Further Hints for TinyJ Assignment 2

The Method `printArgument()` [gap on line 627]

The relevant EBNF rule is

```ebnf
<printArgument> ::= CHARSTRING | <expr3>
```

(a) In the case `<printArgument> ::= <expr3>` the code to be generated is given by

```java
<printArgument>.code = <expr3>.code
WRITEINT
```

Assuming you correctly filled in the gap in the method `printArgument()` in Assignment 1, if you copy just that code into the body of Assignment 2's `printArgument()` then its call of `expr3()` will generate `<expr3>.code`. To complete the `printArgument()` method, you would also need to insert a `new WRITEINTInstr();` statement.

(b) In the case `<printArgument> ::= CHARSTRING` the code to be generated is given by

```java
<printArgument>.code = WRITESTRING a b
```

where `a` and `b` are the data memory addresses of the first and last characters of the `CHARSTRING` string literal that is to be printed. The `WRITESTRING a b` instruction can be generated by `new WRITESTRINGInstr(a,b);` with the appropriate addresses `a` and `b`; but how can your code find the two addresses `a` and `b`?

The solution is provided by the lexical analyzer: When `LexicalAnalyzer.nextToken()` sets `LexicalAnalyzer.currentToken` to `CHARSTRING`, it also sets the private variables `LexicalAnalyzer.startOfString` and `LexicalAnalyzer.endOfString` to the addresses of the memory locations where the first and last characters of the `CHARSTRING` will be placed. `LexicalAnalyzer.getStartOfString()` and `LexicalAnalyzer.getEndOfString()` are public accessor methods that return the two addresses.

The Method `expr1()` [gap on line 723]

The relevant EBNF rule is

```ebnf
<expr1> ::= (' <expr7> ') | (+|-|!) <expr1> | UNSIGNEDINT | null |
           new int '[' <expr3> ']' '{' '[' ']' }
           IDENTIFIER ( . nextInt '(' ')' | [<argumentList>]['[ '<expr3> ']' ] )
```

The `null` and the `IDENTIFIER ( . nextInt '(' ')' | [<argumentList>]['[ '<expr3> ']' ] )` cases have been done for you in `ParserAndTranslator.java`. Here are hints for the other cases:

(a) In the case `<expr1> ::= (' <expr7> ')` the code to be generated is given by

```java
expr1_.code = expr7_.code
```

Similarly, in the case `<expr1> ::= + <expr1>` the code to be generated is given by

```java
expr1_.code = expr1_.code
```

In these two cases, assuming you correctly completed the body of the method `expr1()` when doing Assignment 1, if you use that code as the body of Assignment 2's `expr1()` then in the first case the call of `expr7()` will generate `expr7_.code`, and in the second case the recursive call of `expr1()` will generate `expr1_.code`.

(b) In the case `<expr1> ::= - <expr1>` the code to be generated is given by

```java
expr1_.code = expr1_.code
CHANGESIGN
```

Similarly, in the case `<expr1> ::= ! <expr1>` the code to be generated is given by

```java
expr1_.code = expr1_.code
NOT
```

These two cases are similar to the second case of (a), except that you need to insert a `new CHANGESIGNInstr();` or a `new NOTInstr();` statement.
(c) In the case \( \langle \text{EXPR1} \rangle \ ::= \text{UNSIGNEDINT} \) the code to be generated is given by
\[
\langle \text{EXPR1} \rangle\text{.code} = \text{PUSHNUM } v
\]
where \( v \) is the numerical value of the \text{UNSIGNEDINT} integer literal. The \text{PUSHNUM} \( v \) instruction can be generated by \text{new} \text{PUSHNUMInstr}(v); with the appropriate value \( v \); but how can your code find the value \( v \)?

The solution is provided by the lexical analyzer: When \text{LexicalAnalyzer.nextToken()} sets \text{LexicalAnalyzer.currentToken} to \text{UNSIGNEDINT}, it also sets the private variable \text{LexicalAnalyzer.currentValue} to the numerical value of the \text{UNSIGNEDINT} integer literal. \text{LexicalAnalyzer.getCurrentValue()} is a public accessor method that returns this value.

(d) In the case \( \langle \text{EXPR1} \rangle \ ::= \text{new int }['<\text{EXPR3}>']\{['']\} \) the code to be generated is given by
\[
\langle \text{EXPR1} \rangle\text{.code} = \langle \text{EXPR3} \rangle\text{.code}
\]
Assuming you correctly completed the body of \text{EXPR1()} when doing Assignment 1, if you use that code as the body of Assignment 2's \text{EXPR1()} then \langle \text{EXPR3} \rangle\text{.code} will be generated by a call of \text{EXPR3()}. You would need to insert a \text{new HEAPALLOCInstr();} statement.

**The Method whileStmt() [gap on line 593]**

The relevant EBNF rule is
\[
\langle \text{whileStmt} \rangle \ ::= \text{while }('\langle \text{EXPR7} \rangle ')\langle \text{statement} \rangle
\]
and the code to be generated is given by:
\[
\langle \text{whileStmt} \rangle\text{.code} = a:\langle \text{EXPR7} \rangle\text{.code}
\]
\[
\quad \text{JUMPONFALSE } b
\]
\[
\quad \langle \text{statement} \rangle\text{.code}
\]
\[
\quad \text{JUMP } a
\]
\[
\quad b:
\]
Before you try to complete the method \text{whileStmt()} I recommend you study the method \text{ifStmt()} (line 555), which has already been written for you. In that case the EBNF rule is
\[
\langle \text{ifStmt} \rangle \ ::= \text{if }('\langle \text{EXPR7} \rangle ')\langle \text{statement} \rangle \quad \text{[else }\langle \text{statement} \rangle\text{]}
\]
and the code to be generated is as follows:

**Case 1:** \( \langle \text{ifStmt} \rangle ::= \text{if }('\langle \text{EXPR7} \rangle ')\langle \text{statement}_1 \rangle\)
\[
\langle \text{ifStmt} \rangle\text{.code} = \langle \text{EXPR7} \rangle\text{.code}
\]
\[
\quad \text{JUMPONFALSE } a
\]
\[
\quad \langle \text{statement}_1\rangle\text{.code}
\]
\[
\quad a:\langle \text{statement}_1\rangle\text{.code}
\]

**Case 2:** \( \langle \text{ifStmt} \rangle ::= \text{if }('\langle \text{EXPR7} \rangle ')\langle \text{statement}_1 \rangle \text{else }\langle \text{statement}_2 \rangle\)
\[
\langle \text{ifStmt} \rangle\text{.code} = \langle \text{EXPR7} \rangle\text{.code}
\]
\[
\quad \text{JUMPONFALSE } a
\]
\[
\quad \langle \text{statement}_1\rangle\text{.code}
\]
\[
\quad \text{JUMP } b
\]
\[
\quad a:\langle \text{statement}_2\rangle\text{.code}
\]
\[
\quad b:\langle \text{statement}_2\rangle\text{.code}
\]