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

Static Memory Allocation for Static Variables:
The \( n \)th static int or array reference variable in a TinyJ source file is allocated the data memory location whose address is \( n-1 \). It follows that the address of the first such variable is 0. [No space is allocated to a Scanner variable; in TinyJ, Scanner variables are fictitious variables.]

Static Memory Allocation for String Literals:
The \( k \)th string literal character is placed into the data memory location whose address is \( m+k \), where \( m \) is the address of the last static int or array reference variable.

Allocation of Stackframe Locations for Formal Parameters and the Return Address of a Method:
For all methods other than \( \text{main}() \), the last formal parameter of the method is given a stackframe offset of \(-2\), the second-last parameter is given an offset of \(-3\), and so on. The return address is given an offset of \(-1\) in all stackframes other than \( \text{main}() \)'s stackframe. Thus if a method other than \( \text{main}() \) has \( m \) formal parameters, then its \( k \)th formal parameter is given a stackframe offset of \( k-m-2 \) and its stackframe has a total of \( m+1 \) negative offsets. The stackframe of \( \text{main}() \) has no negative offsets.

Allocation of Stackframe Locations for the Dynamic/Control Link:
The dynamic/control link is given an offset of 0 in each stackframe other than \( \text{main}() \)'s stackframe. In \( \text{main}() \)'s stackframe, the location at offset 0 stores an implementation-dependent pointer.

Allocation of Stackframe Locations for Local Variables Declared in Method Bodies:
Whenever the compiler sees a declaration of a local variable that is not a Scanner variable in the body of any method (including \( \text{main}() \)), that local variable is given the first stackframe offset \( \geq +1 \) which has NOT already been allocated to another local variable that is still in scope. (So, if we ignore Scanner variables, the stackframe offset of the first local variable in each method's body is +1.) The following example illustrates the allocation of stackframe offsets to local variables:

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

If no other local variables are declared in this method, then the stackframe offsets of the local variables are as follows:

- \( a \) is given offset 1
- \( b \) is given offset 2
- \( c \) is given offset 3
- \( d \) is given offset 4
- \( e \) is given offset 5
- \( f \) is given offset 4
- \( g \) is given offset 5
- \( h \) is given offset 6
- \( i \) is given offset 4

Note that, at the point where \( f \) is declared, \( d \) and \( e \) are no longer in scope.

Note that, at the point where \( i \) is declared, \( f, g, \) and \( h \) are no longer in scope.
An Example of Stack-Dynamic Allocation

Suppose that a certain TinyJ program has methods main, f, g, and h, and that the following happens when the program 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 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 the method 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 keyboard and returning its value, and 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(): Since Java programs, unlike TinyJ programs, can take input from either a file or the keyboard, when a Java program executes x.nextInt() it cannot assume—as a TinyJ program would—that an integer is to be read from the keyboard, but must look at the object x to determine where to read the integer from.) In TinyJ, the Scanner variable x in x.nextInt() is there only because we want TinyJ to be a subset of Java (i.e., we want TinyJ programs to be compilable by a Java compiler); it serves no other purpose.
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
- $a$ denotes an arbitrary data memory address
- $a'$ denotes an arbitrary data memory address
- $s$ 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.

<table>
<thead>
<tr>
<th>TinyJ VM Instruction</th>
<th>Effects of Executing the Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>Halts the machine.</td>
</tr>
<tr>
<td>NOP</td>
<td>Does nothing.</td>
</tr>
<tr>
<td>DISCARDVALUE</td>
<td>Pops an item.</td>
</tr>
<tr>
<td>PUSHNUM $n$</td>
<td>Pushes the nonnegative integer value $n$.</td>
</tr>
<tr>
<td>PUSHSTATADDR $a$</td>
<td>Pushes a pointer to the data memory location whose address is $a$.</td>
</tr>
<tr>
<td>PUSHLOCADDR $s$</td>
<td>Pushes a pointer to the data memory location that is at offset $s$ in the currently executing method activation's stackframe.</td>
</tr>
<tr>
<td>SAVETOADDR</td>
<td>Pops an item $v$. Pops an item $p$, which is assumed to be a pointer to a data memory location.</td>
</tr>
<tr>
<td></td>
<td>Stores $v$ in the memory location to which $p$ points.</td>
</tr>
<tr>
<td>LOADFROMADDR</td>
<td>Pops an item $p$, which is assumed to be a pointer to a data memory location.</td>
</tr>
<tr>
<td></td>
<td>Pushes the value that is stored in the memory location to which $p$ points.</td>
</tr>
<tr>
<td>WRITELNOP</td>
<td>Writes a newline to the screen.</td>
</tr>
<tr>
<td>WRITEINT</td>
<td>Pops an item $i$, which is assumed to be an integer.</td>
</tr>
<tr>
<td></td>
<td>Writes the integer $i$ to the screen.</td>
</tr>
<tr>
<td>WRITESTRING $a$ $a'$</td>
<td>Assumes that the data memory locations whose addresses are $\geq a$ but $\leq a'$ contain the characters of a string literal.</td>
</tr>
<tr>
<td></td>
<td>Writes that string literal to the screen.</td>
</tr>
<tr>
<td>READINT</td>
<td>Assumes that the digit sequence of an int will be entered on the keyboard.</td>
</tr>
<tr>
<td></td>
<td>Reads that digit sequence and computes the value of the int it represents.</td>
</tr>
<tr>
<td></td>
<td>Pushes that integer value.</td>
</tr>
<tr>
<td>CHANGESIGN</td>
<td>Pops an item $i$, which is assumed to be an integer.</td>
</tr>
<tr>
<td></td>
<td>Pushes the value $-i$.</td>
</tr>
<tr>
<td>NOT</td>
<td>Pops an item $b$, which is assumed to be a Boolean value.</td>
</tr>
<tr>
<td></td>
<td>Pushes the Boolean value NOT $b$.</td>
</tr>
<tr>
<td>$\text{op} = \text{ADD, SUB, MUL, DIV, MOD, EQ, LT, GT, NE, GE, or LE}$</td>
<td>Pops an item $i$, which is assumed to be an integer.</td>
</tr>
<tr>
<td></td>
<td>Pops an item $j$, which is also assumed to be an integer.</td>
</tr>
<tr>
<td></td>
<td>Pushes the integer or Boolean value $j \text{op} i$.</td>
</tr>
</tbody>
</table>
\textbf{op} = \text{AND or OR} \quad \text{Pops an item } b, \text{ which is assumed to be a Boolean value.}
\text{Pops an item } c, \text{ which is also assumed to be a Boolean value.}
\text{Pushes the Boolean value } c \text{ op } b.

\textbf{JUMP addr} \quad \text{Loads } addr \text{ into the program counter register.}

\textbf{JUMPONFALSE addr} \quad \text{Pops an item } b, \text{ which is assumed to be a Boolean value.}
\text{Loads } addr \text{ into the program counter register if (and only if) } b \text{ is } \text{false.}

\textbf{PASSPARAM} \quad \text{Allocates one location in the stack-dynamically allocated part of data memory.}
\text{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.}

\textbf{CALLSTATMETHOD addr} \quad \text{Allocates one location in the stack-dynamically allocated part of data memory; this will be the location at offset } –1 \text{ in the callee's stackframe.}
\text{Stores the program counter in the allocated location; the stored address is the call's return address.}
\text{Loads } addr \text{ into the program counter register.}

\textbf{INITSTKFRM } n \quad \text{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.}
\text{Stores the frame pointer in the allocated location; this will serve as the stackframe's dynamic/control link pointer.}
\text{Loads a pointer to the allocated location into the frame pointer register.}
\text{Allocates } n \text{ more locations in the stack-dynamically allocated part of data memory; these will be the locations at offsets 1 through } n \text{ in the current method activation's stackframe.}

\textbf{RETURN } n \quad \text{Assumes } n \text{ is the number of parameters of the currently executing method.}
\text{Assumes the location at offset 0 in the currently executing method activation's stackframe contains the dynamic/control link pointer.}
\text{Assumes the location at offset } –1 \text{ in the currently executing method activation's stackframe contains the return address.}
\text{Loads the dynamic/control link pointer into the frame pointer register.}
\text{Loads the return address into the program counter register.}
\text{Deallocates the data memory locations that constitute the currently executing method activation's stackframe.}

\textbf{HEAPALLOC} \quad \text{Pops an item } i, \text{ which is assumed to be a nonnegative integer.}
\text{Allocates } i+1 \text{ contiguous locations in the heap region of data memory; it is expected that the second through } i+1^{\text{st}} \text{ of those locations will be used to store the elements of an array of } i \text{ elements.}
\text{Stores in the first of the } i+1 \text{ locations a pointer to the first location above the } i+1 \text{ locations; the second through } i+1^{\text{st}} \text{ locations will all contain 0.}
\text{Pushes a pointer to the second of the } i+1 \text{ locations.}

\textbf{ADDTOPTR} \quad \text{Pops an item } i, \text{ which is assumed to be a nonnegative integer.}
\text{Pops an item } p, \text{ which is assumed to be a pointer to the data memory location of the first element of an array } arr.
\text{Pushes } p+i \text{ (which is a pointer to the location of the array element } arr[i]), \text{ unless } arr \text{ has } \leq i \text{ elements in which case an error is reported.}
Hints for TinyJ Assignment 2

As the method `expr2()` (which was 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 that relate to the gaps on lines 549 and 610 – 4 in ParserAndTranslator.java.

**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.
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 34: INITSTKFRM 3
1: PUSHNUM 10 35: PUSHLOCADDR 3
2: SAVETOADDR 36: PUSHSTATADDR 0

===================================
3: INITSTKFRM 0 38: SAVETOADDR
4: WRITESTRING 2 12 39: PUSHLOCADDR -2
5: PUSHSTATADDR 0 40: LOADFROMADDR
6: READINT 41: PASSPARAM
7: SAVETOADDR 42: PUSHLOCADDR -3
8: PUSHNUM 17 43: LOADFROMADDR
9: PASSPARAM 44: PUSHLOCADDR 3
10: PUSHSTATADDR 1 45: LOADFROMADDR
11: LOADFROMADDR 46: ADD
12: PASSPARAM 47: PASSPARAM
13: PUSHSTATADDR 0 48: CALLSTATMETHOD 60
14: LOADFROMADDR 49: NOP
15: PUSHSTATADDR 1 50: WRITESTRING 13 33
16: LOADFROMADDR 51: PUSHSTATADDR 1
17: SUB 52: LOADFROMADDR
18: PASSPARAM 53: PUSHLOCADDR -4
19: CALLSTATMETHOD 34 54: LOADFROMADDR
20: DISCARDVALUE 55: PUSHLOCADDR 3
21: PUSHSTATADDR 1 56: LOADFROMADDR
22: LOADFROMADDR 57: MOD
23: PUSHNUM 21 58: SUB
24: PASSPARAM 59: RETURN 3
25: PUSHNUM 22

===================================
26: PASSPARAM 60: INITSTKFRM 1
27: PUSHNUM 23 61: PUSHSTATADDR 1
28: PASSPARAM 62: PUSHLOCADDR -3
29: CALLSTATMETHOD 34 63: LOADFROMADDR
30: ADD 64: PUSHLOCADDR -2
31: WRITEINT 65: LOADFROMADDR
32: WRITELNOP 66: DIV
33: STOP 67: SAVETOADDR

=====================================
68: RETURN 2
Code Generation Rules Used by the TinyJ Compiler

1. The null pointer is represented by 0.

2. The generated code begins with instructions which initialize each static int and static array reference variable that has an explicit initializer. (The TinyJ VM's static memory addresses all contain 0 when execution begins. So static variables that are not initialized will have an initial value of 0 or null.) [See the instructions at addresses 0–2 in the code generated for the Simple3 source file.]

3. Method bodies are translated in the order in which they appear. (E.g., 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:
   \texttt{INITSTKFRM} \quad \text{<total number of stackframe locations needed for local variables declared in that method's body>} \\
   [See the instructions at addresses 3, 34, and 60.]

5. main()'s code ends with: \texttt{STOP}  \\
   [See the instruction at address 33.]

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

7. A \texttt{return expression;} statement in a method is translated into:
   \texttt{<code which leaves the value of expression on top of EXPRSTACK>}  \\
   \texttt{RETURN \ k}  \\
   Again, \( k \) is the number of formal parameters that the method has.  \\
   [See the instructions generated for \texttt{return y-a+u;} at addresses 51–59.]

8. A method call \( f(arg_1, arg_2, \ldots, arg_k) \) within an expression is translated into:
   \texttt{<code that leaves the value of arg_1 on top of EXPRSTACK>}  \\
   \texttt{PASSPARAM}  \\
   \texttt{<code that leaves the value of arg_2 on top of EXPRSTACK>}  \\
   \texttt{PASSPARAM}  \\
   \ldots  \\
   \texttt{<code that leaves the value of arg_k on top of EXPRSTACK>}  \\
   \texttt{PASSPARAM}  \\
   \texttt{CALLSTATMETHOD \ <address of the first instruction in method f()'s code>}  \\
   [See the instructions generated for \texttt{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 \texttt{CALLSTATMETHOD} may be followed by \texttt{DISCARDVALUE}, \texttt{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 \texttt{CALLSTATMETHOD} must be followed by \texttt{DISCARDVALUE} to pop the returned value off EXPRSTACK.
   (b) If the called method has already been declared as a \texttt{void} method, then no \texttt{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 \texttt{CALLSTATMETHOD} instruction. Later, when the compiler sees the declaration of the called method, it fills in the gap with either a \texttt{NOP} or a \texttt{DISCARDVALUE} instruction, according to whether the called method is declared to be a \texttt{void} method or a method that returns a value.  \\
   [See the instructions generated for \texttt{f(17, y, x-y)} at addresses 8–20, and the instructions generated for \texttt{g(c, b+u)} at addresses 39–49.]