminal window. If you are working on your PC, do not use Notepad as your editor; you can use Notepad++ or VS Code. (For the second type of comment, the appropriate code should include the `default: throw ...` statement.)

Do **not** put `Parser.java` or `Parser.class` into any directory other than `TJ1asn`. Do **not** change or move other `.java` and `.class` files.

To recompile `TJ1asn/Parser.java` after editing it, enter the following command:

```
javac -cp . TJ1asn/Parser.java
```

IMPORTANT: If you are doing this on **mars**, your current working directory has to be your home directory. If you are doing this on your PC or Mac (in a powershell / terminal window), your working directory has to be `~/316java` (see installation step 5); otherwise `javac` will not be able to find other classes that are used in `Parser.java`!

How to Test Your Solution

To test your completed version of `Parser.java`, first recompile it using `javac -cp . TJ1asn/Parser.java` and then execute `TJ1asn.TJ` with each of the 16 files `CS316exk.java` ($k = 0 - 15$) as the TinyJ source file and `k.out` as the output file, as follows: `java -cp . TJ1asn.TJ CS316exk.java k.out`

If you are doing this on **mars**, your current working directory has to be your home directory. If you are doing this on your PC or Mac (in a powershell / terminal window), your working directory has to be `~/316java` (see installation step 5).

If your program is correct then in each case the output file `k.out` should be identical to the output file `k.sol` that is produced by running my solution with the same source file as follows:

```
java -cp TJ1solclasses:. TJ1asn.TJ CS316exk.java k.sol [on mars or a Mac]
java -cp "TJ1solclasses;." TJ1asn.TJ CS316exk.java k.sol [on a PC]
```

On **mars** or a Mac, you can use `diff -c` to compare the output files produced by your and my solutions. (This outputs a report of the differences, if any, between the two files.) On a PC, you can use `fc.exe /n` instead. For example, the commands `diff -c k.sol k.out > k.dif` [on mars or a Mac] and `fc.exe /n k.sol k.out > k.dif` [on a PC] output to `k.dif` the differences between `k.sol` and `k.out`. (You can view `k.dif` using the command `less k.dif` on **mars** or a Mac, or using the command `more k.dif` on a PC. If your solution is correct, then the file `k.dif` should contain nothing if it was produced by `diff -c` or contain "FC: no differences encountered" if it was produced by `fc.exe /n`.)

How to Submit a Solution to This Assignment (Note that step 4 is very important!)

This assignment is to be submitted *no later than* the due date stated on p. 1. [Note: If **mars** fails to operate normally or becomes inaccessible at any time after **6 p.m.** on the due date, the submission deadline will not be extended. Try to submit no later than noon that day, and on an earlier day if possible.] To submit:

1. Add a comment at the beginning of your completed version of `Parser.java` that gives your name and the names of the students you worked with (if any). As usual, you may work with up to two other students, but see the remarks about this on p. 3 of the first-day announcements document.
2. Leave your completed version of `Parser.java` in the `TJ1asn` directory of your `xxxxx_yyyy316` account on **mars**. When two or three students work together, each of the students must leave his/her completed file in his/her TJ1asn directory on mars. If you did this assignment on your PC / Mac, you can copy the file `TJ1asn/Parser.java` from that machine to your `TJ1asn` directory on **mars** by following the instructions on the next page.
3. Enter the following command at the `xxxxx_yyyy316@mars:~$` prompt: `less TJ1asn/Parser.java`
Check that this displays the beginning of your final version of `Parser.java` (including the comment you added at step 1)!
If you copied `Parser.java` from your PC / Mac to **mars**, then be sure to test your code on mars—see the **How to Test Your Solution** instructions above.
4. Enter the following command at the `xxxxx_yyyy316@mars:~$` prompt: `submit_TJ_asn1`

Note: If your submitted version of `Parser.java` cannot even be compiled without error in on **mars**, then you will receive no credit at all for your submission!

The paragraph on p. 3 of the 1st-day-announcements document that begins with "If when I compute a student's course grade ..." explains how you can verify that this assignment has been submitted.

**A Way to Copy TJ1asn/Parser.java from a PC / Mac
to the TJ1asn Directory of Your xxxxx_yyyy316 mars Account**

NOTE: Steps (i) and (ii) below assume you have already done installation step 1 on page 2. If you haven't, then do installation step 1 on page 2 before following the instructions below.

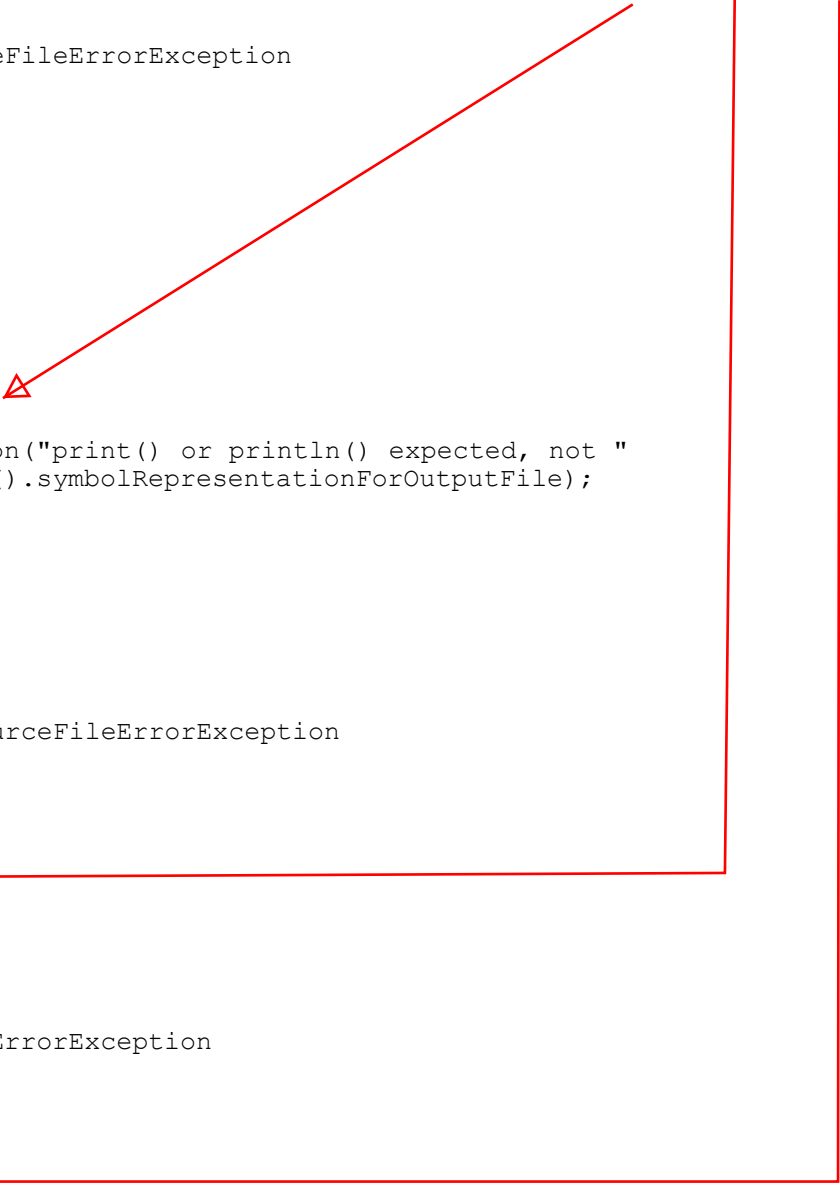
- (i) Open a powershell / terminal window on your PC / Mac and enter the following command at its prompt:
`cd ~/316java`
- (ii) **This step assumes your PC/Mac is connected to the qwifi-secured wireless network or connected to the QC VPN.**
Enter the following command in the powershell / terminal window (where **xxxxx_yyyy316** means the username of your **mars** account for this course):
`scp TJ1asn/Parser.java xxxxx_yyyy316@mars.cs.qc.cuny.edu:TJ1asn`
You will be asked to enter the password of your **xxxxx_yyyy316 mars** account.

IMPORTANT: After doing (i) and (ii), be sure to also do steps 3 and 4 of the submission instructions on the previous page. The assignment will **NOT** be submitted until you do step 4!

3 examples of ???????? comments in Parser.java
that you need to replace with appropriate code.

Parser.java.txt

```
283
284 private static void outputStmt() throws SourceFileErrorException
285 {
286     TJ.output.printSymbol (NToutputStmt);
287     TJ.output.incTreeDepth();
288
289     accept (SYSTEM);
290     accept (DOT);
291     accept (OUT);
292     accept (DOT);
293
294     switch (getCurrentToken()) {
295         /* ????????
296
297         default: throw new SourceFileErrorException("print() or println() expected, not "
298                                     + getCurrentToken().symbolRepresentationForOutputFile);
299     */
300     }
301
302     TJ.output.decTreeDepth();
303 }
304
305 private static void printArgument() throws SourceFileErrorException
306 {
307     TJ.output.printSymbol (NTprintArgument);
308     TJ.output.incTreeDepth();
309
310     /* ???????? */
311
312     TJ.output.decTreeDepth();
313 }
314
315 private static void expr7() throws SourceFileErrorException
316 {
317     TJ.output.printSymbol (NTexpr7);
318     TJ.output.incTreeDepth();
319
320     /* ???????? */
321
322     TJ.output.decTreeDepth();
323 }
324
325
326
327
328
329
```



```

1  package TJ1asn;
2
3  import TJlexer.SourceHandler;
4  import TJlexer.LexicalAnalyzer;
5  import TJlexer.Symbols;
6  import java.io.PrintWriter;
7
8  public final class TJ {
9      public static SourceHandler input;
10     public static OutputFileHandler output;
11
12     public static final int DATA_MEMORY_SIZE = 20000;
13     public static final int data[] = new int [DATA_MEMORY_SIZE];
14                                     /* space for string literals */
15
16     public static void main(String args[])
17     {
18         final String inputFileName = args.length == 0 ? null : args[0];
19         final String outputFileName = args.length <= 1 ? null : args[1];
20
21         try {
22             output = new OutputFileHandler(outputFileName);
23             input = new SourceHandler(inputFileName);
24
25             LexicalAnalyzer.setIO(input, output);
26             LexicalAnalyzer.setStringTable(data);
27             LexicalAnalyzer.nextToken();
28
29             Parser.program();
30
31             if (LexicalAnalyzer.getCurrentToken() != Symbols.ENDOFINPUT)
32                 throw new SourceFileErrorException("Token encountered after end of program");
33
34         } catch (SourceFileErrorException theError) {
35             System.out.println("\n\nERROR!  " + theError.errorMessage);
36             if (input != null) {
37                 if (input.getCurrentChar() != SourceHandler.eofDesignator)
38                     System.out.println("input.currentChar = " + (char) input.getCurrentChar() + '\');
39                 else
40                     System.out.println("input.currentChar = EOF");
41                 System.out.print("LexicalAnalyzer.currentToken = ");
42                 TJ.output.outputSymbol(LexicalAnalyzer.getCurrentToken(), LexicalAnalyzer.getTokenValue(),
43                                         new PrintWriter(System.out, true));
44                 System.out.println();
45             }
46         } finally {
47             if (output != null)

```

the "TJ.output" in TJ.output.printSymbol(...) etc.

The most important line: This tries to read a <program> from the input file and output the parse tree of that <program>.

Error messages are printed to the screen by this exception handler.

A TinyJ Input File with a Syntax Error, and the Output That is Generated by a Solution to TinyJ Assignment 1

Input File (the error is that `public` should be preceded by a semicolon):

```
import java.util.Scanner;

class Simple2 {

    static Scanner input = new Scanner(System.in)

    public static void main(String args[])
    {
        int x = input.nextInt();
        x = x % 3;
        System.out.println(x + 2);
    }
}
```

Output on the Screen (shows the tokens that are read before the error is detected):

```
1:  import java.util.Scanner;
2:
3:  class Simple2 {
4:
5:      static Scanner input = new Scanner(System.in)
6:
7:      public
```

```
ERROR!  Something's wrong--maybe the following token is missing: ;
input.currentChar = ' '
LexicalAnalyzer.currentToken = Reserved Word: public
```

```
11 public final class Parser {
12
13     private static void accept (Symbols expectedToken) throws SourceFileErrorException
14     {
15         if (getCurrentToken() == expectedToken)
16             nextToken();
17         else throw new SourceFileErrorException("Something's wrong--maybe the following token is missing: "
18                                                 + expectedToken.symbolRepresentationForOutputFile);
19     }
```

```

1  package TJlasn;
2
3  import java.io.*;
4  import java.util.Scanner;
5  import TJlexer.Symbols;
6
7
8  public class OutputFileHandler {
9
10     protected PrintWriter outFileWriter;
11
12     public final PrintWriter getOutFileWriter()
13     {
14         return outFileWriter;
15     }
16
17     protected int treeDepth = 0;
18
19     public final int getTreeDepth() {
20         return treeDepth;
21     }
22
23     public final void incTreeDepth() {
24         treeDepth++;
25     }
26
27     public final void decTreeDepth() {
28         for (int i = 0; i < treeDepth; i++)
29             outFileWriter.print(" ");
30         outFileWriter.println("... node has no more children");
31         treeDepth--;
32     }
33
34
35     public final void printSymbol(Symbols nodeName)  called by Parser.java's parsing methods to print nonterminals
36     {
37         printSymbol(nodeName, null);
38     }
39
40
41     public final void printSymbol(Symbols nodeName, Object nodeValue)  called by nextToken() to print currentToken
42     {
43         if (nodeName != Symbols.NONE) {
44             for (int i = 0; i < treeDepth; i++)  prints treeDepth spaces
45                 outFileWriter.print(" ");
46             outputSymbol(nodeName, nodeValue, outFileWriter);
47         }

```



```
48     }
49
50
51     public void outputSymbol(Symbols nodeName, Object nodeValue, PrintWriter out)
52     {
53         out.print(nodeName.symbolRepresentationForOutputFile);
54
55         if (nodeValue == null)
56             out.println();
57         else
58             out.println(": " + nodeValue);
59     }
60
61
62     protected final void openOutputFile (String filename) throws SourceFileErrorException
63     {
64         System.out.print("Enter name for output file: ");
65         if (filename != null)
66             System.out.print(filename+"\n\n");
67         else {
68             System.out.flush();
69             filename = (new Scanner(System.in)).nextLine();
70             System.out.println();
71         }
72         try {
73             outFileWriter = new PrintWriter(new FileWriter(filename));
74         }
75         catch (IOException e) {
76             throw new SourceFileErrorException("Failed to open output file");
77         }
78     }
79
80
81     protected OutputFileHandler(String filename) throws SourceFileErrorException
82     {
83         openOutputFile(filename);
84     }
85 }
86
87
88
89
```

A Correct TinyJ Input File and the Corresponding Parse Tree That is Written to the Output File by a Solution to TinyJ Assignment 1

Input File (a TinyJ program):

```
import java.util.Scanner;

class Simple2 {

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

    public static void main(String args[])
    {
        int x = input.nextInt();
        x = x % 3;
        System.out.println(x + 2);
    }
}
```

```
<singleVarDecl>
  IDENTIFIER: x
  =
  <expr3>
    <expr2>
      <expr1>
        IDENTIFIER: input
        .
        Reserved Word: nextInt
        (
          )
          ... node has no more children
          ... node has no more children
          ... node has no more children
          ... node has no more children
        ;
        ... node has no more children
        ... node has no more children
```

Output File (the above program's parse tree, in sideways representation):

Each line of the output file / parse tree is printed by 1 call of TJ.output.printSymbol(...) or by 1 call of TJ.output.decTreeDepth().

```
treeDepth = 0 <program> ← printed by TJ.output.printSymbol(Symbols.NTprogram)
treeDepth = 1 <importStmt> ← printed by TJ.output.printSymbol(Symbols.NTimport)
treeDepth = 2 Reserved Word: import ← printed by TJ.output.printSymbol(Symbols.IMPORT, null)
Reserved Word: java
.
Reserved Word: util
.
Reserved Word: Scanner
; ← printed by TJ.output.printSymbol(Symbols.SEMICOLON, null)
... node has no more children printed by TJ.output.decTreeDepth()
treeDepth = 1 Reserved Word: class
IDENTIFIER: Simple2
{ ← printed by TJ.output.printSymbol(Symbols.LBRACE, null)
<dataFieldDecl>
treeDepth = 2 Reserved Word: static
<varDecl> ← printed by TJ.output.printSymbol(Symbols.NTvarDecl)
Reserved Word: Scanner
IDENTIFIER: input
= ← printed by TJ.output.printSymbol(Symbols.NEW, null)
Reserved Word: new
Reserved Word: Scanner
(
Reserved Word: System
.
Reserved Word: in
)
;
... node has no more children printed by TJ.output.decTreeDepth()
... node has no more children printed by TJ.output.decTreeDepth()
treeDepth = 2 <mainDecl>
treeDepth = 1 Reserved Word: public
Reserved Word: static
Reserved Word: void
Reserved Word: main
(
Reserved Word: String
IDENTIFIER: args
[
]
)
<compoundStmt>
treeDepth = 3 {
<statement>
treeDepth = 4 <varDecl>
treeDepth = 5 Reserved Word: int
```

```
<statement>
<assignmentOrInvoc>
  IDENTIFIER: x
  =
  <expr3> ← printed by calls of TJ.output.printSymbol(Symbols.IDENT, "x")
  <expr2>
    <expr1>
      IDENTIFIER: x
      ... node has no more children
      % ← printed by TJ.output.printSymbol(Symbols.UNSIGNEDINT, 3)
      <expr1>
        UNSIGNED INTEGER LITERAL: 3
        ... node has no more children
        ... node has no more children
        ... node has no more children
      ;
      ... node has no more children
      ... node has no more children
    <statement>
    <outputStmt>
      Reserved Word: System
      .
      Reserved Word: out
      .
      Reserved Word: println
      (
      <printArgument>
      <expr3>
      <expr2>
      <expr1>
        IDENTIFIER: x
        ... node has no more children
        ... node has no more children
      +
      <expr2>
      <expr1>
        UNSIGNED INTEGER LITERAL: 2
        ... node has no more children
        ... node has no more children
        ... node has no more children
        ... node has no more children
      )
      ;
      ... node has no more children
      ... node has no more children
    }
    ... node has no more children
    ... node has no more children
  }
  ... node has no more children
```

```

1  package TJlexer;
2
3  import static TJlexer.Symbols.*;  allows Symbols.SEMICOLON to be written as just SEMICOLON, etc.
4
5  import TJlasn.OutputFileHandler;
6  import TJlasn.SourceFileErrorException;
7
8  public final class LexicalAnalyzer {
9
10     private static SourceHandler input;
11     private static OutputFileHandler output;
12     private static int stringTable[];
13
14     public static void setIO(SourceHandler sourceHandler, OutputFileHandler outputFileHandler) {
15         input = sourceHandler;
16         output = outputFileHandler;
17     }
18
19     public static void setStringTable(int[] tbl) {
20         stringTable = tbl;
21     }
22
23     private static Symbols currentToken = NONE;
24
25     public static Symbols getCurrentToken() {
26         currentTokenNeedsToBeInspected = false;
27         return currentToken;
28     }
29
30     private static boolean currentTokenNeedsToBeInspected;
31
32     private static int currentValue;    // numerical value of UNSIGNEDINT token
33
34     public static int getCurrentValue() {
35         return currentValue;
36     }
37
38
39     private static String currentSpelling;    // spelling of IDENT
40
41     public static String getCurrentSpelling() {
42         return currentSpelling;
43     }
44
45
46     private static int startOfString;
47     private static int endOfString = -1;

```

Called by the
main() method
of TJ.java.

currentToken contains the Symbols.X constant which
represents the TinyJ token that this program "is now looking at".
getCurrentToken() is the public getter method for currentToken.

```

48
49
50 public static int getStartOfString() {
51     return startOfString;
52 }
53
54 public static int getEndOfString() {
55     return endOfString;
56 }
57
58 public static void setEndOfString(int addr) { // called in ParserAndTranslator's program() method
59     endOfString = addr;
60 }
61
62
63 private static Object tokenValue; // passed to output.printSymbol() at start of nextToken();
64                                   // contains information used to output the currentToken
65
66 public static Object getTokenValue() { return tokenValue; }
67
68
69 public static void nextToken() throws SourceFileErrorException
70 {
71     output.printSymbol(currentToken, tokenValue);
72     if (currentTokenNeedsToBeInspected)
73         throw new SourceFileErrorException("Internal error in parser: Token discarded without being inspected");
74     else
75         currentTokenNeedsToBeInspected = true;
76
77     StringBuilder currentTokenString = new StringBuilder(10);
78
79     while (input.getCurrentChar() == ' ') input.nextChar();
80
81     tokenValue = null;
82
83     if (Character.isLetter((char) input.getCurrentChar())
84         || input.getCurrentChar() == '_'
85         || input.getCurrentChar() == '$') {
86         /* identifier or reserved word */
87
88         do {
89             currentTokenString.append((char) input.getCurrentChar());
90             input.nextChar();
91         } while (Character.isLetterOrDigit((char) input.getCurrentChar())
92                 || input.getCurrentChar() == '_'
93                 || input.getCurrentChar() == '$');
94

```

A successful call of **nextToken()** does the following:

1. Prints **currentToken** to the *output file*.
2. Skips white space in the input file, then reads the characters of the next token from the input file and **sets currentToken to the Symbols.X constant which represents that token**.
3. Sets tokenValue appropriately if that token is an *identifier, string literal, or unsigned int literal* token; otherwise tokenValue will be null (from line 81).

```
95     currentSpelling = currentTokenString.toString();
96
97     for (Symbols resWord : Symbols.reservedWords) {
98         if (currentSpelling.equals(resWord.reservedWordSpelling)) {
99             currentToken = resWord; return;
100     }
101 }
102 currentToken = IDENT;
103 tokenValue = currentSpelling;
104 return;
105 } /* identifier or reserved word */
106
107 else {
108     switch (input.getCurrentChar()) {
109         case '0': /* unsigned integer 0 */
110             currentToken = UNSIGNEDINT; tokenValue = currentValue = 0; input.nextChar(); return;
111
112         case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
113             currentToken = UNSIGNEDINT;
114             do {
115                 currentTokenString.append((char) input.getCurrentChar());
116                 input.nextChar();
117             } while (Character.isDigit(input.getCurrentChar()));
118             tokenValue = currentValue = Integer.parseInt(currentTokenString.toString());
119
120             return;
121
122         case '"':
123             currentToken = CHARSTRING;
124             startOfString = endOfString + 1;
125
126             int lineNum = input.getSourceFileReader().getLineNumber();
127
128             input.nextChar();
129
130             int c;
131
132             while ((c = input.getCurrentChar()) != '"') {
133
134                 if (c == SourceHandler.eofDesignator)
135                     throw new SourceFileErrorException("End of file occurred within a string.");
136                 else if (c == '\\') {
137                     input.nextChar();
138                     switch (input.getCurrentChar()) {
139                         case 'n': c = '\n'; break;
140                         case '\\': c = '\\'; break;
141                         case '"': c = '"'; break;
```

Looks in a table of reserved word tokens to see if the current identifier or reserved word's spelling is the same as that of a reserved word. If not, then this token is an IDENTIFIER.

```
142         default: throw new SourceFileErrorException("Illegal escape character.");
143     }
144 }
145
146     currentTokenString.append((char) c);
147     stringTable[++endOfString] = c;
148     input.nextChar();
149 }
150 if (input.getSourceFileReader().getLineNumber() != lineNum)
151     throw new SourceFileErrorException("Multi-line string literals are not allowed.");
152
153     tokenValue = '"' + currentTokenString.toString() + '"';
154
155     input.nextChar();
156
157     return;
158
159 case '=':
160     input.nextChar();
161     if (input.getCurrentChar() == '=') {
162         currentToken = EQ;
163         input.nextChar();
164     }
165     else currentToken = BECOMES;
166
167     return;
168
169 case '!':
170     input.nextChar();
171     if (input.getCurrentChar() == '=') {
172         currentToken = NE;
173         input.nextChar();
174     }
175     else currentToken = NOT;
176
177     return;
178
179 case '<':
180     input.nextChar();
181     if (input.getCurrentChar() == '=') {
182         currentToken = LE;
183         input.nextChar();
184     }
185     else currentToken = LT;
186
187     return;
188
```

```
189         case '>':
190             input.nextChar();
191             if (input.getCurrentChar() == '=') {
192                 currentToken = GE;
193                 input.nextChar();
194             }
195             else currentToken = GT;
196
197         return;
198
199         case '+':
200             input.nextChar();
201             if (input.getCurrentChar() == '+') {
202                 currentToken = BADTOKEN;
203                 tokenValue = "\"++\"";
204                 throw new SourceFileErrorException("Unrecognized token: " + tokenValue);
205             }
206             else currentToken = PLUS;
207
208         return;
209
210         case '-':
211             input.nextChar();
212             if (input.getCurrentChar() == '-') {
213                 currentToken = BADTOKEN;
214                 tokenValue = "\"--\"";
215                 throw new SourceFileErrorException("Unrecognized token: " + tokenValue);
216             }
217             else currentToken = MINUS;
218
219         return;
220
221         case '&': currentToken = AND; input.nextChar(); return;
222         case '|': currentToken = OR; input.nextChar(); return;
223         case '{': currentToken = LBRACE; input.nextChar(); return;
224         case '}': currentToken = RBRACE; input.nextChar(); return;
225         case ',': currentToken = COMMA; input.nextChar(); return;
226         case ';': currentToken = SEMICOLON; input.nextChar(); return;
227         case '(': currentToken = LPAREN; input.nextChar(); return;
228         case ')': currentToken = RPAREN; input.nextChar(); return;
229         case '[': currentToken = LBRACKET; input.nextChar(); return;
230         case ']': currentToken = RBRACKET; input.nextChar(); return;
231         case '.': currentToken = DOT; input.nextChar(); return;
232         case '*': currentToken = TIMES; input.nextChar(); return;
233         case '/': currentToken = DIV; input.nextChar(); return;
234         case '%': currentToken = MOD; input.nextChar(); return;
235
```

```
236         case SourceHandler.eofDesignator:
237             currentToken = ENDOFINPUT;
238             return;
239
240         default:
241             currentToken = BADTOKEN;
242             tokenValue = "'" + (char) input.getCurrentChar() + "'";
243             throw new SourceFileErrorException("Unrecognized token: " + tokenValue);
244     }
245 }
246 } /* nextToken */
247
248 }
249
250
251
```

Examples of the Effects of Calls of nextToken():

Suppose the program is at the token instance `apple` in the following assignment statement: **`apple = 25;`**

Then **`currentToken == Symbols.IDENT`** and **`tokenValue == "apple"`**.

The next call of **`nextToken()`** will:

Call **`output.printSymbol(Symbols.IDENT, "apple")`**. This prints `IDENTIFIER: apple` to the output file (with an indentation of `treeDepth`).

Set **`currentToken`** to **`Symbols.BECOMES`**.

Set **`tokenValue`** to **`null`**.

The next call of **`nextToken()`** will:

Call **`output.printSymbol(Symbols.BECOMES, null)`**. This prints `=` to the output file (with an indentation of `treeDepth`).

Set **`currentToken`** to **`Symbols.UNSIGNEDINT`**.

Set **`tokenValue`** to **`25`**.

The next call of **`nextToken()`** will:

Call **`output.printSymbol(Symbols.UNSIGNEDINT, 25)`**. This prints `UNSIGNED INTEGER LITERAL: 25` (with indentation of `treeDepth`).

Set **`currentToken`** to **`Symbols.SEMICOLON`**.

Set **`tokenValue`** to **`null`**.


```

1  package TJlasm;
2
3  import static TJlexer.LexicalAnalyzer.getCurrentToken; allows LexicalAnalyzer.getCurrentToken() to be written as just getCurrentToken().
4  import static TJlexer.LexicalAnalyzer.nextToken; allows LexicalAnalyzer.nextToken() to be written as just nextToken().
5  import static TJlexer.Symbols.*; allows Symbols.SEMICOLON to be written as just SEMICOLON, etc.
6  import TJlexer.Symbols;
7
8
9  // ***** Recursive Descent Parser *****
10
11 public final class Parser {
12
13     private static void accept (Symbols expectedToken) throws SourceFileErrorException
14     {
15         if (getCurrentToken() == expectedToken) ←
16             nextToken();
17         else throw new SourceFileErrorException("Something's wrong--maybe the following token is missing: "
18             + expectedToken.symbolRepresentationForOutputFile);
19     }
20
21     static void program () throws SourceFileErrorException
22     {
23         TJ.output.printSymbol (NTprogram);
24         TJ.output.incTreeDepth();
25
26         if (getCurrentToken() == IMPORT) importStmt();
27
28         accept(CLASS);
29         accept(IDENT);
30         accept(LBRACE);
31
32         while (getCurrentToken() == STATIC)
33             dataFieldDecl();
34
35         mainDecl();
36
37         while (getCurrentToken() == STATIC)
38             methodDecl();
39
40         accept(RBRACE);
41
42         TJ.output.decTreeDepth();
43     }
44
45     private static void importStmt() throws SourceFileErrorException

```

A call of **accept(X)** does the following:

1. It checks that **currentToken == Symbols.X**.
2. Assuming **currentToken == Symbols.X** it calls **nextToken()**, which:
 - (a) Prints token **Symbols.X** to the output file.
 - (b) Reads in the next token from the input file and sets **currentToken** (& **tokenValue**) accordingly.

But if **currentToken != Symbols.X** when **accept(X)** is called, an exception is thrown and the program will terminate after printing an error message on the screen!

Calling **accept(X)** has the same effect as calling **nextToken()** unless **currentToken != Symbols.X**, in which case the program terminates with an error message.

Parser.java.txt

```
48  {
49    TJ.output.printSymbol (NTimport);
50    TJ.output.incTreeDepth();
51
52    accept (IMPORT);
53    accept (JAVA);
54    accept (DOT);
55    accept (UTIL);
56    accept (DOT);
57    accept (SCANNER);
58    accept (SEMICOLON);
59
60    TJ.output.decTreeDepth();
61  }
62
63
64  private static void dataFieldDecl() throws SourceFileErrorException
65  {
66    TJ.output.printSymbol (NTdataFieldDecl);
67    TJ.output.incTreeDepth();
68
69    accept (STATIC);
70    varDecl();
71
72    TJ.output.decTreeDepth();
73  }
74
75
76  private static void varDecl() throws SourceFileErrorException
77  {
78    TJ.output.printSymbol (NTvarDecl);
79    TJ.output.incTreeDepth();
80
81    if (getCurrentToken() == INT) {
82      nextToken();
83      singleVarDecl();
84      while (getCurrentToken() == COMMA) {
85        nextToken();
86        singleVarDecl();
87      }
88      accept (SEMICOLON);
89    }
90    else if (getCurrentToken() == SCANNER) {
91      nextToken();
92
93      if (getCurrentToken() == IDENT) {
94        nextToken();
```

RECURSIVE DESCENT PARSING

For each nonterminal **<n>** of the EBNF specification, Parser.java has a corresponding static **parsing method n()**.

When **n()** is called, it expects **currentToken** to be a possible first token* of an instance of **<n>**. If this is so, then the call of **n()** will (if possible):

1. Read in the tokens in the rest of an instance of **<n>**.
2. Output (with an indentation of **TJ.output.treeDepth**) a sideways parse tree, with root **<n>**, that generates the instance of **<n>** that is read.

On return from a successful call of **n()**, **currentToken** will be the first token after the instance of **<n>** that it read.

*If **<n>** can generate an empty string, as in the case **<n> = <parameterDeclList>**, then when **n()** is called **currentToken** might also be the first token after an empty sequence that is derived from **<n>**.

The body of a parsing method **n()** has the form

```
{
  TJ.output.printSymbol(NTn);
  TJ.output.incTreeDepth();

  ...

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

where the code in **...** should be derived from the EBNF definition of the nonterminal **<n>**.

IMPORTANT: See the

RecursiveDescentParsingCode-Slides

file on Brightspace for an example of how to write such a parsing method!

The arrows on [page 1](#) tell you which methods you must complete!

Also see the following pages of this document:

1. [A Mistake to Avoid When Doing TinyJ Assignment 1 \(p. 34\)](#)
2. [Debugging Hints for TinyJ Assignment 1 \(p. 36\)](#)
3. [An Old Exam Question \(p. 35\)](#)

```
95         }
96     else
97         throw new SourceFileErrorException("Scanner name expected");
98
99     accept(BECOMES);
100    accept(NEW);
101    accept(SCANNER);
102    accept(LPAREN);
103    accept(SYSTEM);
104    accept(DOT);
105    accept(IN);
106    accept(RPAREN);
107    accept(SEMICOLON);
108    }
109    else throw new SourceFileErrorException("\"int\" or \"Scanner\" expected");
110
111    TJ.output.decTreeDepth();
112 }
113
114
115 private static void singleVarDecl() throws SourceFileErrorException
116 {
117     TJ.output.printSymbol(NTsingleVarDecl);
118     TJ.output.incTreeDepth();
119
120     /* ???????? */
121
122     TJ.output.decTreeDepth();
123 }
124
125
126 private static void mainDecl() throws SourceFileErrorException
127 {
128     TJ.output.printSymbol(NTmainDecl);
129     TJ.output.incTreeDepth();
130
131     accept(PUBLIC);
132     accept(STATIC);
133     accept(VOID);
134     accept(MAIN);
135     accept(LPAREN);
136     accept(STRING);
137     accept(IDENT);
138     accept(LBRACKET);
139     accept(RBRACKET);
140     accept(RPAREN);
141 }
```

```
142     compoundStmt();
143
144     TJ.output.decTreeDepth();
145 }
146
147
148 private static void methodDecl() throws SourceFileErrorException
149 {
150     TJ.output.printSymbol(NTmethodDecl);
151     TJ.output.incTreeDepth();
152
153     /* ???????? */
154
155     TJ.output.decTreeDepth();
156 }
157
158
159 private static void parameterDeclList() throws SourceFileErrorException
160 {
161     TJ.output.printSymbol(NTparameterDeclList);
162     TJ.output.incTreeDepth();
163
164     if (getCurrentToken() == INT) {
165         parameterDecl();
166         while (getCurrentToken() == COMMA) {
167             nextToken();
168             parameterDecl();
169         }
170     }
171     else TJ.output.printSymbol(EMPTY);
172
173     TJ.output.decTreeDepth();
174 }
175
176
177 private static void parameterDecl() throws SourceFileErrorException
178 {
179     TJ.output.printSymbol(NTparameterDecl);
180     TJ.output.incTreeDepth();
181
182     accept(INT);
183     accept(IDENT);
184     while (getCurrentToken() == LBRACKET) {
185         nextToken();
186         accept(RBRACKET);
187     }
188 }
```

```
189     TJ.output.decTreeDepth();
190 }
191
192
193 private static void compoundStmt() throws SourceFileErrorException
194 {
195     TJ.output.printSymbol(NTcompoundStmt);
196     TJ.output.incTreeDepth();
197
198     /* ????????? */
199
200     TJ.output.decTreeDepth();
201 }
202
203
204 private static void statement() throws SourceFileErrorException
205 {
206     TJ.output.printSymbol(NTstatement);
207     TJ.output.incTreeDepth();
208
209     switch (getCurrentToken()) {
210         case SEMICOLON: nextToken(); break;
211         case RETURN: nextToken();
212                     if (getCurrentToken() != SEMICOLON)
213                         expr3();
214                     accept(SEMICOLON);
215                     break;
216         case INT: case SCANNER: varDecl(); break;
217         case IDENT: assignmentOrInvoc(); break;
218         case LBRACE: compoundStmt(); break;
219         case IF: ifStmt(); break;
220         case WHILE: whileStmt(); break;
221         case SYSTEM: outputStmt(); break;
222         default: throw new SourceFileErrorException("Expected first token of a <statement>, not "
223                                                     + getCurrentToken().symbolRepresentationForOutputFile);
224     }
225
226     TJ.output.decTreeDepth();
227 }
228
229
230 private static void assignmentOrInvoc() throws SourceFileErrorException
231 {
232     TJ.output.printSymbol(NTassignmentOrInvoc);
233     TJ.output.incTreeDepth();
234
235     /* ????????? */
```

```
236
237     TJ.output.decTreeDepth();
238 }
239
240
241 private static void argumentList() throws SourceFileErrorException
242 {
243     TJ.output.printSymbol(NTargumentList);
244     TJ.output.incTreeDepth();
245
246     /* ????????? */
247
248     TJ.output.decTreeDepth();
249 }
250
251
252 private static void ifStmt() throws SourceFileErrorException
253 {
254     TJ.output.printSymbol(NTifStmt);
255     TJ.output.incTreeDepth();
256
257     accept(IF);
258     accept(LPAREN);
259     expr7();
260     accept(RPAREN);
261
262     statement();
263
264     if (getCurrentToken() == ELSE) {
265         nextToken();
266         statement();
267     }
268
269     TJ.output.decTreeDepth();
270 }
271
272
273 private static void whileStmt() throws SourceFileErrorException
274 {
275     TJ.output.printSymbol(NTwhileStmt);
276     TJ.output.incTreeDepth();
277
278     /* ????????? */
279
280     TJ.output.decTreeDepth();
281 }
282
```

```
283
284 private static void outputStmt() throws SourceFileErrorException
285 {
286     TJ.output.printSymbol(NToutputStmt);
287     TJ.output.incTreeDepth();
288
289     accept(SYSTEM);
290     accept(DOT);
291     accept(OUT);
292     accept(DOT);
293
294     switch (getCurrentToken()) {
295
296         /* ????????
297
298         default: throw new SourceFileErrorException("print() or println() expected, not "
299                 + getCurrentToken().symbolRepresentationForOutputFile);
300
301         */
302     }
303
304     TJ.output.decTreeDepth();
305 }
306
307
308 private static void printArgument() throws SourceFileErrorException
309 {
310     TJ.output.printSymbol(NTprintArgument);
311     TJ.output.incTreeDepth();
312
313     /* ???????? */
314
315     TJ.output.decTreeDepth();
316 }
317
318
319 private static void expr7() throws SourceFileErrorException
320 {
321     TJ.output.printSymbol(NTexpr7);
322     TJ.output.incTreeDepth();
323
324     /* ???????? */
325
326     TJ.output.decTreeDepth();
327 }
328
329
```

```
330 private static void expr6() throws SourceFileErrorException
331 {
332     TJ.output.printSymbol(NTexpr6);
333     TJ.output.incTreeDepth();
334
335     /* ???????? */
336
337     TJ.output.decTreeDepth();
338 }
339
340
341 private static void expr5() throws SourceFileErrorException
342 {
343     TJ.output.printSymbol(NTexpr5);
344     TJ.output.incTreeDepth();
345
346     /* ???????? */
347
348     TJ.output.decTreeDepth();
349 }
350
351
352 private static void expr4() throws SourceFileErrorException
353 {
354     TJ.output.printSymbol(NTexpr4);
355     TJ.output.incTreeDepth();
356
357     /* ???????? */
358
359     TJ.output.decTreeDepth();
360 }
361
362
363 private static void expr3() throws SourceFileErrorException
364 {
365     TJ.output.printSymbol(NTexpr3);
366     TJ.output.incTreeDepth();
367
368     /* ???????? */
369
370     TJ.output.decTreeDepth();
371 }
372
373
374 private static void expr2() throws SourceFileErrorException
375 {
376     TJ.output.printSymbol(NTexpr2);
```



```
377     TJ.output.incTreeDepth();
378
379     expr1();
380
381     while (    getCurrentToken() == TIMES
382             || getCurrentToken() == DIV
383             || getCurrentToken() == MOD) {
384
385         nextToken();
386
387         expr1();
388     }
389
390     TJ.output.decTreeDepth();
391 }
392
393
394 private static void expr1() throws SourceFileErrorException
395 {
396     TJ.output.printSymbol(NTexpr1);
397     TJ.output.incTreeDepth();
398
399     switch (getCurrentToken()) {
400
401         /* ????????
402
403         default: throw new SourceFileErrorException("Malformed expression");
404
405         */
406     }
407
408     TJ.output.decTreeDepth();
409 }
410 }
411
412
```

A Mistake to Avoid When Doing TinyJ Assignment 1

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

1. When writing the method `argumentList()`, which should be based on the EBNF rule

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

it would be wrong to write:

```
accept(LPAREN);
if (getCurrentToken() == NExpr3) /* 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()`, one case you need to deal with relates to the following part of the 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>}] ')'
```

An Old Exam Question

A student is debugging his current version of Parser.java for TinyJ Assignment 1. He compiles his file and then runs his program as follows:

```
java -cp . TJlasn.TJ X.java X.out
```

He also runs the solution that was provided, as follows:

```
java -cp TJ1solclasses:. TJlasn.TJ X.java X.sol
```

The first difference between the output files X.out and X.sol is that X.sol has a comma on **line 567**, but this is missing in X.out. Lines 556 – 568 of X.sol and X.out are reproduced below with line numbers. (Lines 556 – 566 are the same in both output files.)

```
Lines 556 - 568 of X.sol [Output produced by java -cp TJ1solclasses:. TJlasn.TJ ...]:
556         <expr1>
557         IDENTIFIER: leq
558         <argumentList>
559         (
560         <expr3>
561         <expr2>
562         <expr1>
563         IDENTIFIER: size
564         ... node has no more children
565         ... node has no more children
566         ... node has no more children
567         ,
568         <expr3>
```

```
Lines 556 - 568 of X.out [Output produced by java -cp . TJlasn.TJ ...]:
556         <expr1>
557         IDENTIFIER: leq
558         <argumentList>
559         (
560         <expr3>
561         <expr2>
562         <expr1>
563         IDENTIFIER: size
564         ... node has no more children
565         ... node has no more children
566         ... node has no more children
567         <expr3>
568         <expr2>
```

Hint: In reading this output, recall that the indentation levels of consecutive lines are either the same or differ by just 1; thus line 567 has the same indentation as line 559.

Now answer the following two questions. In each case, *circle the correct choice*. [The answers are given on the next page.]

- (i) The output files show there is probably an error in the student's version of the method
(a) `expr1()` (b) `expr2()` (c) `expr3()` (d) `argumentList()` (e) `ifStmt()`
[1 pt.]
- (ii) Which one of the following changes might well fix this error?
(a) Insert a missing call of `accept (COMMA)` or `nextToken()` in the student's Parser.java.
(b) Delete a call of `accept (COMMA)` from the student's Parser.java.
(c) Delete a call of `nextToken()` from the student's Parser.java.
(d) Insert a missing call of `expr3()` in the student's Parser.java.
(e) Delete a call of `expr2()` from the student's Parser.java.
[1 pt.]

Debugging Hints for TinyJ Assignment 1

1. It is a very common mistake to omit a call of `accept (...)` or `nextToken ()`: For *each* token in the EBNF definition of a non-terminal $\langle N \rangle$, the body of the corresponding parsing method $N ()$ should contain a call of `accept (...)` or `nextToken ()` whose execution may cause that token to be output as a parse tree node. Another common mistake is to call `nextToken ()` when `accept (...)` should be called; this often produces the following error message:
Internal error in parser: Token discarded without being inspected
A third common mistake is to pass a `Symbols` object that represents a *non*-terminal as an argument to `accept (...)`, as in `accept (NTexpr7)`;—see [A Mistake to Avoid When Doing TinyJ Assignment 1](#) above.
2. The sideways parse tree in the output file can be regarded as an *execution trace* of your program, and can be useful when debugging your code! If your program is not working correctly, and you have produced both *k.sol* and *k.out* for some *k* (as described on page 4 of the assignment document), then the first line in *k.sol* that isn't in *k.out* shows "something my solution did that your program didn't do". (You can find that line from the output of `diff -c [on mars or a Mac]` or `fc.exe /n [on a PC]`.) When reading the output file for debugging purposes, bear the following in mind:
 - A. In a sideways parse tree, the parent of a node appears on the most recent previous line that has lower indentation. (Note that adjacent lines of the tree either have the same indentation or have indentation levels that differ by just 1.) For example, in the Old Exam Question, the parent of the comma on line 567 of *X.sol*, and of $\langle \text{expr3} \rangle$ on line 568, is $\langle \text{argumentList} \rangle$ on line 558.
 - B. Each non-terminal $\langle N \rangle$ in the output file is written when the corresponding parsing method $N ()$ is called (by the call of `TJ.output.printSymbol (...)` at the beginning of N 's body). The value of `getCurrentToken ()` at that time is shown by the first token in the output file *after* $\langle N \rangle$'s line. $\langle N \rangle$'s parent in the parse tree shows the caller of $N ()$. For example, in the Old Exam Question, $\langle \text{expr3} \rangle$ on line 560 of *X.sol* was written when `expr3 ()` was called. The value of `getCurrentToken ()` was `IDENT` at the time of the call (as shown by line 563); `expr3 ()` was called by the method corresponding to the parent of the $\langle \text{expr3} \rangle$ node on line 560—i.e., by `argumentList ()`, as we see from line 558.
 - C. Each token in the output file is written during execution of a call of `accept (T)` or `nextToken ()` in some non-terminal's parsing method, at a time when the value of `getCurrentToken ()` is *T*; here *T* is the `Symbols` object that represents the token. The parsing method in question is shown by the token's parent in the parse tree. For example, in the Old Exam Question, the comma on line 567 of *X.sol* was written during execution of a call of `accept (COMMA)` or `nextToken ()` in a non-terminal's parsing method; the value of `getCurrentToken ()` was `COMMA` at the time of the call, and we see from line 558 that the parsing method in question was `argumentList ()`.
 - D. The *... node has no more children* line that is a child of a node $\langle N \rangle$ of the tree is written *just before* the corresponding call of method $N ()$ returns control to its caller. The value of `getCurrentToken ()` at that time is shown by the first token in the output file *after* the line *... node has no more children*. For example, in the Old Exam Question, the line *... node has no more children* on line 565 of *X.sol* is a child of the node $\langle \text{expr2} \rangle$ on line 561, and was therefore written just before the corresponding call of `expr2 ()` returned control to its caller. The caller was `expr3 ()`, since the parent of $\langle \text{expr2} \rangle$ is the node $\langle \text{expr3} \rangle$ on line 560. Line 567 of *X.sol* shows that the value of `getCurrentToken ()` was `COMMA` when `expr2 ()` returned control to `expr3 ()`.

The correct answers to the [Old Exam Question](#) are (i)—(d) and (ii)—(a). This follows from 2A, 2B, and 2C above.