Stack Applications

Three applications of stacks are presented here. These examples are central to many activities that a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

Expression evaluation

In particular we will consider arithmetic expressions. Understand that there are boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.

This study of arithmetic expression evaluation is an example of problem solving where you solve a simpler problem and then **transform** the actual problem to the simpler one.

Aside: *The NP-Complete problem*. There are a set of apparently intractable problems: finding the shortest route in a graph (Traveling Salesman Problem), bin packing, linear programming, etc. that are similar enough that if a polynomial solution is ever found (exponential solutions abound) for one of these problems, then the solution can be applied to all problems.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +

(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
x * x + y / (z -1)		
40 - 3 * 5 + 1		

Postfix expressions are easily evaluated with the aid of a stack.

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	? × (4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

Postfix Evaluation Algorithm

Assume we have a string of operands and operators, an informal, by hand process is

1. Scan the expression left to right
2. Skip values or variables (operands)
3. When an operator is found, apply the operation to the preceding two operands
4. Replace the two operands and operator with the calculated value (three symbols are replaced with one operand)
5. Continue scanning until only a value remains--the result of the expression

The time complexity is O(n) because each operand is scanned once, and each operation is performed once.

A more formal algorithm:

```
create a new stack
while(input stream is not empty){
   token = getNextToken();
   if(token instanceof operand){
       push(token);
   } else if (token instance of operator) {
       op2 = pop();
       op1 = pop();
       result = calc(token, op1, op2);
       push(result);
   }
}
return pop();
```

Demonstration with 2 3 4 + * 5 -

and 5 7 + 6 2 - *

Infix transformation to Postfix

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.
5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the output string.

This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lack of parenthesis could point to such error determination.

Apply the algorithm to the above expressions.

Backtracking

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative

Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.

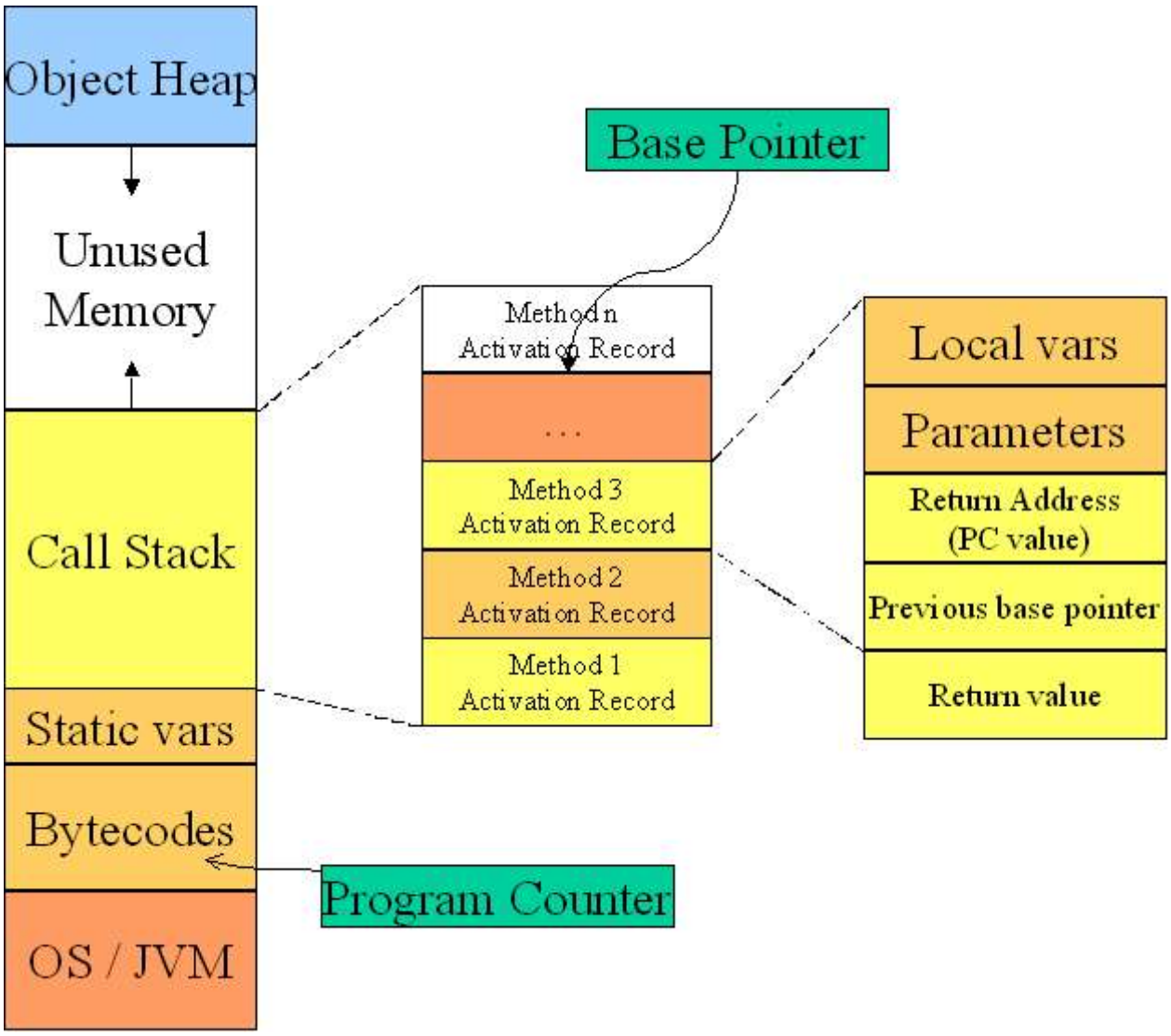
Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

Memory Management

Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.

The discussion of JVM in the text is consistent with older and modern OS such as Windows NT 8 or 10, Solaris, Unix/Linux runtime environments.

Each program that is running in a computer system has its own memory allocation containing the typical layout as shown below. [Thread management has a variant of this.]



Call and return process

When a method/function is called

1. An **Activation Record i**s created; its size depends on the number and size of the local variables and parameters.
2. The **Base Pointer** "Register" value is saved in the special location reserved for it
3. The **Program Counter** "Register" value is saved in the **Return Address** location
4. The Base Pointer is now reset to the new base (top of the call stack prior to the creation of the AR)
5. The Program Counter is set to the location of the first bytecode of the method being called
6. Copies the calling parameters into the Parameter region
7. Initializes local variables in the local variable region

While the method executes, the local variables and parameters are simply found by adding a constant associated with each variable/parameter to the Base Pointer.

When a method returns

1. Get the program counter from the activation record and replace what's in the PC register
2. Get the base pointer value from the AR and replace what's in the BP register
3. Pop the Activation Record entirely from the stack.

There is also a **Stack Pointer** that allow other temporary use of the stack above the top AR.