The TinyJ Compiler's Static and Stack-Dynamic Memory Allocation Rules

Static Memory Allocation for Static Variables in TinyJ:

The n^{th} static variable in a TinyJ source file is given the data memory location whose address is $n-1$. (It follows that the address of the first static variable is 0.) But this rule ***does not apply to Scanner variables***; in TinyJ, Scanner variables are fictitious variables and no space is allocated to them.

Static Memory Allocation for String Literals in TinyJ:

The k^{th} string literal character in the source file is placed into the data memory location whose address is $m+k$, where m is the last address allocated to a static variable.

Stackframes of Method Calls and How Locations Within Stackframes are Allocated:

Each time a method is called during program execution, a block of contiguous data memory locations known as the call's ***stackframe*** or ***activation record*** is allocated; this block of memory locations will be deallocated when the method returns to its caller. Each formal parameter of the method and each local variable declared in the method's body will be allocated a location within that stackframe—see the allocation rules below. Each location within the stackframe is referred to by its ***offset*** relative to the stackframe location at offset 0. (If in a certain stackframe the data memory address of the location at offset 0 is 73, then the data memory address of the stackframe location at offset +5 is 73 + 5 = 78.)

Memory Allocation Rule for Local Variables Declared in TinyJ Method Bodies:

Whenever the compiler sees a declaration of a local variable (other than a Scanner variable) in the body of a method, that local variable is given the first stackframe location with offset ≥ +1 which has NOT already been allocated to another local variable *that is still in scope*. (So, ignoring Scanner variables, the stackframe offset of the ***first*** local variable in each method's body is +1.) **EXAMPLE**:

```
int func()
{
  int a, b[], c;
  ...
  if ( ... ) {
    int d, e[];
    ...
  }
  else {
    int f, g;
    ...
    int h;
    ...
  }
  ...
  int i;
  ...
}
```

In this example: `a` gets offset 1 `b` gets offset 2 `c` gets offset 3 `d` gets offset 4 `e` gets offset 5
When `f` is declared, `d` and `e` are out of scope. So: `f` gets offset 4 `g` gets offset 5 `h` gets offset 6
When `i` is declared, `f`, `g`, and `h` are out of scope. So: `i` gets offset 4

Memory Allocation Rule for Formal Parameters of TinyJ Methods:

Formal parameters are given locations with ***negative*** offsets; the *last* formal parameter of the method gets the stackframe location at offset –2, the *second-last* parameter gets the location at offset –3, etc. But this ***does not apply to main( )'s parameter***: In TinyJ—unlike Java—main( )'s parameter is ***not*** a real parameter. (**main( )'s stackframe has no negative offsets**.) **EXAMPLE**: In a stackframe of any call of `int g(int p, int q[], int r)`, `r` gets offset –2, `q` gets offset –3, and `p` gets offset –4.

Use of Offsets 0 and –1 [This subsection is relevant mainly to TinyJ Assignment 3.]

The stackframe locations at offsets 0 and –1 store information that is used to support return of control from a called method to its caller. Specifically:

In each stackframe other than main( )'s stackframe, the ***dynamic/control link*** is stored at offset 0. (In main( )'s stackframe, the location at offset 0 stores an implementation-dependent pointer.) In stackframes of methods other than main( ), the dynamic/control link is a pointer to the data memory location at offset 0 in the stackframe of the method's caller.

In each stackframe other than main( )'s stackframe, the ***return address*** is stored at offset –1. The return address is the ***code memory*** address of the next VM instruction to be executed after the current method returns control to its caller. (In main( )'s stackframe there's no location at offset –1.)

Allocation and Deallocation of Stackframes: An Example

Suppose a TinyJ program has methods `main`, `f()`, `g()`, `h()`, and this happens when it is executed:

(1) `main()` is called
(2) `main()` calls `f()`
(3) `f()` calls `g()`
(4) `g()` calls `h()`
(5) `h()` calls `f()`
(6) `f()` returns control to `h()`
(7) `h()` returns control to `g()`
(8) `g()` calls `f()`

Then stackframes are *allocated* in data memory at times (1), (2), (3), (4), (5), and (8); stackframes are *deallocated* at times (6) and (7). Thus there will be just 4 stackframes in data memory immediately after (8). Listed ***in order of increasing memory addresses***, these 4 stackframes will be:

the stackframe of `main()` allocated for call (1)
the stackframe of `f()` allocated for call (2)
the stackframe of `g()` allocated for call (3)
the stackframe of `f()` allocated for call (8)

Note that the stackframes of `h()` and `f()` allocated at times (4) and (5) would no longer exist: The stackframe of `f()` allocated at time (5) would have been deallocated at time (6), and the stackframe of `h()` allocated at time (4) would have been deallocated at time (7).

Comment on Scanner Variables

The data memory allocation rules for TinyJ variables do ***not*** apply to Scanner variables (such as the local variable `userInput` of `howManyRings()` in CS316ex2.java and the static variable `input` in CS316ex5.java). ***No memory at all is allocated for Scanner variables*** in TinyJ. A Scanner variable `x` in TinyJ can only be used in `x.nextInt()`. This is executed by reading an integer from the standard input stream `System.in` (which is usually associated with the keyboard) and returning its value. So the Scanner variable `x` is completely irrelevant. That is why TinyJ essentially ignores Scanner variables and never allocates memory for them. (In contrast, the Scanner variable `x` is ***not*** irrelevant when a ***Java*** program executes `x.nextInt()`: In Java, a Scanner object need not be associated with `System.in`—a Scanner object may, for example, be associated with any input file.) In TinyJ, the Scanner variable `x` in `x.nextInt()` is there only because we want TinyJ to be a subset of Java so that TinyJ programs will be compilable by a Java compiler.

Effects of Executing Each TinyJ Virtual Machine Instruction

"Push" and "pop" refer to the TinyJ VM's **expression evaluation stack** (the **EXPRSTACK**).

n denotes an arbitrary nonnegative integer   *addr* denotes an arbitrary code memory address
a denotes an arbitrary data memory address   *a'* denotes an arbitrary data memory address ≥ *a*
s denotes an arbitrary stackframe offset in the currently executing method activation's stackframe

If an assumption that is made by a VM instruction is ***not*** satisfied when that instruction is executed (e.g., if the item popped by LOADFROMADDR is not a pointer), then the effects of executing the instruction are unspecified.

TinyJ VM Instruction	**Effects of Executing the Instruction**
STOP	Halts the machine.
NOP	Does nothing.
DISCARDVALUE	Pops an item.
PUSHNUM *n*	Pushes the nonnegative integer value *n*.
PUSHSTATADDR *a*	Pushes a pointer to the data memory location whose address is *a*.
PUSHLOCADDR *s*	Pushes a pointer to the data memory location that is at offset *s* in the currently executing method activation's stackframe.
SAVETOADDR	Pops an item *v*. Pops an item *p*, which is assumed to be a pointer to a data memory location. Stores *v* in the memory location to which *p* points.
LOADFROMADDR	Pops an item *p*, which is assumed to be a pointer to a data memory location. Pushes the value that is stored in the memory location to which *p* points.
WRITELNOP	Writes a newline to the screen.
WRITEINT	Pops an item *i*, which is assumed to be an integer. Writes the integer *i* to the screen.
WRITESTRING *a* *a'*	Assumes that the data memory locations whose addresses are ≥ *a* but ≤ *a'* contain the characters of a string literal. Writes that string literal to the screen.
READINT	Assumes the character sequence of an int will be entered on the keyboard. Reads that character sequence and computes the int value it represents. Pushes that integer value.
CHANGESIGN	Pops an item *i*, which is assumed to be an integer. Pushes the value –*i*.
NOT	Pops an item *b*, which is assumed to be a Boolean value. Pushes the Boolean value NOT *b*.
op = ADD, SUB, MUL, DIV, MOD, EQ, LT, GT, NE, GE, or LE	Pops an item *i*, which is assumed to be an integer. Pops an item *j*, which is also assumed to be an integer. Pushes the integer or Boolean value *j* **op** *i*.

op = AND or OR	Pops an item *b*, which is assumed to be a Boolean value. Pops an item *c*, which is also assumed to be a Boolean value. Pushes the Boolean value *c* **op** *b*.
JUMP *addr*	Loads *addr* into the program counter register.
JUMPONFALSE *addr*	Pops an item *b*, which is assumed to be a Boolean value. Loads *addr* into the program counter register if (and only if) *b* is **false**.
PASSPARAM	Allocates one location in the stack-dynamically allocated part of data memory. Pops an item and stores that item in the allocated location; it is expected that the item which is popped and stored will be the value of an actual argument of a method that is about to be called.
CALLSTATMETHOD *addr*	Allocates one location in the stack-dynamically allocated part of data memory; this will be the location at offset –1 in the callee's stackframe. Stores the program counter in the allocated location; the stored address is the call's return address. Loads *addr* into the program counter register.
INITSTKFRM *n*	Allocates one location in the stack-dynamically allocated part of data memory; this will be the location at offset 0 in the current method activation's stackframe. Stores the frame pointer in the allocated location; this will serve as the stackframe's dynamic/control link pointer. Loads a pointer to the allocated location into the frame pointer register. Allocates *n* more locations in the stack-dynamically allocated part of data memory; these will be the locations at offsets 1 through *n* in the current method activation's stackframe.
RETURN *n*	Assumes *n* is the number of parameters of the currently executing method. Assumes the location at offset 0 in the currently executing method activation's stackframe contains the dynamic/control link pointer. Assumes the location at offset –1 in the currently executing method activation's stackframe contains the return address. Loads the dynamic/control link pointer into the frame pointer register. Loads the return address into the program counter register. Deallocates the data memory locations that constitute the currently executing method activation's stackframe.
HEAPALLOC	Pops an item *i*, which is assumed to be a nonnegative integer. Allocates *i*+1 contiguous locations in the heap region of data memory; it is expected that the second through $i+1^{st}$ of those locations will be used to store the elements of an array of *i* elements. Stores in the first of the *i*+1 locations a pointer to the first location above the *i*+1 locations; the second through $i+1^{st}$ locations will all contain 0. Pushes a pointer to the second of the *i*+1 locations.
ADDTOPTR	Pops an item *i*, which is assumed to be a nonnegative integer. Pops an item *p*, which is assumed to be a pointer to the data memory location of the first element of an array *arr*. Pushes *p*+*i* (which is a pointer to the location of the array element *arr*[*i*]), unless *arr* has ≤ *i* elements in which case an error is reported.

Example: The TinyJ Compiler of Assignment 2 should translate the following TinyJ source file into the TinyJ VM instructions shown on the next page.

```
import java.util.Scanner;

class Simple3 {

 static Scanner input = new Scanner(System.in);
 static int x, y = 10;

 public static void main(String args[])
 {
     System.out.print("Enter num: ");
     x = input.nextInt();
     f(17, y, x-y);
     System.out.println(y + f(21,22,23));
 }

 static int f (int a, int b, int c)
 {
     int v[], w;
     int u = x;

     g(c, b + u);
     System.out.print("returning from f ... ");
     return y - a % u;
 }

 static void g (int d, int e)
 {
     int z;

     y = d / e;
 }
}
```

Instructions Generated:

```
0:      PUSHSTATADDR     1
1:      PUSHNUM          10
2:      SAVETOADDR
===================================
3:      INITSTKFRM       0
4:      WRITESTRING      2      12
5:      PUSHSTATADDR     0
6:      READINT
7:      SAVETOADDR
8:      PUSHNUM          17
9:      PASSPARAM
10:     PUSHSTATADDR     1
11:     LOADFROMADDR
12:     PASSPARAM
13:     PUSHSTATADDR     0
14:     LOADFROMADDR
15:     PUSHSTATADDR     1
16:     LOADFROMADDR
17:     SUB
18:     PASSPARAM
19:     CALLSTATMETHOD   34
20:     DISCARDVALUE
21:     PUSHSTATADDR     1
22:     LOADFROMADDR
23:     PUSHNUM          21
24:     PASSPARAM
25:     PUSHNUM          22
26:     PASSPARAM
27:     PUSHNUM          23
28:     PASSPARAM
29:     CALLSTATMETHOD   34
30:     ADD
31:     WRITEINT
32:     WRITELNOP
33:     STOP
===================================
34:     INITSTKFRM       3
35:     PUSHLOCADDR      3
36:     PUSHSTATADDR     0
37:     LOADFROMADDR
38:     SAVETOADDR
39:     PUSHLOCADDR      -2
40:     LOADFROMADDR
41:     PASSPARAM
42:     PUSHLOCADDR      -3
43:     LOADFROMADDR
44:     PUSHLOCADDR      3
45:     LOADFROMADDR
46:     ADD
47:     PASSPARAM
48:     CALLSTATMETHOD   60
49:     NOP
50:     WRITESTRING      13     33
51:     PUSHSTATADDR     1
52:     LOADFROMADDR
53:     PUSHLOCADDR      -4
54:     LOADFROMADDR
55:     PUSHLOCADDR      3
56:     LOADFROMADDR
57:     MOD
58:     SUB
59:     RETURN           3
===================================
60:     INITSTKFRM       1
61:     PUSHSTATADDR     1
62:     PUSHLOCADDR      -3
63:     LOADFROMADDR
64:     PUSHLOCADDR      -2
65:     LOADFROMADDR
66:     DIV
67:     SAVETOADDR
68:     RETURN           2
```

Code Generation Rules Used by the TinyJ Compiler

1. The generated code begins with instructions which initialize each static int and static array reference variable *that has an explicit initializer*. [**Example**: The instructions at addresses 0 – 2 in the code generated for the **`Simple3`** source file.]

2. For variables that do ***not*** have an explicit initializer, no initialization code is generated. Static variables that are ***not*** explicitly initialized will have a value of 0 (in the case of static **`int`** variables) or **`null`** (in the case of static array reference variables) when code execution begins: In the TinyJ VM, the data memory locations allocated to static variables all contain 0 when execution begins, and the **`null`** pointer is represented by 0.

3. Method bodies are translated in the order in which they appear. [**Example**: The code generated for main()'s body appears before the code generated for other methods' bodies.]

4. The code generated for each method (including main) starts with:
   `INITSTKFRM` <total number of stackframe locations needed for local variables declared in that method's body>
   [**Example**: The instructions at addresses 3, 34, and 60.]

5. **`main()`**'s code ends with: **`STOP`** [**Example**: The instruction at address 33.]

6. The code generated for each **`void`** method (other than **`main()`**) ends with: **`RETURN`** *k* Here *k* is the number of formal parameters that the method has. [**Example**: The instruction at address 68.]

7. A **`return`** *expression*`;` statement in a method is translated into:
   <code which leaves the value of *expression* on top of EXPRSTACK>
   `RETURN` *k*
   Again, *k* is the number of formal parameters that the method has. [**Example**: The instructions generated for **`return y–a%u;`** at addresses 51 – 59.]

8. A method call **`f(`**arg_1**`,`** arg_2**`, ...,`** arg_k**`)`** *within an expression* is translated into:
   <code that leaves the value of arg_1 on top of EXPRSTACK>
   `PASSPARAM`
   <code that leaves the value of arg_2 on top of EXPRSTACK>
   `PASSPARAM`
   ...
   <code that leaves the value of arg_k on top of EXPRSTACK>
   `PASSPARAM`
   `CALLSTATMETHOD` <address of the first instruction in method **`f()`**'s code>
   [**Example**: The instructions generated for **`f(21, 22, 23)`** at addresses 23 – 29.]

9. A method call that is a *standalone statement* is translated in the same way as a method call within an expression, except that the **`CALLSTATMETHOD`** may be followed by **`DISCARDVALUE`**, **`NOP`**, or neither:
   (a) If the called method is known to return a value (either because it has already been declared to return a value, or because it has previously been called within an expression) then the **`CALLSTATMETHOD`** must be followed by **`DISCARDVALUE`** to pop the returned value off EXPRSTACK.
   (b) If the called method has already been declared as a **`void`** method, then no **`DISCARDVALUE`** instruction is generated.
   (c) If the called method has not yet been declared, and has not previously been called within an expression, then the compiler cannot tell if the method returns a value or not. In this case, the compiler essentially leaves a one-instruction gap after generating the **`CALLSTATMETHOD`** instruction. Later, when the compiler sees the declaration of the called method, it fills in the gap with either a **`NOP`** or a **`DISCARDVALUE`** instruction, according to whether the called method is declared to be a **`void`** method or a method that returns a value. [**Examples**: The instructions generated for **`f(17, y, x-y)`** at addresses 8 – 20, and the instructions generated for **`g(c, b+u)`** at addresses 39 – 49.]

Hints Relating to the Gaps on Lines 549 and 610 – 4 in ParserAndTranslator.java

As the method `expr2()` (which has been or will be discussed in class) illustrates, a good way to write a method `N()` in Assignment 2's ParserAndTranslator.java that corresponds to a nonterminal `<N>` is to start with the parsing method `N()` in Assignment 1's Parser.java and decide what (if anything) must be added for Assignment 2. Here are two more examples of this.

Example 1: Consider the method `argumentList()` in ParserAndTranslator.java. We see from p. 1 of the handout for TinyJ Assignment 1 that the EBNF rule for `<argumentList>` is

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

Note that there may be any number of `<expr3>`'s (and possibly none at all) between the opening and closing parentheses. Based on this, and the part of Code Generation Rule 8 that relates to the list of arguments, we see that

```
<argumentList>.code =     <expr3>1.code
                          PASSPARAM
                          <expr3>2.code
                          PASSPARAM
                              .
                              .
                              .
                          <expr3>k.code
                          PASSPARAM
```

where *k* is the number of `<expr3>`'s in the `<argumentList>`, and `<expr3>`$_i$ means the i^{th} of those *k* `<expr3>`'s. Assuming you correctly filled in the gap in the method `argumentList()` in Assignment 1, if you copy just that code into the body of Assignment 2's `argumentList()` then its calls of `expr3()` will generate `<expr3>`$_1$.code, `<expr3>`$_2$.code, ... , `<expr3>`$_k$.code. To complete Assignment 2's `argumentList()` method, you would also need to insert one or more statements of the form `new PASSPARAMinstr();` in appropriate places to generate the *k* `PASSPARAM` instructions.

Example 2: Consider the method `outputStmt()` in ParserAndTranslator.java. We see from the EBNF rule for `<outputStmt>` that there are three cases:

1. `<outputStmt>  ::=  System.out.print '(' <printArgument> ')' ;`

   In this case, `<outputStmt>`.code = `<printArgument>`.code
   where `<printArgument>`.code is the code that prints the `<printArgument>` to the screen.

2. `<outputStmt>  ::=  System.out.println '(' ')' ;`

   In this case, `<outputStmt>`.code = `WRITELNOP`

3. `<outputStmt>  ::=  System.out.println '(' <printArgument> ')' ;`

   In this case, `<outputStmt>`.code = `<printArgument>`.code
   `WRITELNOP`

Assuming you correctly filled in the gap in the method `outputStmt()` in Assignment 1, if you copy just that code into the body of Assignment 2's `outputStmt()` then its calls of `printArgument()` will generate `<printArgument>`.code in cases 1 and 3. To complete Assignment 2's `outputStmt()`, you would also need to insert one or more statements of the form `new WRITELNOPinstr();` to generate the `WRITELNOP` instructions in cases 2 and 3.

Hints Relating to the Gaps on Lines 627, 723, and 593 in ParserAndTranslator.java

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

The relevant EBNF rule is `<printArgument> ::= CHARSTRING | <expr3>`

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

```
<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

```
<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

```
<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

```
<expr1>.code = <expr7>.code
```

Similarly, in the case `<expr1> ::= + <expr1>`$_1$ the code to be generated is given by

```
<expr1>.code = <expr1>₁.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>$_1$.code.

(b) In the case `<expr1> ::= - <expr1>`$_1$ the code to be generated is given by

```
<expr1>.code = <expr1>₁.code
               CHANGESIGN
```

Similarly, in the case `<expr1> ::= ! <expr1>`$_1$ the code to be generated is given by

```
<expr1>.code = <expr1>₁.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 `<expr1> ::= UNSIGNEDINT` the code to be generated is given by

`<expr1>`.code = `PUSHNUM v`

where `v` is the numerical value of the `UNSIGNEDINT` integer literal. The `PUSHNUM v` instruction can be generated by `new 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 `LexicalAnalyzer.nextToken()` sets `LexicalAnalyzer.currentToken` to `UNSIGNEDINT`, it also sets the private variable `LexicalAnalyzer.currentValue` to the numerical value of the `UNSIGNEDINT` integer literal. **`LexicalAnalyzer.getCurrentValue()`** is a public accessor method that returns this value.

(d) In the case `<expr1> ::= new int '[' <expr3> ']' { '[' ']' }` the code to be generated is given by

```
<expr1>.code = <expr3>.code
               HEAPALLOC
```

Assuming you correctly completed the body of `expr1()` when doing Assignment 1, if you use that code as the body of Assignment 2's `expr1()` then `<expr3>`.code will be generated by a call of `expr3()`. You would need to insert a `new HEAPALLOCinstr();` statement.

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

The relevant EBNF rule is

`<whileStmt> ::= while '(' <expr7> ')' <statement>`

and the code to be generated is given by:

```
<whileStmt>.code = a: <expr7>.code
                      JUMPONFALSE b
                      <statement>.code
                      JUMP a
                   b:
```

Before you try to complete the method `whileStmt()` I recommend you study the method `ifStmt()` (line 555), which has already been written for you. In that case the EBNF rule is

`<ifStmt> ::= if '(' <expr7> ')' <statement> [else <statement>]`

and the code to be generated is as follows:

Case 1: `<ifStmt> ::= if '(' <expr7> ')' <statement>`$_1$

```
<ifStmt>.code =      <expr7>.code
                     JUMPONFALSE a
                     <statement>1.code
                  a:
```

Case 2: `<ifStmt> ::= if '(' <expr7> ')' <statement>`$_1$ `else <statement>`$_2$

```
<ifStmt>.code =      <expr7>.code
                     JUMPONFALSE a
                     <statement>1.code
                     JUMP b
                  a: <statement>2.code
                  b:
```

Hints Relating to the Gaps on Lines 492, 495, and 511 in ParserAndTranslator.java

Here the relevant EBNF rule is:

```
<assignmentOrInvoc> ::= IDENTIFIER ( { '['<expr3>']' } = <expr3> ; | <argumentList> ; )
```

This rule has two cases:

Case 1: `<assignmentOrInvoc> ::= IDENTIFIER { '['<expr3>']' } = <expr3>;`

Case 2: `<assignmentOrInvoc> ::= IDENTIFIER <argumentList> ;`

<u>The gaps you have to fill in relate only to Case 1</u>. In that case the code to be generated is as follows:

```
<assignmentOrInvoc>.code = PUSHLOCADDR IDENTIFIER.stackframe_offset   or   PUSHSTATADDR IDENTIFIER.address
                           LOADFROMADDR
                           <expr3>index_1.code
                           ADDTOPTR
                               .
                               .
                               .
                           LOADFROMADDR
                           <expr3>index_k.code
                           ADDTOPTR
                           <expr3>right_side.code
                           SAVETOADDR
```

where <expr3>$_{\text{right\_side}}$ means the expression on the right side of `=`, and where the number of occurrences of

```
LOADFROMADDR
<expr3>index_i.code
ADDTOPTR
```

is equal to the *number of indexes* after the IDENTIFIER—i.e., the number of times `'['<expr3>']'` occurs. Often there are no indexes (i.e., the assignment is of the form IDENTIFIER = <expr3>$_{\text{right\_side}}$;). In any case the loop on lines 500–507 generates LOADFROMADDR, <expr3>$_{\text{index\_}i}$.code, and ADDTOPTR as many times as is needed. But you must fill in the ***<u>gap on line 511</u>*** in such a way that <expr3>$_{\text{right\_side}}$.code and SAVETOADDR are generated.

Note that the compiler must generate PUSHLOCADDR IDENTIFIER.stackframe_offset if the IDENTIFIER is a local variable or formal parameter, but it must instead generate PUSHSTATADDR IDENTIFIER.address if the IDENTIFIER is a static variable. To determine whether the IDENTIFIER is a local variable / formal parameter or a static variable, and to determine its stackframe offset in the former case and its data memory address in the latter case, the compiler looks up the IDENTIFIER in the ***<u>symbol table</u>***. The symbol table is a table, maintained by the compiler, which contains:

1. A `LocalVariableRec` object for each parameter and each local variable that is in scope at the point the compiler has reached in the program it is compiling.
2. A `ClassVariableRec` object for each static variable whose declaration has been seen by the compiler.

The compiler records information about each variable or parameter in its `LocalVariableRec` or its `ClassVariableRec` object. For example, it stores the data memory address of each static variable in the `offset` field of that variable's `ClassVariableRec` object. Similarly, the compiler stores the stackframe offset of each parameter or local variable v in the `offset` field of v's `LocalVariableRec` object. (The symbol table also contains a `MethodRec` object for each method that has been declared or called in the part of the program that has been seen by the compiler. But you will ***not*** have to write any code that deals with `MethodRec` objects to complete Assignment 2.)

Line 480 of ParserAndTranslator.java sets `identName` to the name of the IDENTIFIER. On line 485, `t = symTab.searchForVariable(identName);` looks in the symbol table for the IDENTIFIER's `LocalVariableRec` or `ClassVariableRec` object, and sets `t` to refer to that object. Therefore the Boolean value of `t instanceof LocalVariableRec` on line 491 will be **`true`** or **`false`** according to whether the IDENTIFIER is a local variable / formal parameter or a static variable. In the former case `t.offset` will contain IDENTIFIER.stackframe_offset; in the latter case `t.offset` will contain IDENTIFIER.address. Use `t.offset` to fill in the ***<u>gaps on lines 492 and 495</u>*** in such a way that PUSHLOCADDR IDENTIFIER.stackframe_offset is generated if the IDENTIFIER is a local variable or formal parameter, but PUSHSTATADDR IDENTIFIER.address is generated if the IDENTIFIER is a static variable.