

ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of February 2, 2013

Abstract

- Abstract data type: Stack
 - Stack-based algorithms

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Evaluating arithmetic expressions

Stack-based algorithms are used for **syntactical analysis** (*parsing*).

For example to evaluate the following expression:

$$1 + 2 * 3 - 4$$

Compilers use similar algorithms to check the syntax of your programs and generate machine instructions (executable).

To verify that parentheses are balanced: '([])' is ok, but not '([)]' or ')((())('.

The first two steps of the analysis of a source program by a compiler are the **lexical analysis** and the **syntactical analysis**.

During the **lexical analysis** (*scanning*) the source code is read from left to right and the characters are regrouped into **tokens**, which are successive characters that constitute numbers or identifiers. One of the tasks of the lexical analyser is to remove spaces from the input.

E.g.:

·10 · + · ·2 + · · ·300

where “·” represent blank spaces, is transformed into the following list of tokens:

[10,+ ,2,+ ,300]

The next step is the syntactical analysis (*parsing*) and consists in regrouping the tokens into grammatical units, for example the sub-expressions of RPN expressions (seen in class this week).

In the next slides, we look at simple examples of lexical and syntactical analysis.

```
public class Test {

    public static void scan( String expression ) {

        Reader reader = new Reader( expression );

        while ( reader.hasMoreTokens() ) {
            System.out.println( reader.nextToken() );
        }
    }

    public static void main( String[] args ) {
        scan( " 3      + 4 * 567  " );
    }
}

// > java Test
// INTEGER: 3
// SYMBOL: +
// INTEGER: 4
// SYMBOL: *
// INTEGER: 567
```

```
public class Token {
    private static final int INTEGER = 1;
    private static final int SYMBOL = 2;
    private int iValue;
    private String sValue;
    private int type;

    public Token( int iValue ) {
        this.iValue = iValue;
        type = INTEGER;
    }
    public Token( String sValue ) {
        this.sValue = sValue;
        type = SYMBOL;
    }

    public int iValue() { ... }
    public String sValue() { ... }

    public boolean isInteger() { return type == INTEGER; }
    public boolean isSymbol() { return type == SYMBOL; }
}
```

LR Scan

```
public static int execute( String expression ) {
    Token op = null; int l = 0, r = 0;

    Reader reader = new Reader( expression );
    l = reader.nextToken().iValue();

    while ( reader.hasMoreTokens() ) {
        op = reader.nextToken();
        r = reader.nextToken().iValue();
        l = eval( op, l, r );
    }
    return l;
}
```

eval(Token op, int l, int r)

```
public static int eval( Token op, int l, int r ) {  
  
    int result = 0;  
  
    if ( op.sValue().equals( "+" ) )  
        result = l + r;  
    else if ( op.sValue().equals( "-" ) )  
        result = l - r;  
    else if ( op.sValue().equals( "*" ) )  
        result = l * r;  
    else if ( op.sValue().equals( "/" ) )  
        result = l / r;  
    else  
        System.err.println( "not a valid symbol" );  
  
    return result;  
}
```


Evaluating an arithmetic expression: LR Scan

Left-to-right algorithm:

Declare L, R and OP

Read L

While not end-of-expression

do:

 Read OP

 Read R

 Evaluate L OP R

 Store result in L

At the end of the loop the result can be found in L.

$$3 * 8 - 10$$

3 + 4 - 5
^

L = 3

OP =

R =

> Read L

While not end-of-expression

do:

 Read OP

 Read R

 Evaluate L OP R

 Store result in L

3 + 4 - 5
^

L = 3

OP = +

R =

Read L

While not end-of-expression

do:

- > Read OP
- Read R
- Evaluate L OP R
- Store result in L

3 + 4 - 5
 ^

L = 3

OP = +

R = 4

Read L

While not end-of-expression

do:

 Read OP

> Read R

 Evaluate L OP R

 Store result in L

3 + 4 - 5
 ^

L = 3

OP = +

R = 4

Read L

While not end-of-expression

do:

 Read OP

 Read R

> Evaluate L OP R (7)

 Store result in L

3 + 4 - 5
 ^

L = 7

OP = +

R = 4

Read L

While not end-of-expression

do:

 Read OP

 Read R

 Evaluate L OP R

> Store result in L

$$3 + 4 - 5$$

^

$$L = 7$$

$$OP = -$$

$$R = 4$$

Read L

While not end-of-expression

do:

- > Read OP
- Read R
- Evaluate L OP R
- Store result in L

3 + 4 - 5
 ^

L = 7

OP = -

R = 5

Read L

While not end-of-expression

do:

 Read OP

> Read R

 Evaluate L OP R

 Store result in L

3 + 4 - 5
 ^

L = 7

OP = -

R = 5

Read L

While not end-of-expression

do:

 Read OP

 Read R

> Evaluate L OP R (2)

 Store result in L

3 + 4 - 5
 ^

L = 2

OP = -

R = 5

Read L

While not end-of-expression

do:

 Read OP

 Read R

 Evaluate L OP R

> Store result in L

3 + 4 - 5 ^

L = 2

OP = -

R = 5

Read L

While not end-of-expression

do:

 Read OP

 Read R

 Evaluate L OP R

 Store result in L

>

⇒ end of expression, exit the loop, L contains the result.

What do you think?

Without **parentheses** the following expression cannot be evaluated correctly:

$$\Rightarrow 7 - (3 - 2)$$

Because the result of the left-to-right analysis corresponds to:

$$\Rightarrow (7 - 3) - 2$$

Similarly the following expression cannot be evaluated by our simple algorithm:

$$\Rightarrow 7 - 3 * 2$$

Since the left-to-right analysis corresponds to:

$$\Rightarrow (7 - 3) * 2$$

But according to the **operator precedences**, the evaluation should have proceeded as follows:

$$\Rightarrow 7 - (3 * 2)$$

Remarks

The left-to-right algorithm:

- Does not handle parentheses;
- Nor precedence.

Solutions:

1. Use a different notation;
2. Develop more complex algorithms.

⇒ Both solutions involve stacks!

Notations

There are 3 ways to represent the following expression: **L OP R**.

infix: this is the usual notation, the operator is sandwiched in between its operands: L OP R;

postfix: in postfix notation, the operands are placed before the operator, L R OP. This notation is also called *Reverse Polish Notation* or **RPN**, it's the notation used by certain scientific calculators (such as the HP-35 from Hewlett-Packard or the Texas Instruments TI-89 using the RPN Interface by Lars Frederiksen¹) or PostScript programming language.

$$\begin{array}{l} 7 - (3 - 2) = 7 3 2 - - \\ (7 - 3) - 2 = 7 3 - 2 - \end{array}$$

prefix: the third notation consists in placing the operator before the operands, OP L R. The programming language Lisp uses a combination of parentheses and prefix notation: (- 7 (* 3 2)).

¹www.calculator.org/rpn.html

Infix \rightarrow postfix (mentally)

Successively transform, one by one, all the sub-expressions following the same order of evaluation that you would normally follow to evaluate an infix expression.

An infix expression $l \diamond r$ becomes $l r \diamond$, where l and r are sub-expressions and \diamond is an operator.

$$9 / (2 \times 4 - 5)$$

$$9 / (\underbrace{2}_l \underbrace{\times}_\diamond \underbrace{4}_r - 5)$$

$$9 / (\underbrace{2}_l \underbrace{4}_r \underbrace{\times}_\diamond - 5)$$

$$9 / ([2 4 \times] - 5)$$

$$9 / (\underbrace{[2 4 \times]}_l \underbrace{-}_\diamond \underbrace{5}_r)$$

$$9 / [\underbrace{[2 4 \times]}_l \underbrace{5}_r \underbrace{-}_\diamond]$$

$$9 / [2 4 \times 5 -]$$

$$\underbrace{9}_l \underbrace{/}_\diamond \underbrace{[2 4 \times 5 -]}_r$$

$$\underbrace{9}_l \underbrace{[2 4 \times 5 -]}_r \underbrace{/}_\diamond$$

$$9 2 4 \times 5 - /$$

Evaluating a postfix expression (mentally)

Scan the expression from left to right. When the current element is an operator, apply the operator to its operands, i.e. replace $l r \diamond$ by the result of the evaluation of $l \diamond r$.

$$9 \ 2 \ 4 \ \times \ 5 \ - \ /$$

$$9 \ \underbrace{2}_{l} \ \underbrace{4}_{r} \ \underbrace{\times}_{\diamond} \ 5 \ - \ /$$

$$9 \ \underbrace{l \diamond r}_{8} \ 5 \ - \ /$$

$$9 \ \underbrace{8}_{l} \ \underbrace{5}_{r} \ \underbrace{-}_{\diamond} \ /$$

$$9 \ \underbrace{l \diamond r}_{3} \ /$$

$$\underbrace{9}_{l} \ \underbrace{3}_{r} \ \underbrace{/}_{\diamond}$$

$$\underbrace{l \diamond r}_{3}$$

Evaluating a postfix expression

Until the end of the expression has been reached:

1. From left to right until the first operator;
2. Apply the operator to the two preceding operands;
3. Replace the operator and its operands by the result.

At the end we have result.

9 3 / 10 2 3 * - +

9 2 4 * 5 - /

Remarks: infix vs postfix

The order of the operands is the same for both notations, however operators are inserted at different places:

$$2 + (3 * 4) = 2 \ 3 \ 4 \ * \ +$$

$$(2 + 3) * 4 = 2 \ 3 \ + \ 4 \ *$$

Evaluating an infix expression involves handling operators precedence and parenthesis — in the case of the postfix notation, those two concepts are embedded in the expression, i.e. the order of the operands and operators.

Algorithm: Eval Infix

What role will the stack be playing?

```
operands = new stack;
```

```
while ( "has more tokens" ) {  
    t = next token;  
    if ( "t is an operand" ) {  
        operands.push( "the integer value of t" );  
    } else { // this is an operator  
        op = "operator value of t";  
        r = operands.pop();  
        l = operands.pop();  
        operands.push( "eval( l, op, r )" );  
    }  
}  
return operands.pop();
```

Evaluating a postfix expression

The algorithm requires a stack (Numbers), a variable that contains the last element that was read (X) and two more variables, L and R, whose purpose is the same as before.

```
Numbers = [
```

```
While not end-of-expression
```

```
do:
```

```
  Read X
```

```
  If X isNumber, PUSH X onto Numbers
```

```
  If X isOperator,
```

```
    R = POP Numbers (right before left?!) 
```

```
    L = POP Numbers
```

```
    Evaluate L X R; PUSH result onto Numbers
```

```
To obtain the final result: POP Numbers.
```


$$9 \quad 3 \quad - \quad 2 \quad /$$

9 3 / 10 2 3 * - +

$$9 / ((2 * 4) - 5) = \underset{\wedge}{9} \ 2 \ 4 \ * \ 5 \ - \ /$$

```
> Numbers = [  
  X =  
  L =  
  R =
```

```
While not end-of-expression  
do:
```

```
  Read X
```

```
  If X isNumber, PUSH X onto Numbers
```

```
  If X isOperator,
```

```
    R = POP Numbers
```

```
    L = POP Numbers
```

```
    Evaluate L X R; PUSH result onto Numbers
```

⇒ Create a new stack

$$9 / ((2 * 4) - 5) = \underset{\wedge}{9} \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [

X = 9

L =

R =

While not end-of-expression

do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = \underset{\wedge}{9} \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9

X = 9

L =

R =

While not end-of-expression

do:

 Read X

> If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Push X

$$9 / ((2 * 4) - 5) = 9 \quad 2 \quad 4 \quad * \quad 5 \quad - \quad /$$

^

Numbers = [9

X = 2

L =

R =

While not end-of-expression

do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 2

X = 2

L =

R =

While not end-of-expression

do:

Read X

> If X isNumber, PUSH X onto Numbers

If X isOperator,

R = POP Numbers

L = POP Numbers

Evaluate L X R; PUSH result onto Numbers

⇒ Push X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 2

X = 4

L =

R =

While not end-of-expression

do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 2 4

X = 4

L =

R =

While not end-of-expression

do:

 Read X

> If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Push X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 2 4

X = *

L =

R =

While not end-of-expression

do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 2

X = *

L =

R = 4

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

> R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Ah! X is an operator, pop the top element save into R

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

^

Numbers = [9

X = *

L = 2

R = 4

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

> L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Top element is removed and saved into L

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 8

X = 5

L = 2

R = 4

While not end-of-expression

do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 8 5

X = 5

L = 2

R = 4

While not end-of-expression

do:

Read X

> If X isNumber, PUSH X onto Numbers

If X isOperator,

R = POP Numbers

L = POP Numbers

Evaluate L X R; PUSH result onto Numbers

⇒ Push X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 8 5

X = -

L = 2

R = 4

While not end-of-expression
do:

- > Read X
- If X isNumber, PUSH X onto Numbers
- If X isOperator,
 - R = POP Numbers
 - L = POP Numbers
 - Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 8

X = -

L = 2

R = 5

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

> R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Remove the top element and save it into R

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

^

Numbers = [9

X = -

L = 8

R = 5

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

> L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ Remove the top element and save it into L

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9 3

X = -

L = 8

R = 5

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

> Evaluate L X R; PUSH result onto Numbers

⇒ Push the result of L X R, $8 - 5 = 3$, onto the stack

9 / ((2 * 4) - 5) = 9 2 4 * 5 - /
^

Numbers = [9 3

X = /

L = 8

R = 5

While not end-of-expression

do:

> Read X

If X isNumber, PUSH X onto Numbers

If X isOperator,

R = POP Numbers

L = POP Numbers

Evaluate L X R; PUSH result onto Numbers

⇒ Read X

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [9

X = /

L = 8

R = 3

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

> R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ R = POP Numbers.

$$9 / ((2 * 4) - 5) = 9 \ 2 \ 4 \ * \ 5 \ - \ /$$

Numbers = [

X = /

L = 9

R = 3

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

> L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

⇒ L = POP Numbers.

9 / ((2 * 4) - 5) = 9 2 4 * 5 - /
^

Numbers = [3

X = /

L = 9

R = 3

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

> Evaluate L X R; PUSH result onto Numbers

⇒ Push L X R, $9 \div 3 = 3$, onto the stack sur la pile.

9 / ((2 * 4) - 5) = 9 2 4 * 5 - / ^

Numbers = [3

X = /

L = 9

R = 3

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

> Evaluate L X R; PUSH result onto Numbers

⇒ End-of-expression

9 / ((2 * 4) - 5) = 9 2 4 * 5 - / ^

Numbers = [

X = /

L = 9

R = 3

While not end-of-expression

do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

 L = POP Numbers

 Evaluate L X R; PUSH result onto Numbers

>

⇒ The result is “POP Numbers = 3”; the stack is now empty

Problem

Rather than evaluating an RPN expression, we would like to convert an RPN expression to infix (usual notation).

Hum?

Do we need a new algorithm?

No, a simple modification will do, replace “Evaluate L OP R” by “Concatenate (L OP R)”.

Note: parentheses are essential (not all of them but some are).

This time the stack does not contain numbers but character strings that represent sub-expressions.

9 5 6 3 / - /

Postfix → infix

```
String rpnToInfix(String[] tokens)
```

```
Numbers = [
```

```
X =
```

```
L =
```

```
R =
```

```
While not end-of-expression
```

```
do:
```

```
  Read X
```

```
  If X isNumber, PUSH X onto Numbers
```

```
  If X isOperator,
```

```
    R = POP Numbers
```

```
    L = POP Numbers
```

```
    Concatenate ( L X R ); PUSH result onto Numbers
```

Postfix → ?

While not end-of-expression
do:

 Read X

 If X isNumber, PUSH X onto Numbers

 If X isOperator,

 R = POP Numbers

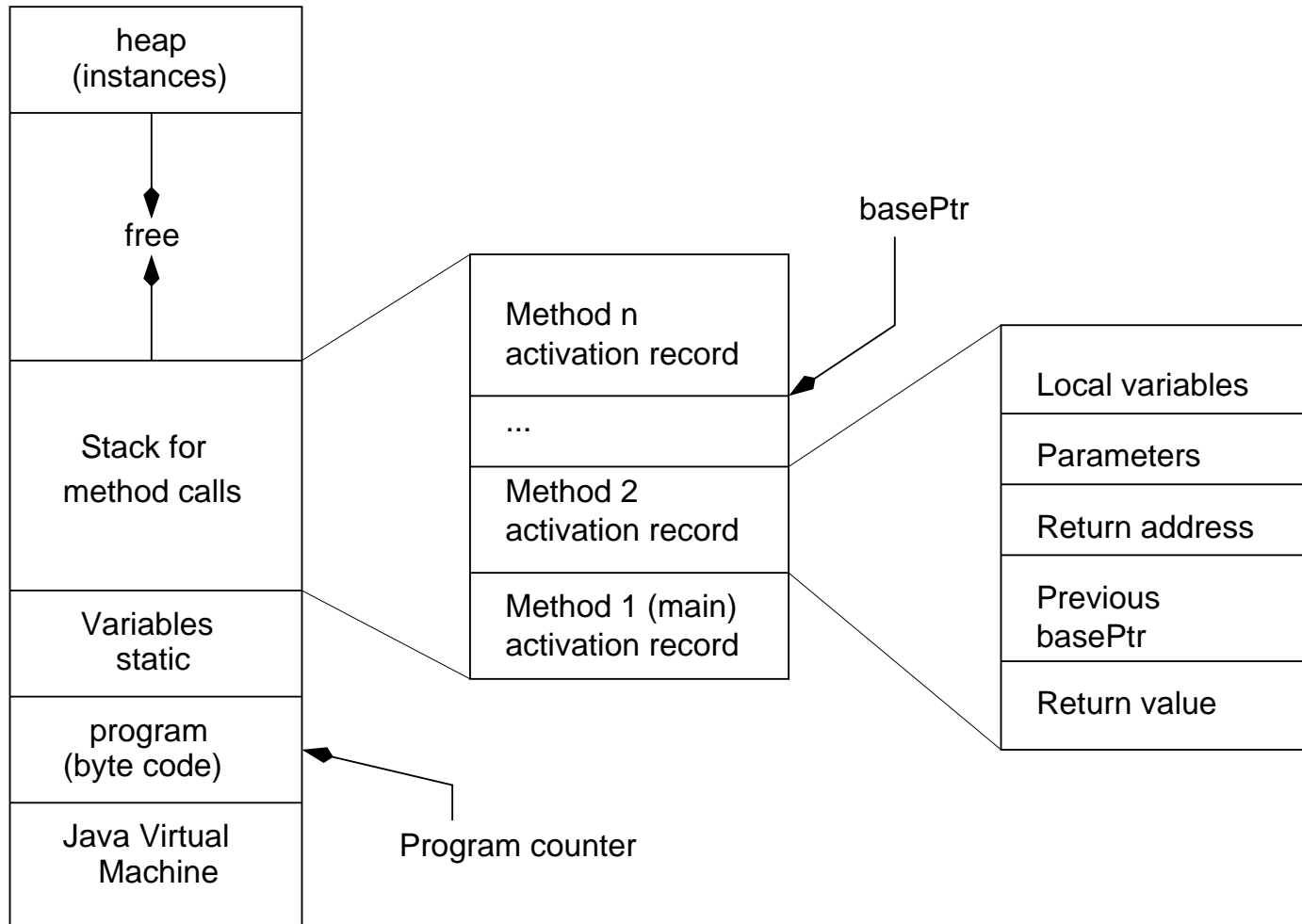
 L = POP Numbers

 Process L X R; PUSH result onto Numbers

We've seen an example where 'Process == Evaluate', then one where 'Process == Concatenate', but Process could also produce assembly code (i.e. machine instructions).

This shows how programs are compiled or translated.

Memory management



⇒ Schematic and simplified representation of the memory during the execution of a Java program.

Method call

The Java Virtual Machine (JVM) must:

1. Create a new activation record/block (which contains space for the local variables and the parameters among other things);
2. Save the current value of basePtr in the activation record and set the basePtr to the address of the current record;
3. Save the value of the program counter in the designated space of the activation record, set the program counter to the first instruction of the current method;
4. Copy the values of the effective parameters into the designated area of the current activation record;
5. Initial the local variables;
6. Start executing the instruction designated by the program counter.

⇒ activation block = *stack frame, call frame or activation record.*

When the method ends

The JVM must:

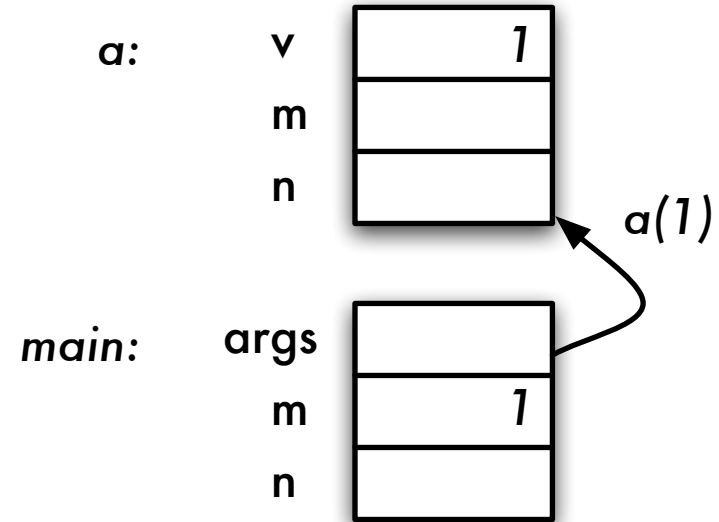
1. Save the return value (at the designated space)
2. Return the control to the calling method, i.e. set the program counter and basePtr back to their previous value;
3. Remove the current block;
4. Execute instruction designated by the current value of the program counter.

Example 1 (simplified)

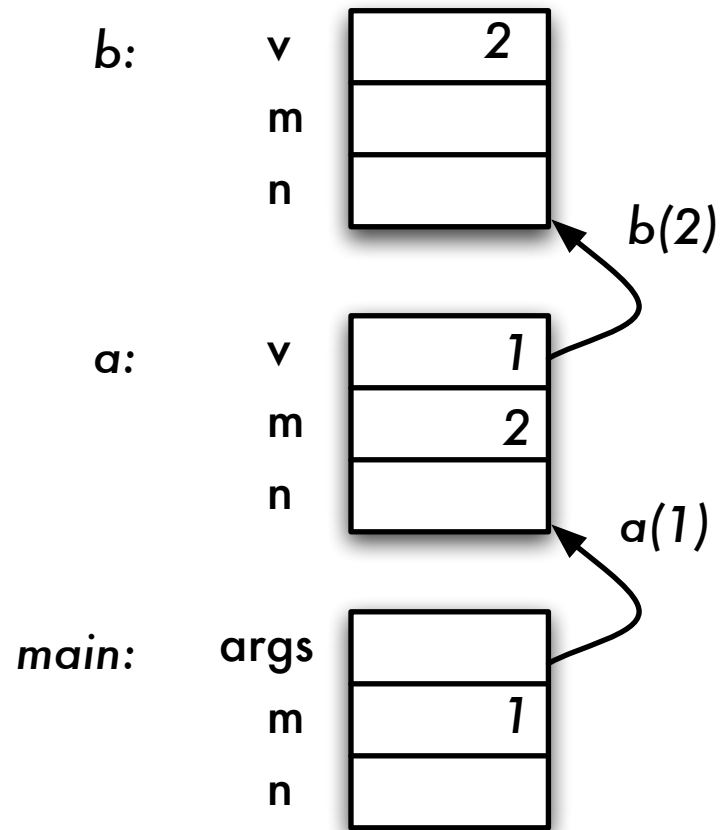
```
public class Calls {
    public static int c( int v ) {
        int n;
        n = v + 1;
        return n;
    }
    public static int b( int v ) {
        int m,n;
        m = v + 1;
        n = c( m );
        return n;
    }
    public static int a( int v ) {
        int m,n;
        m = v + 1;
        n = b( m );
        return n;
    }
}
```

```
public static void main( String[] args ) {  
    int m,n;  
    m = 1;  
    n = a( m );  
    System.out.println( n );  
}  
}
```

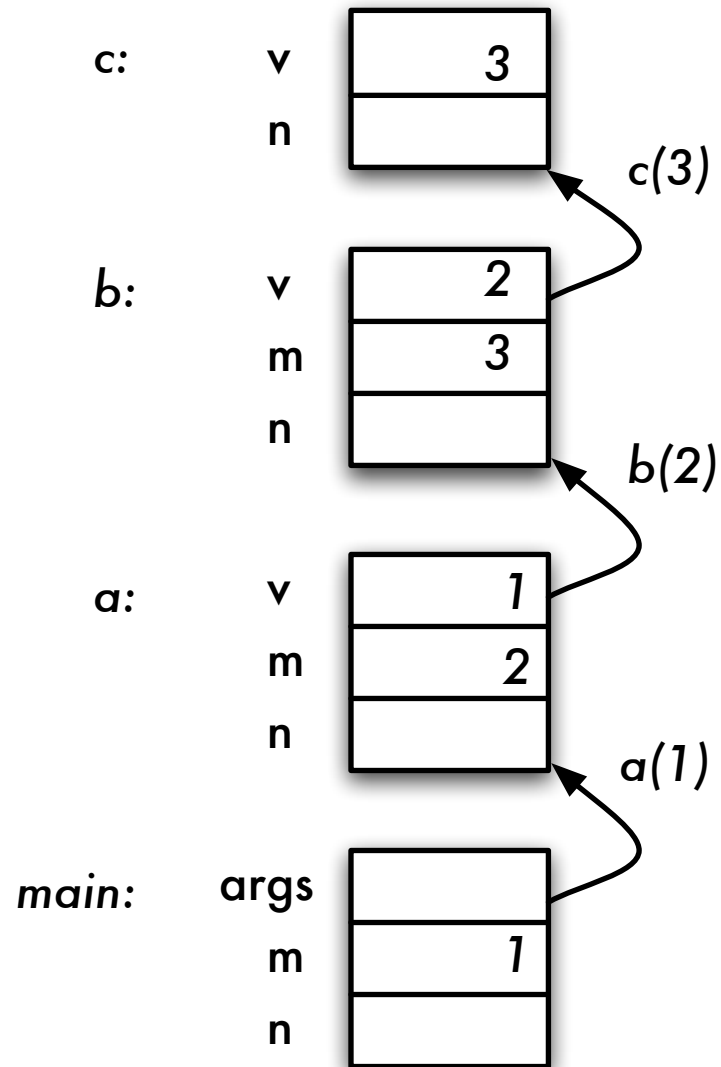
Example 1 (simplified)



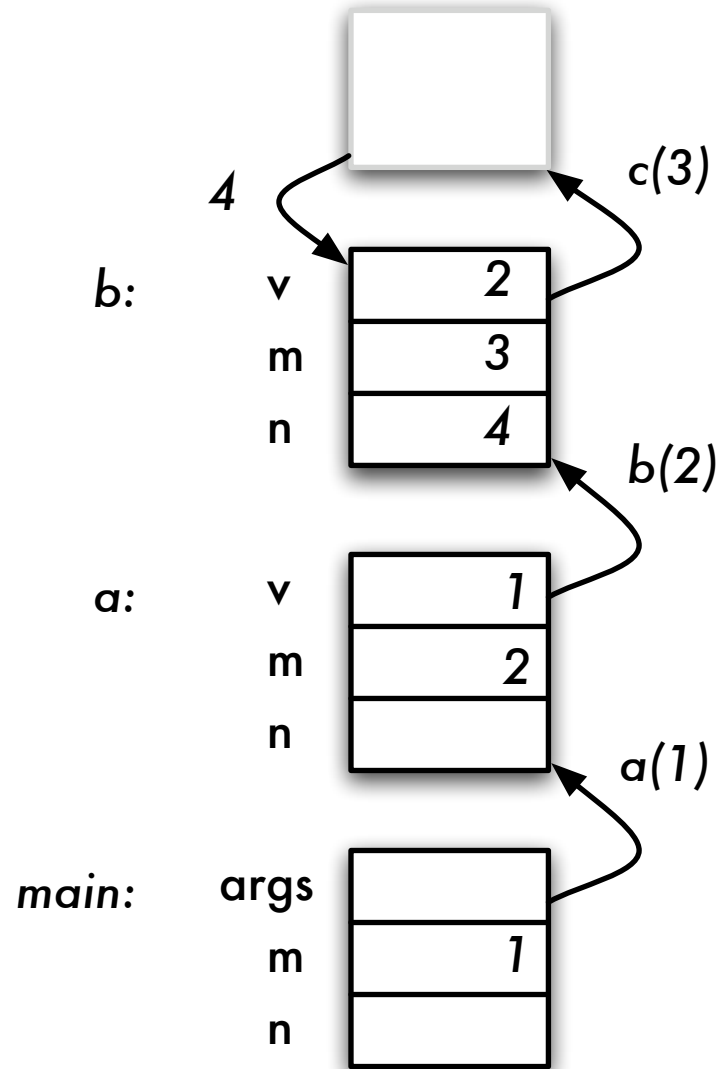
Example 1 (simplified)



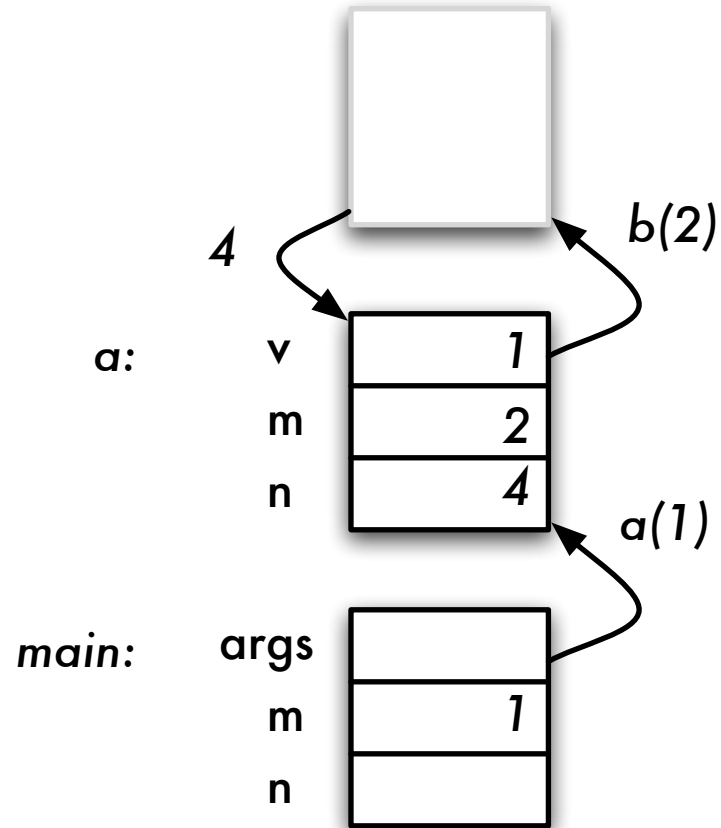
Example 1 (simplified)



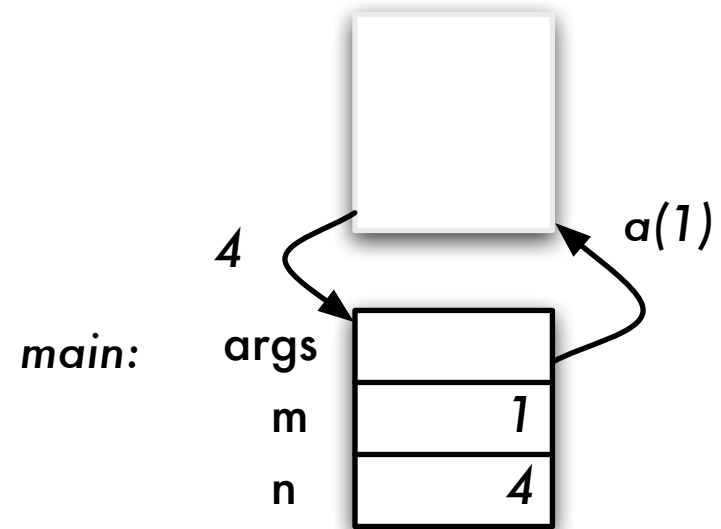
Example 1 (simplified)



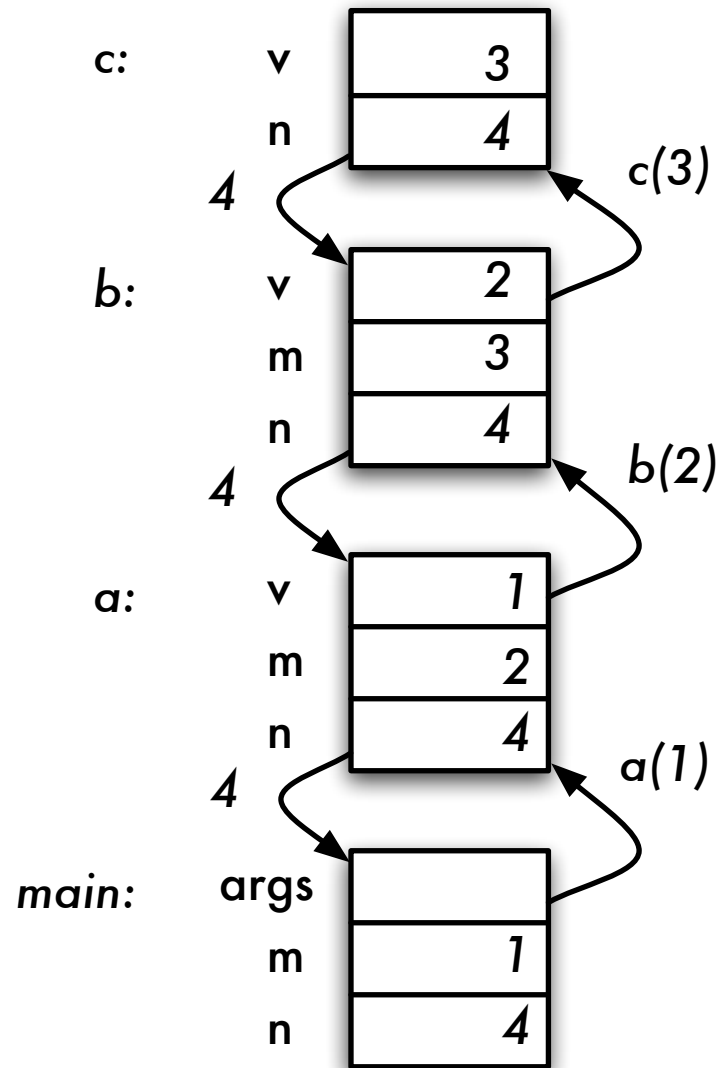
Example 1 (simplified)



Example 1 (simplified)



Example 1: summary



Example 2 (with a program counter)

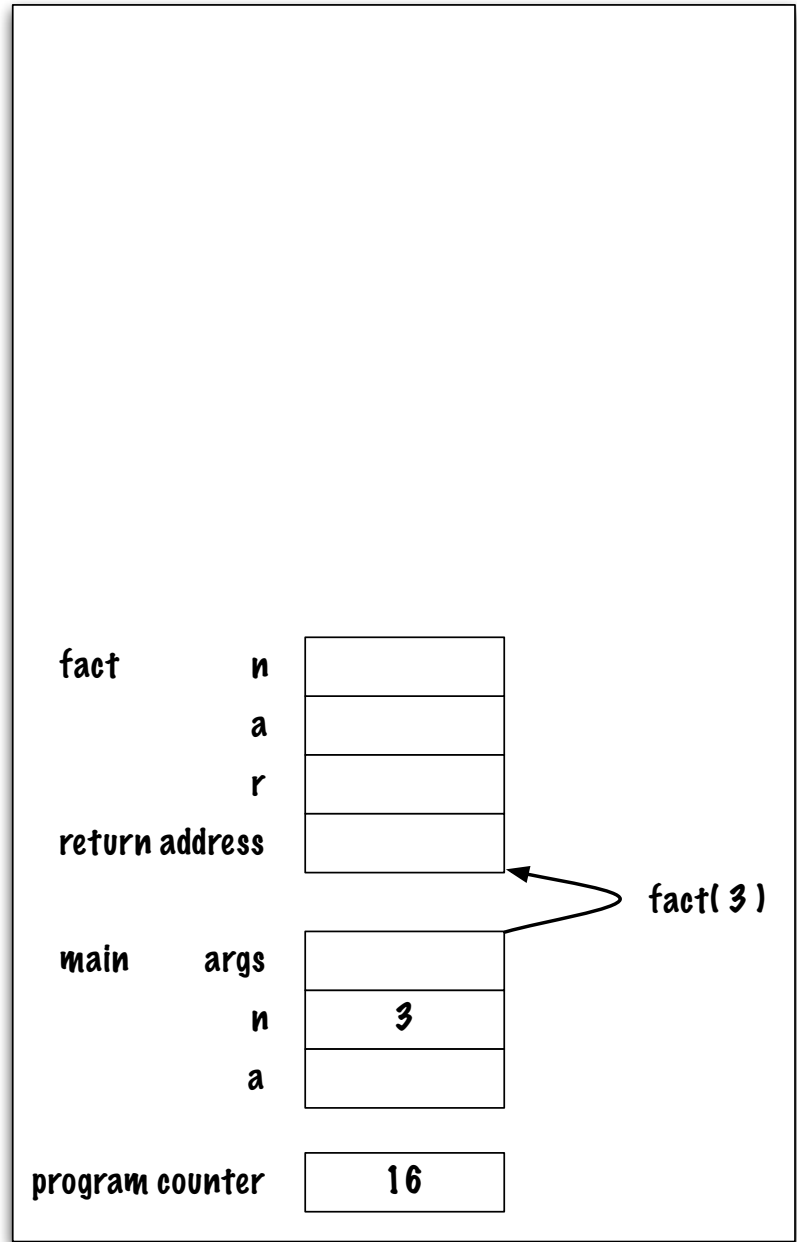
```
01 public class Fact {
02     public static int fact( int n ) {
03         // pre-condition: n >= 0
04         int a, r;
05         if ( n == 0 || n == 1 ) {
06             a = 1;
07         } else {
08             r = fact( n-1 );
09             a = n * r;
10         }
11         return a;
12     }
13     public static void main( String[] args ) {
14         int a, n;
15         n = 3;
16         a = fact( n );
17     }
18 }
```

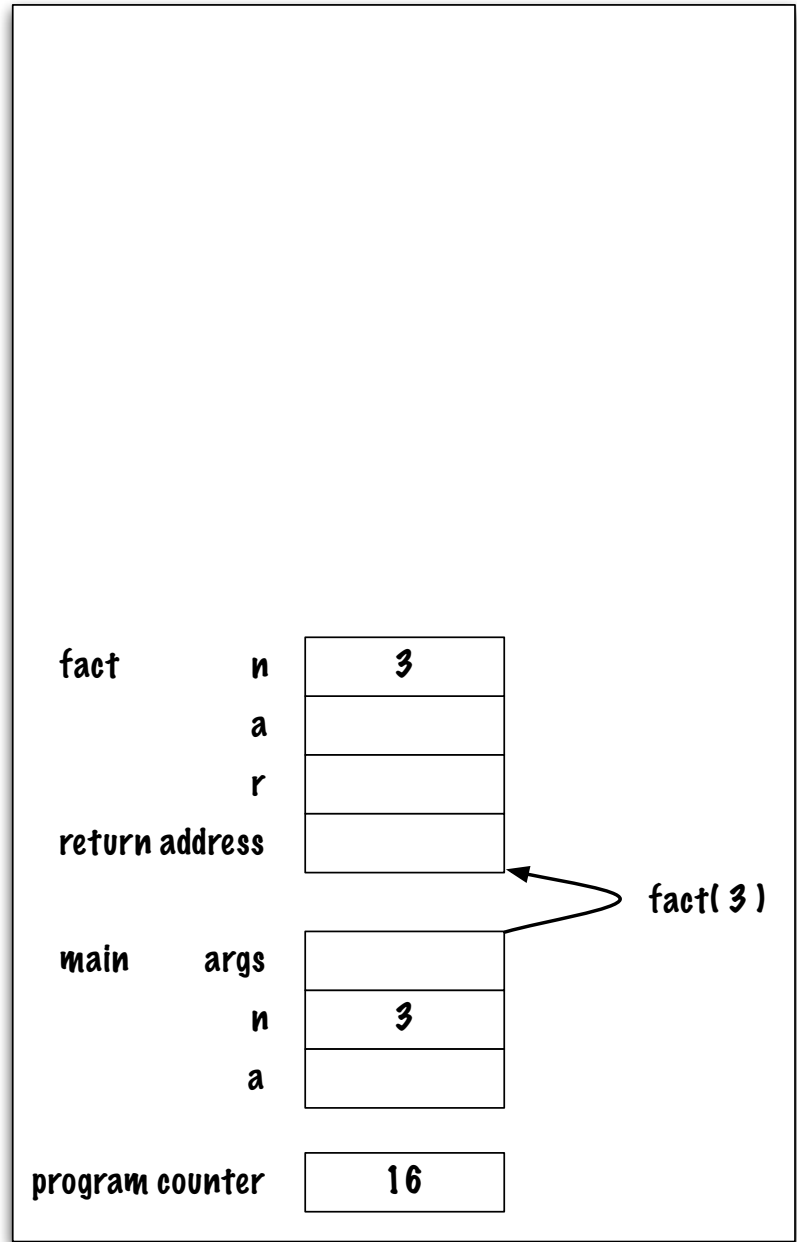
| | | |
|-------------|-------------|--|
| main | args | |
| | n | |
| | a | |

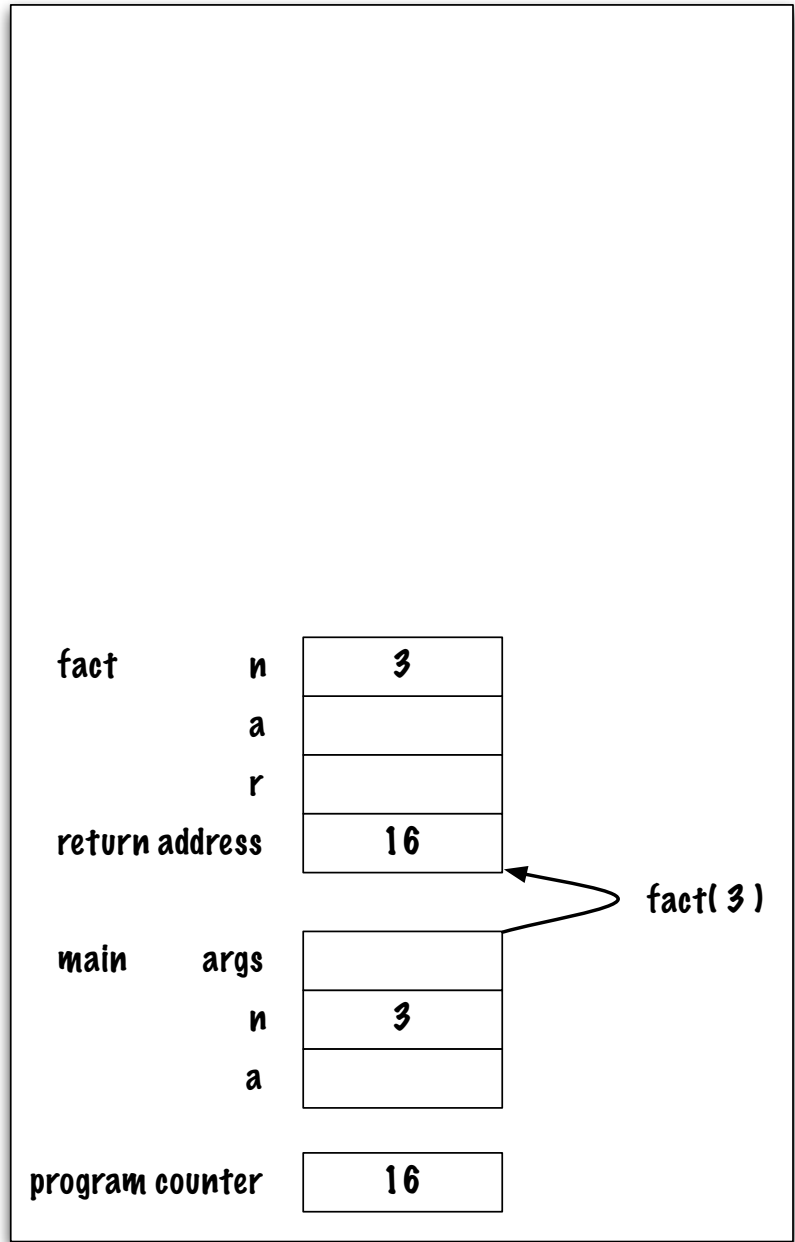
| | |
|------------------------|-----------|
| program counter | 15 |
|------------------------|-----------|

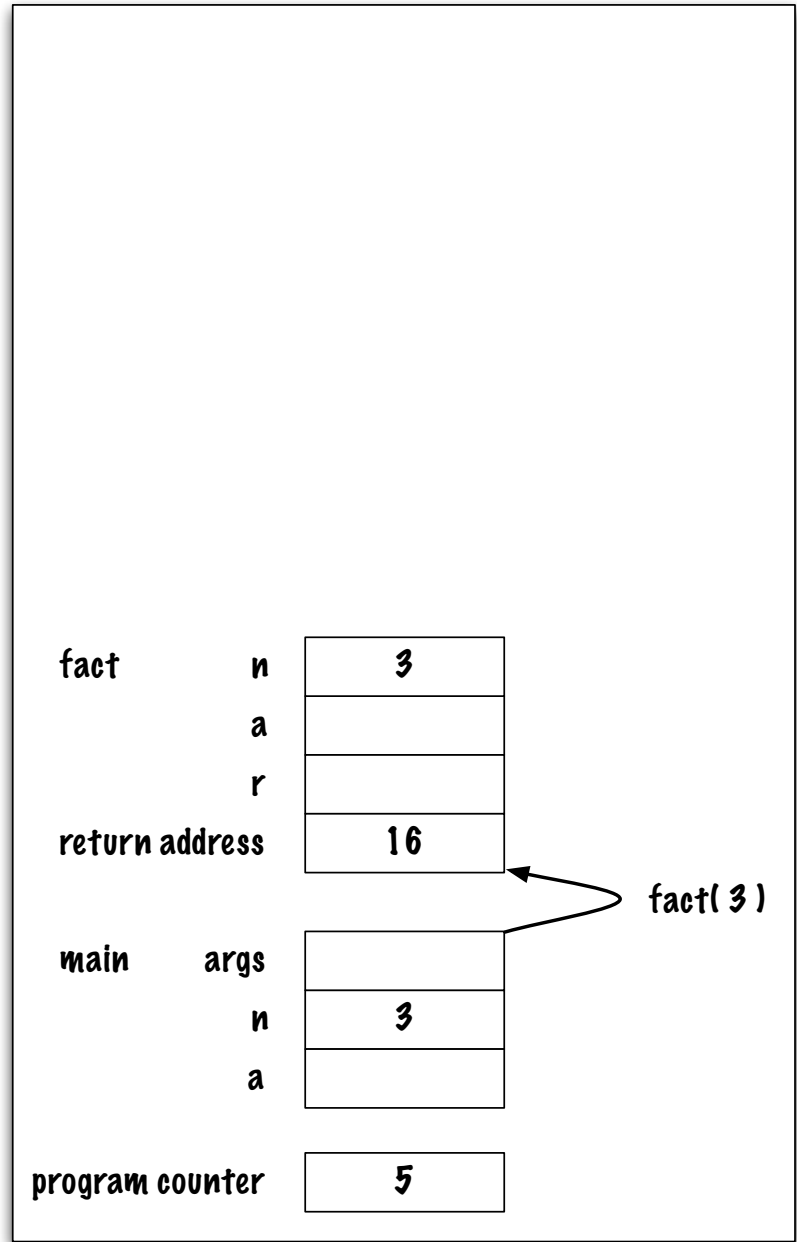
| | | |
|-------------|-------------|----------|
| main | args | |
| | n | 3 |
| | a | |

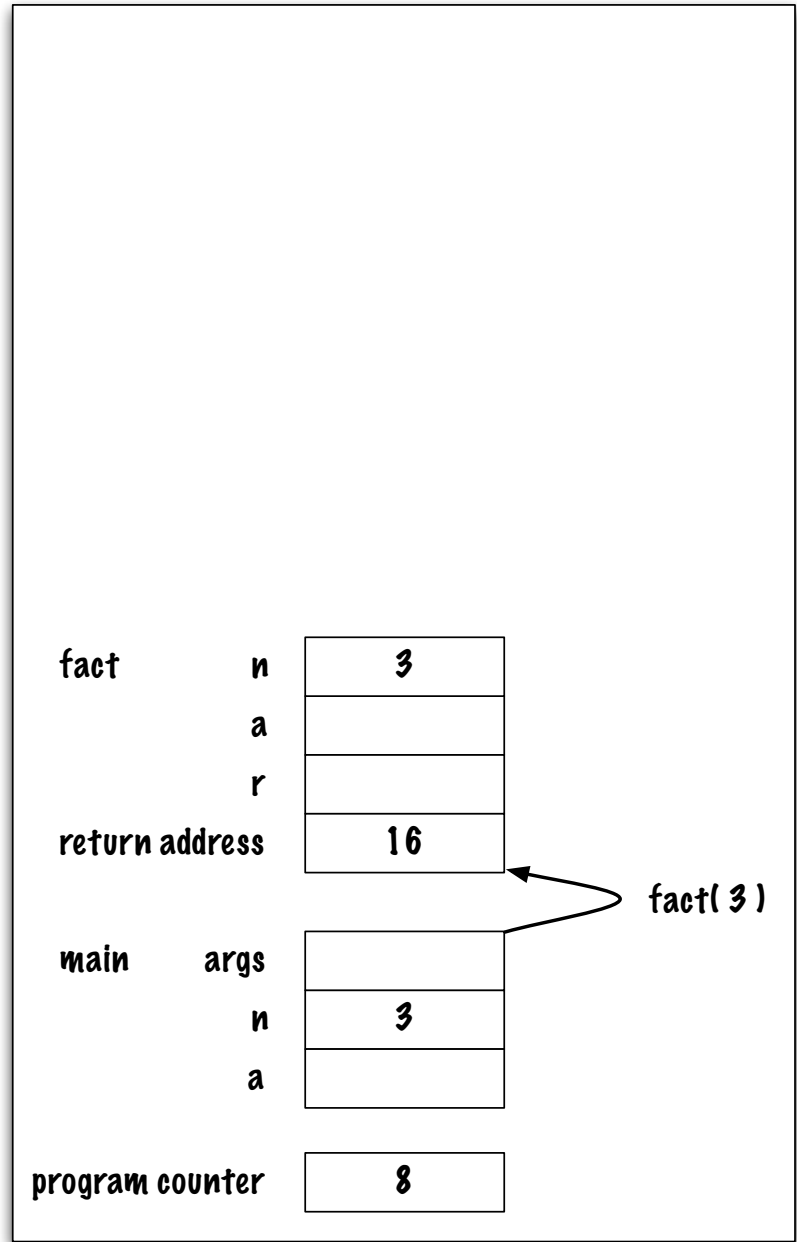
program counter **16**

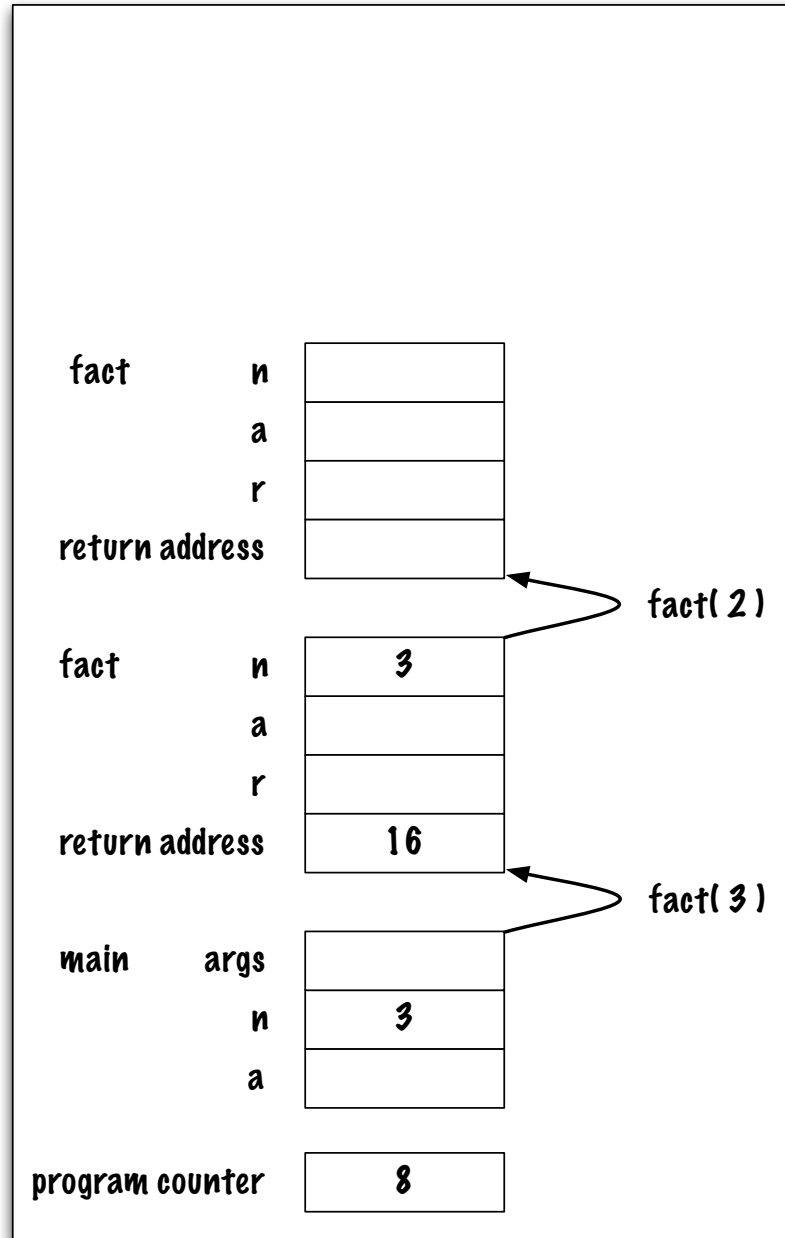


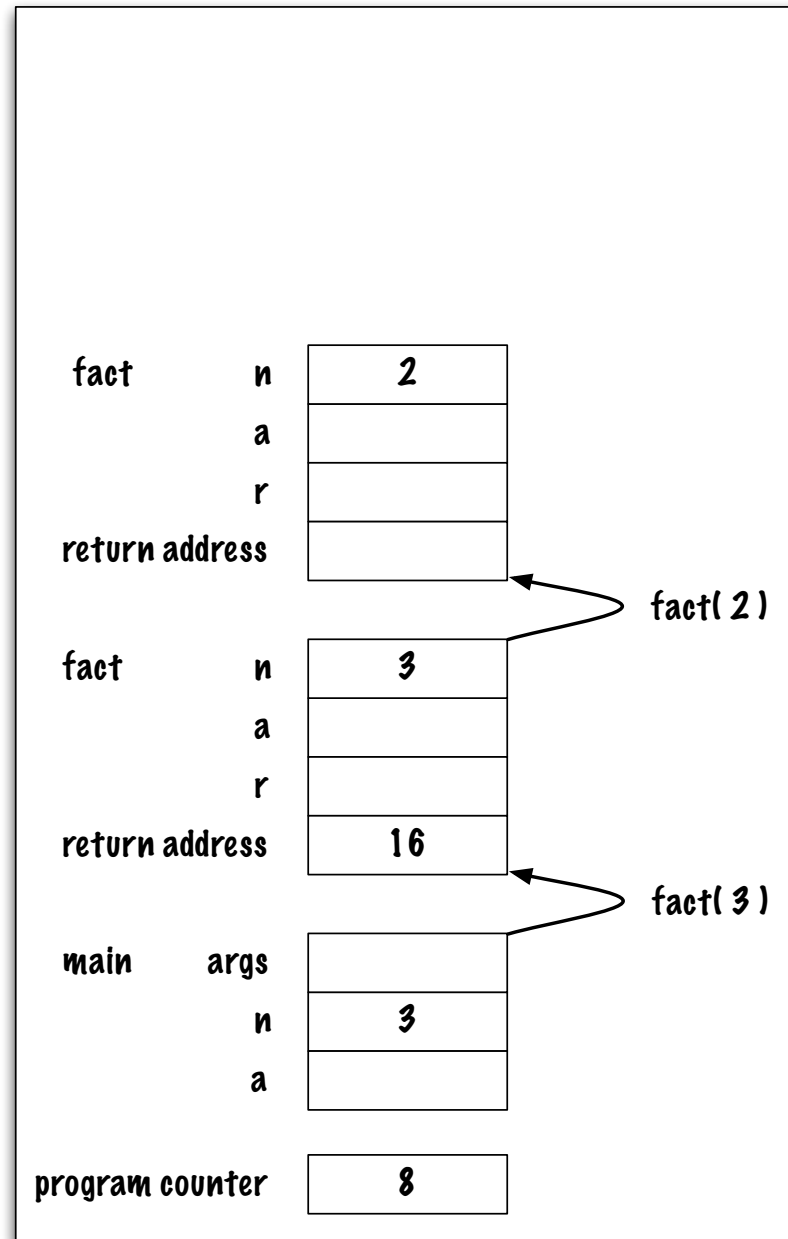


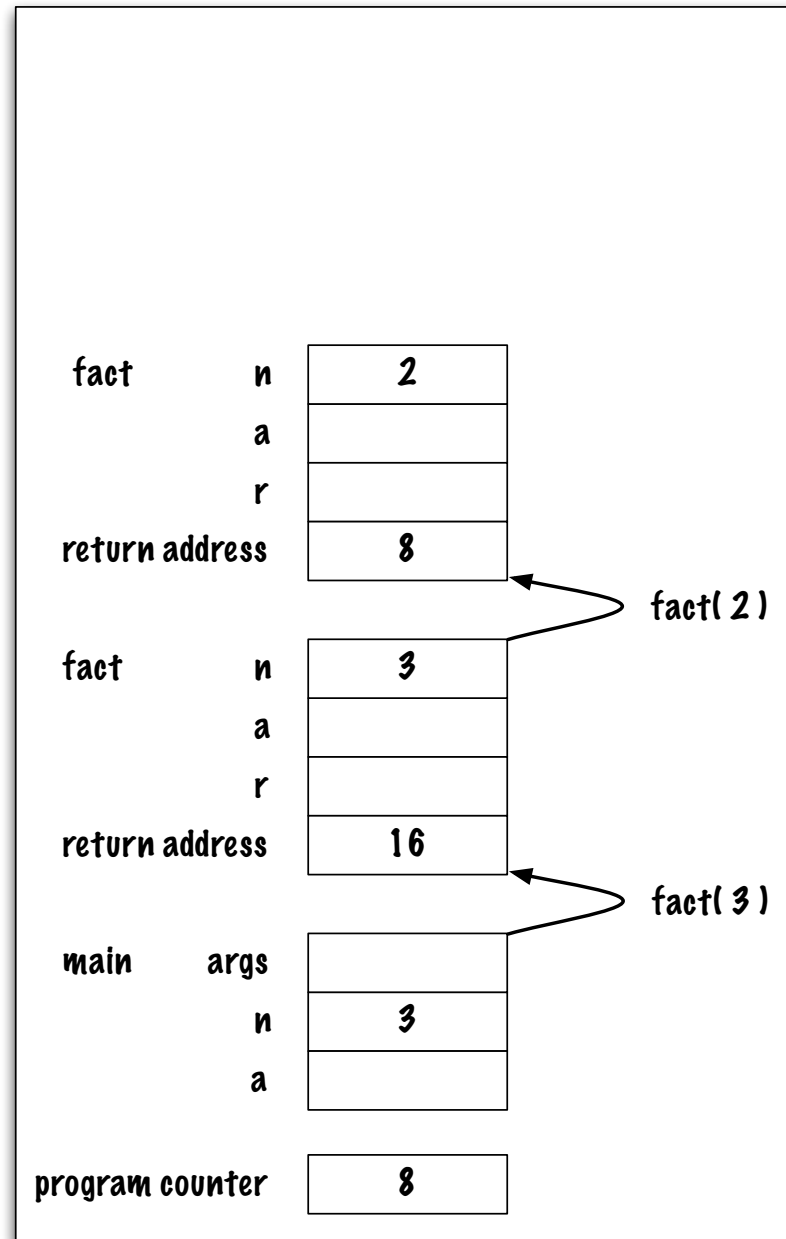


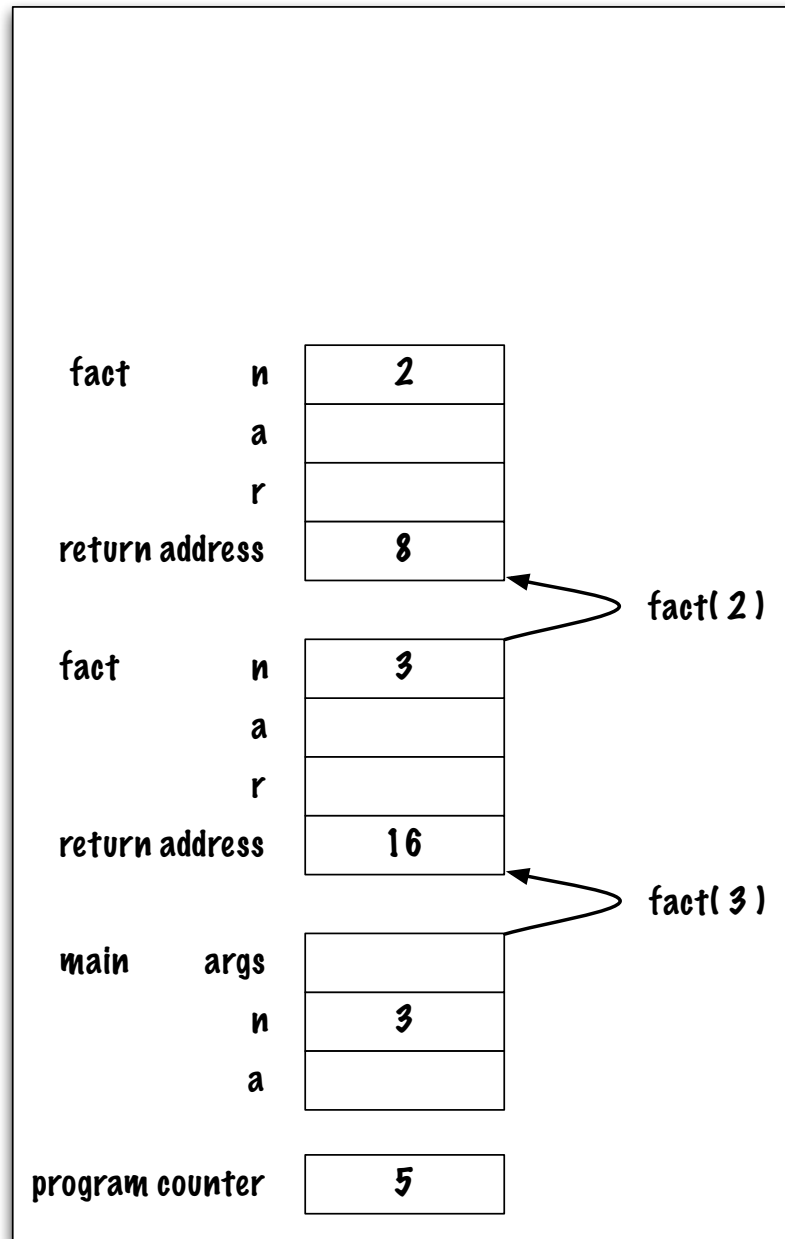


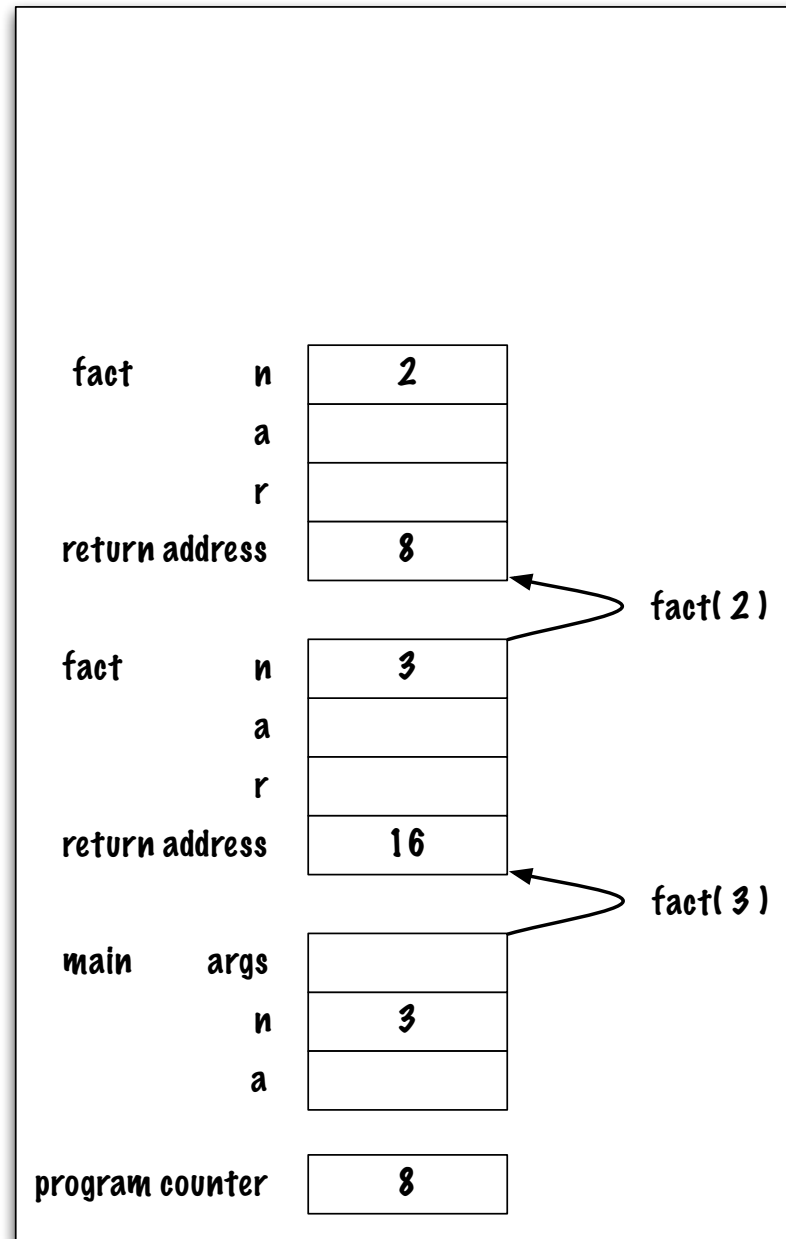


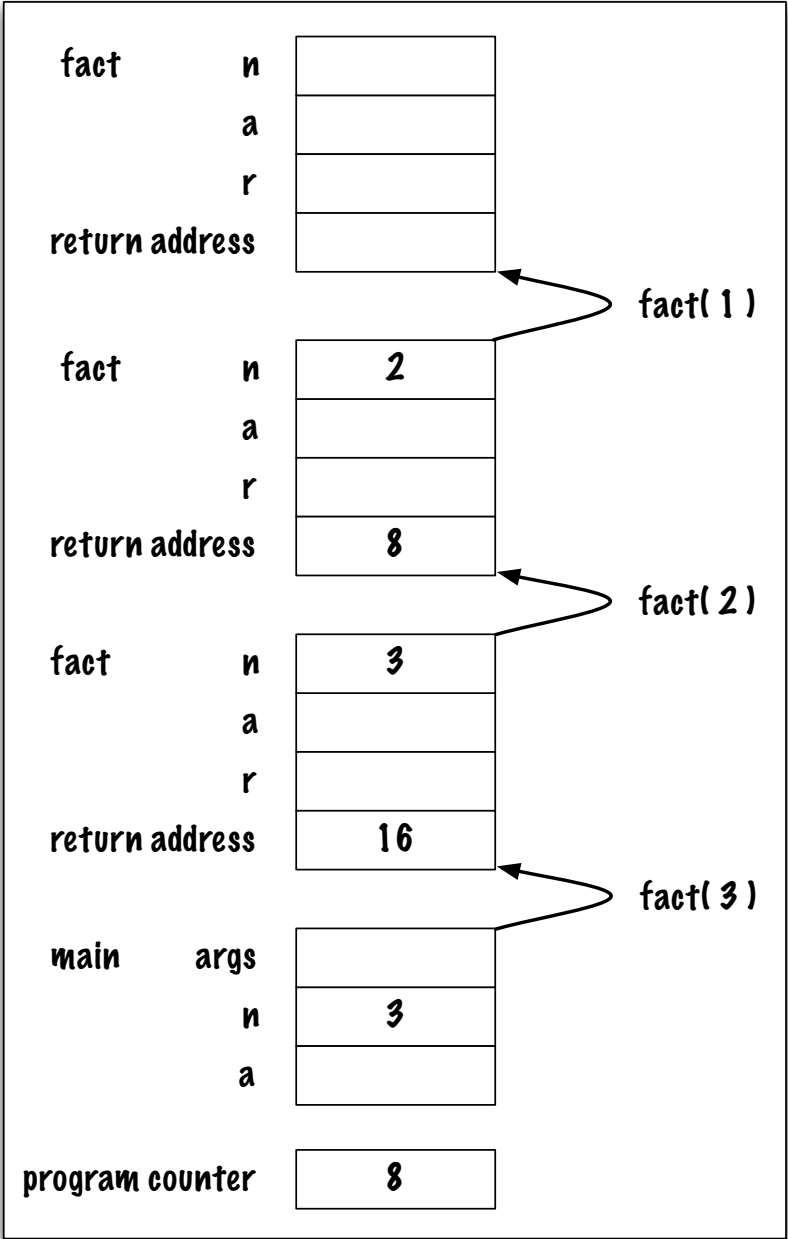


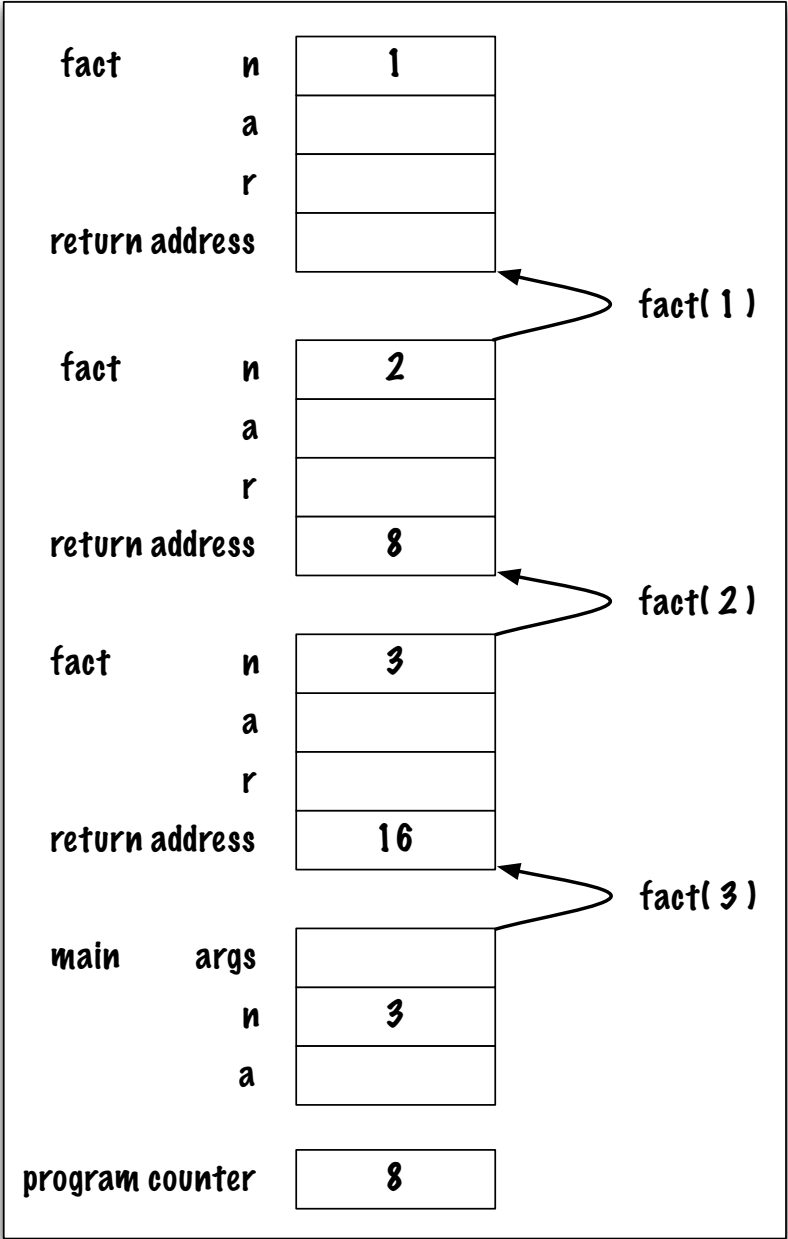


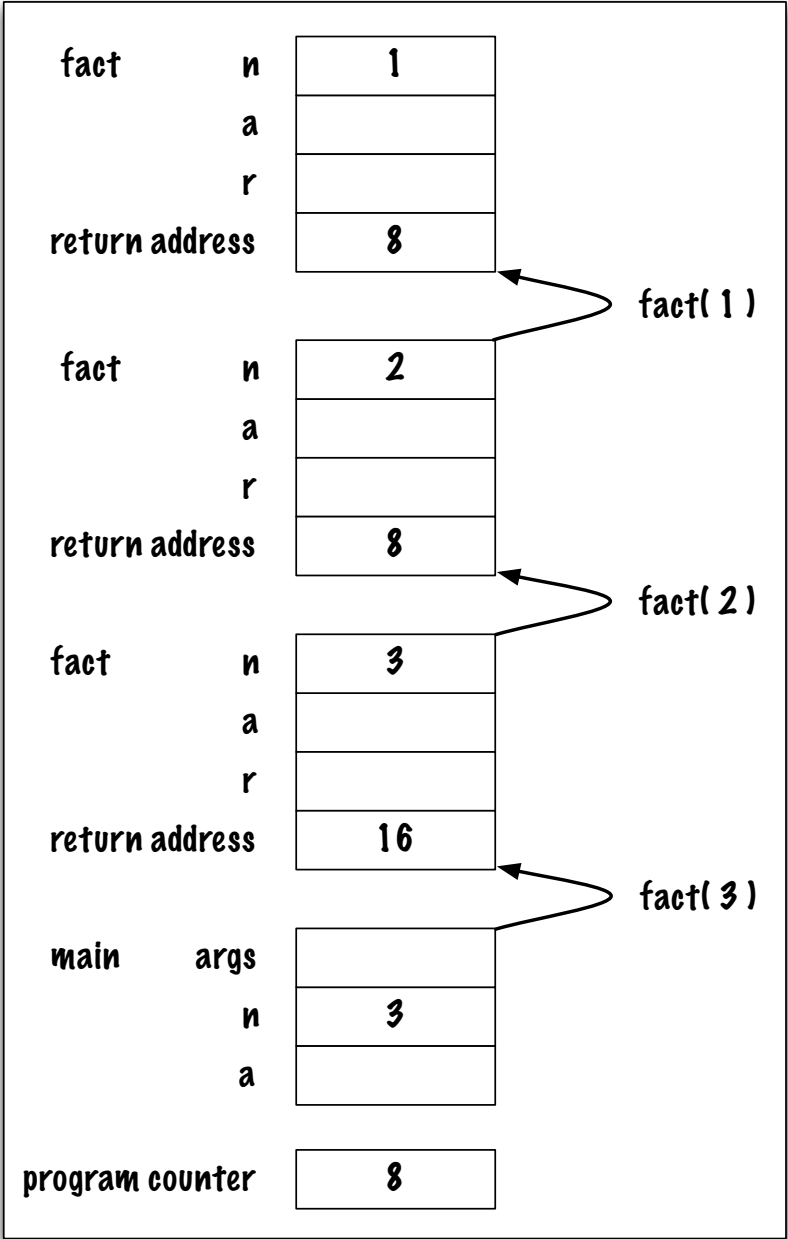


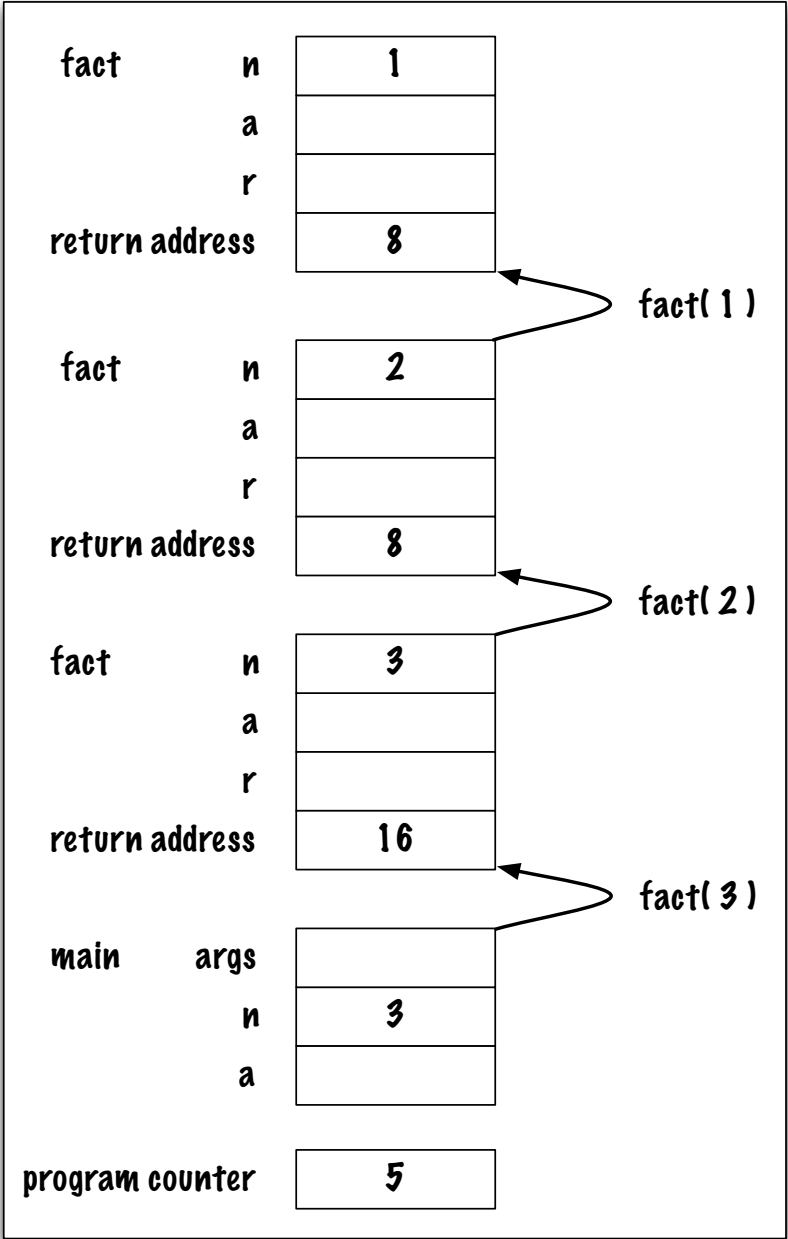


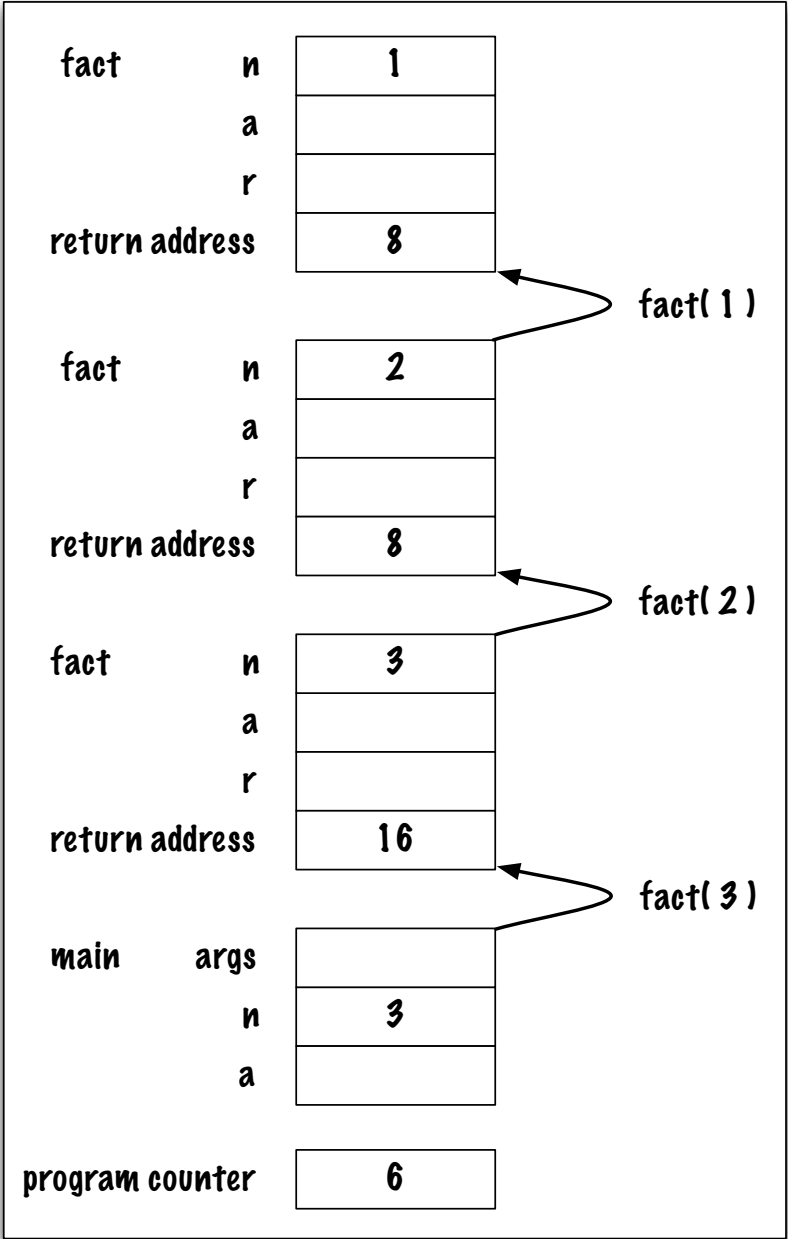


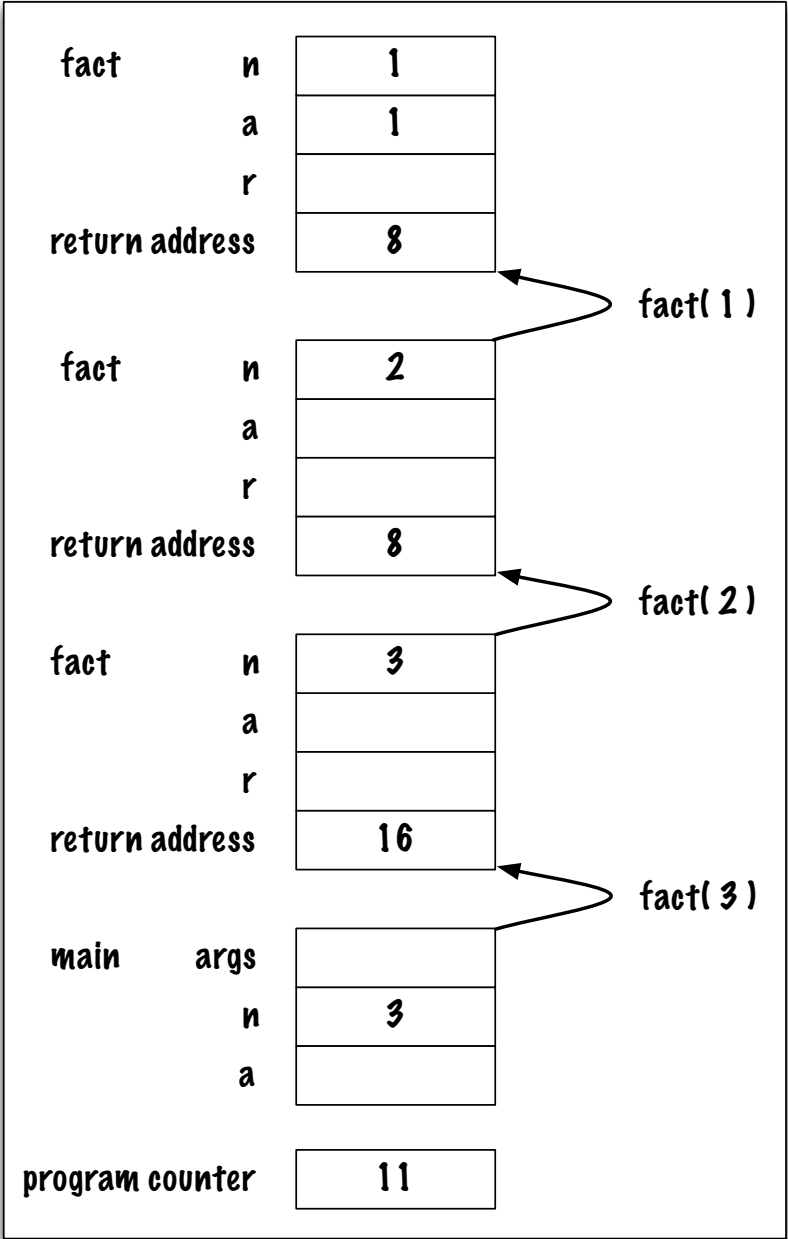


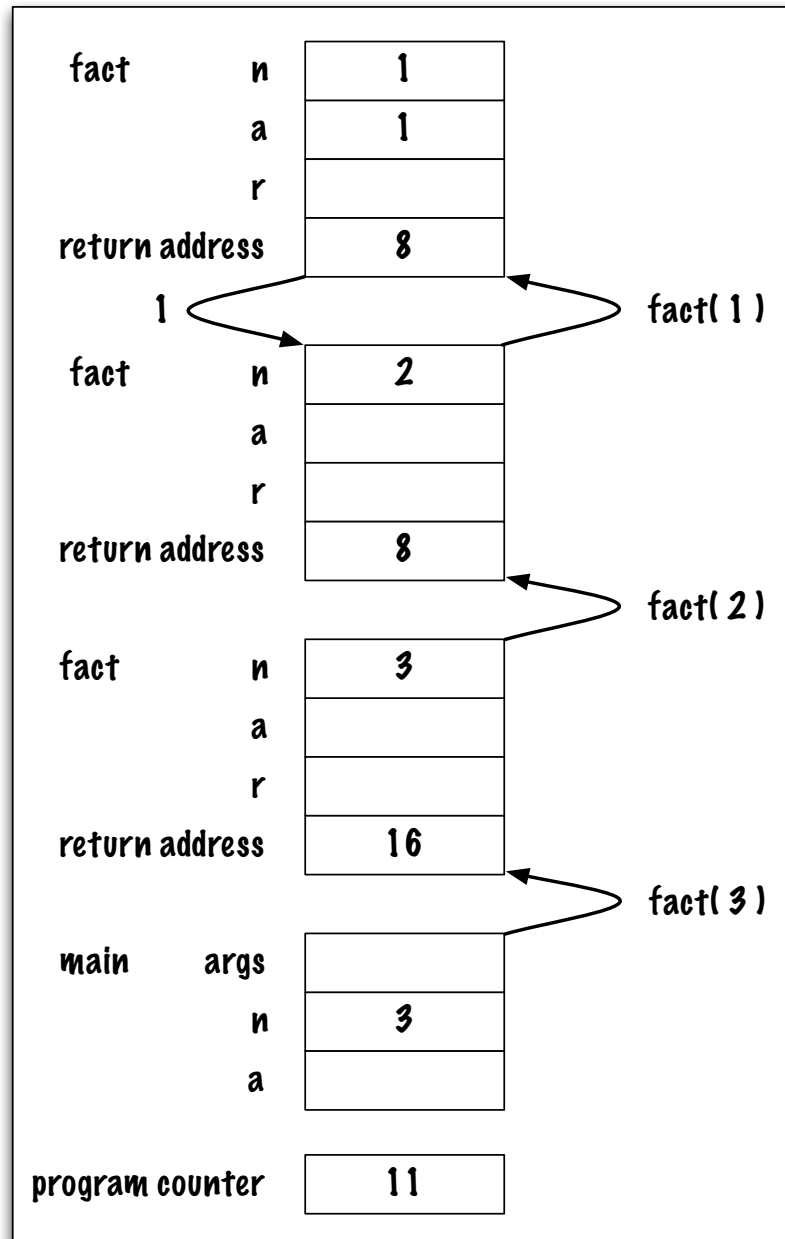


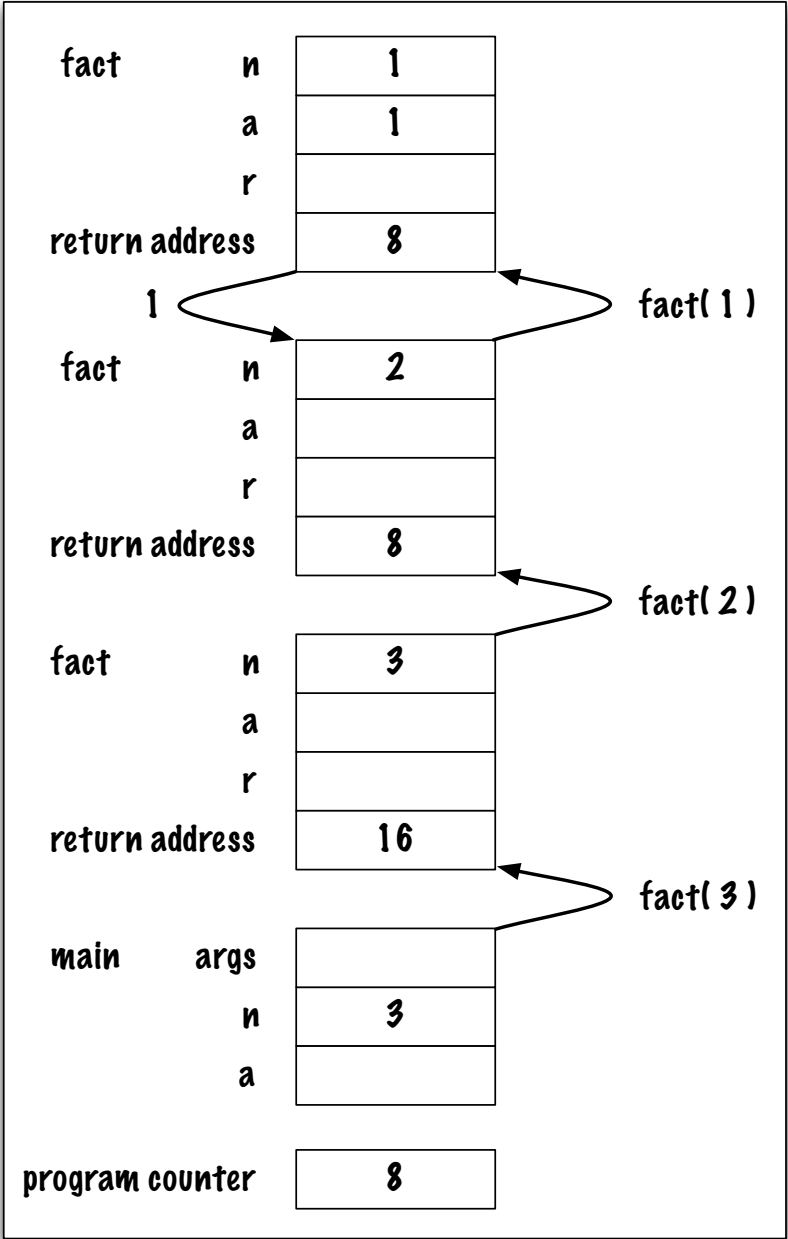


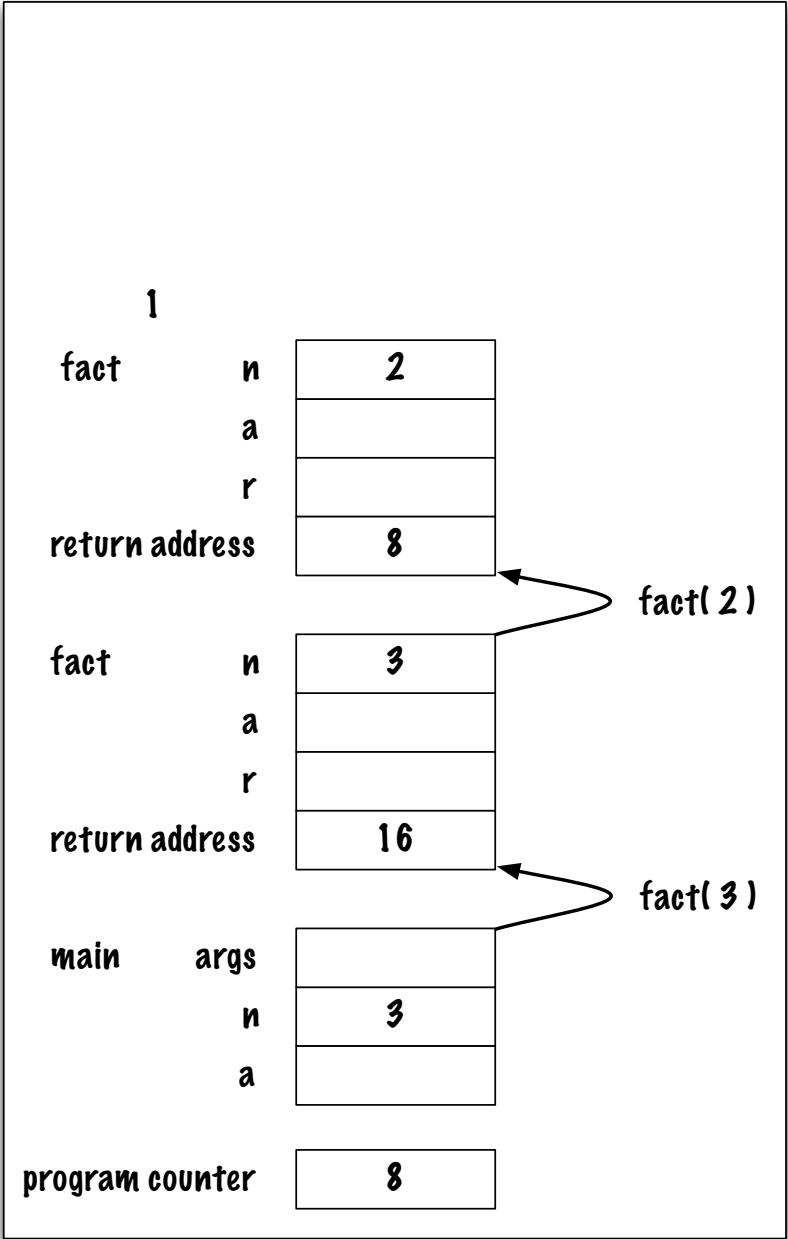


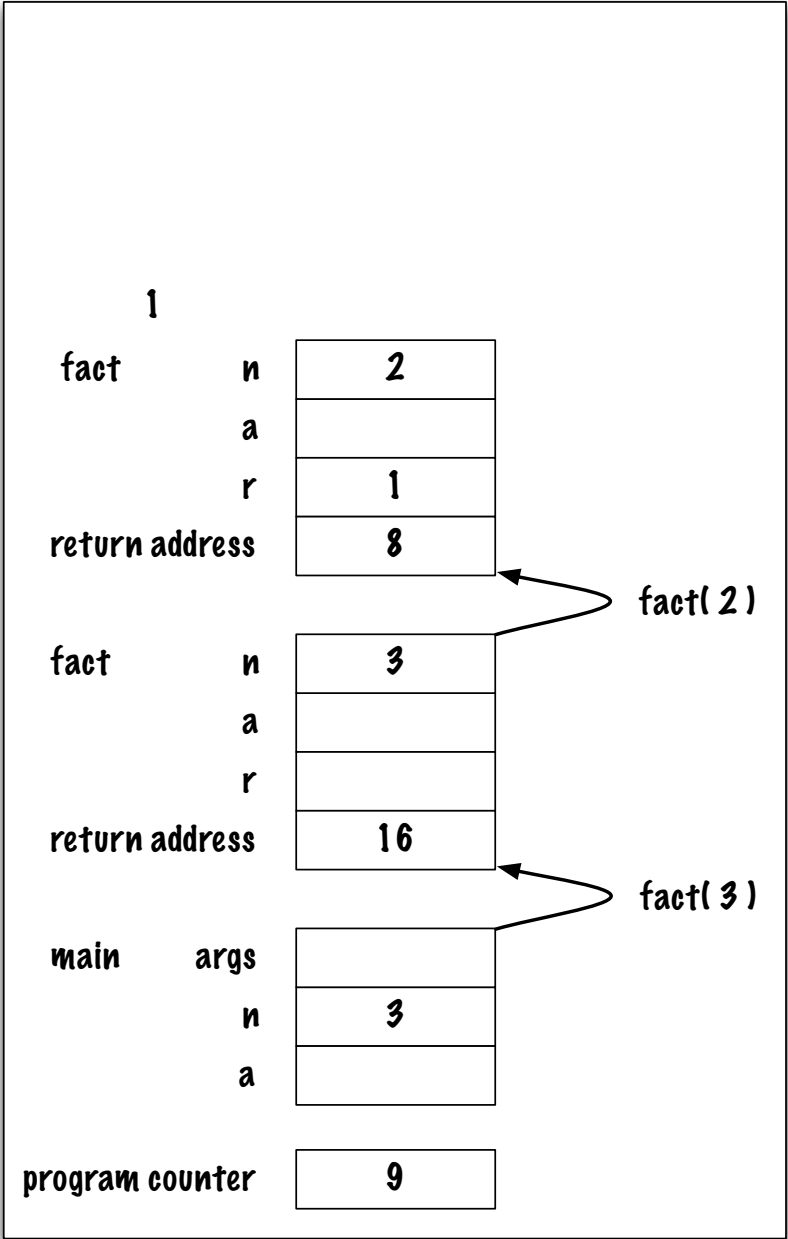


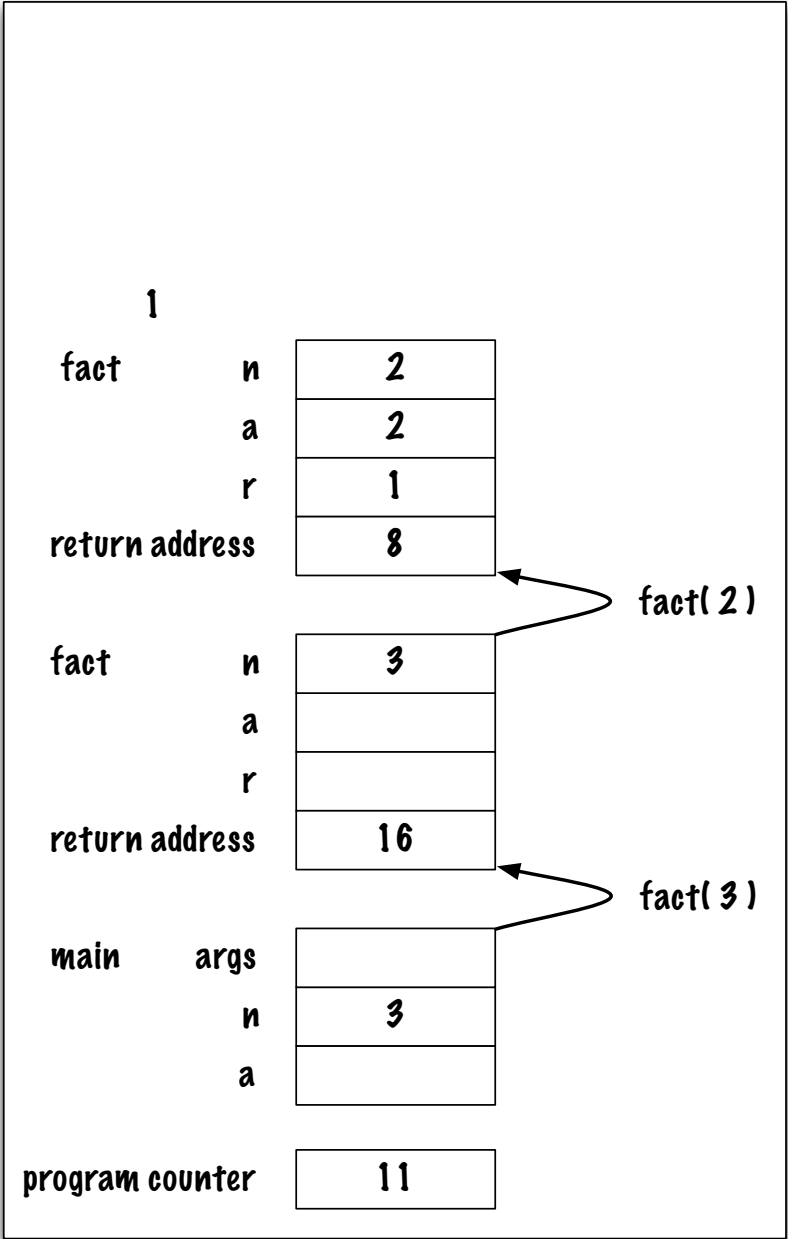


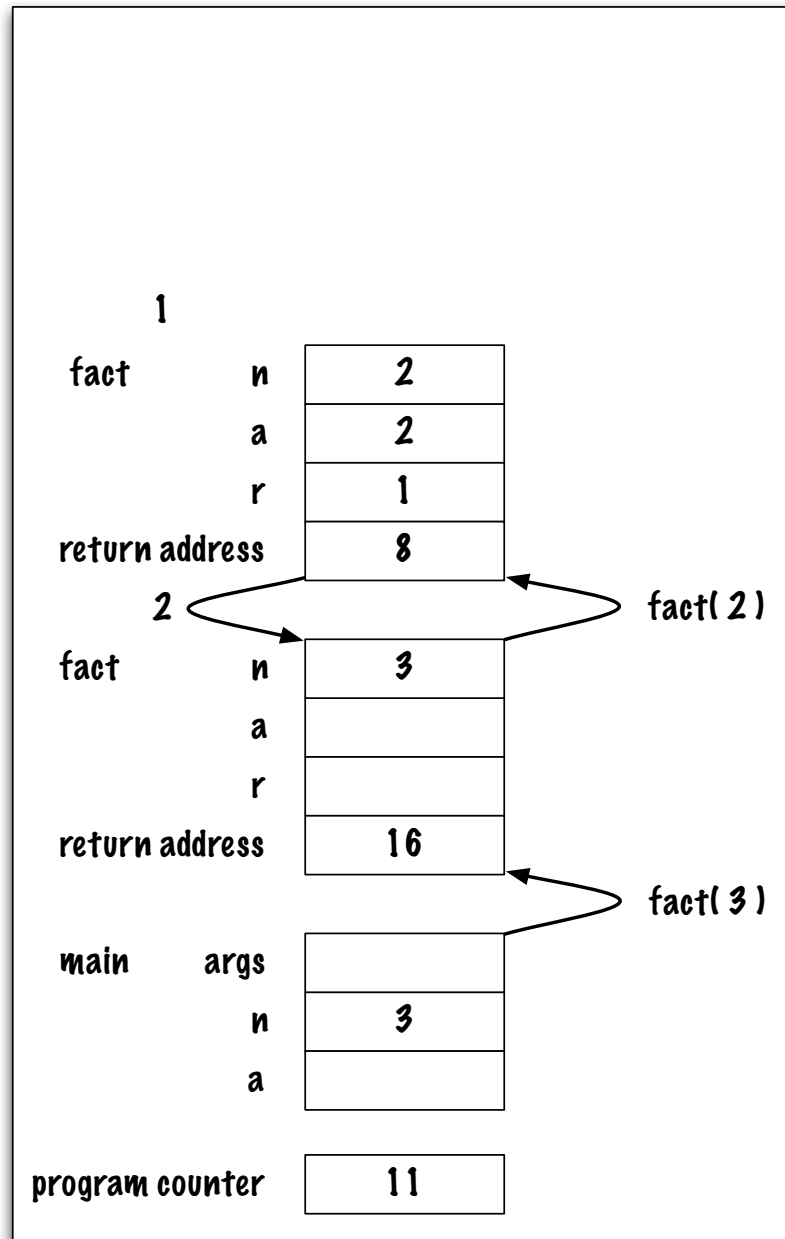


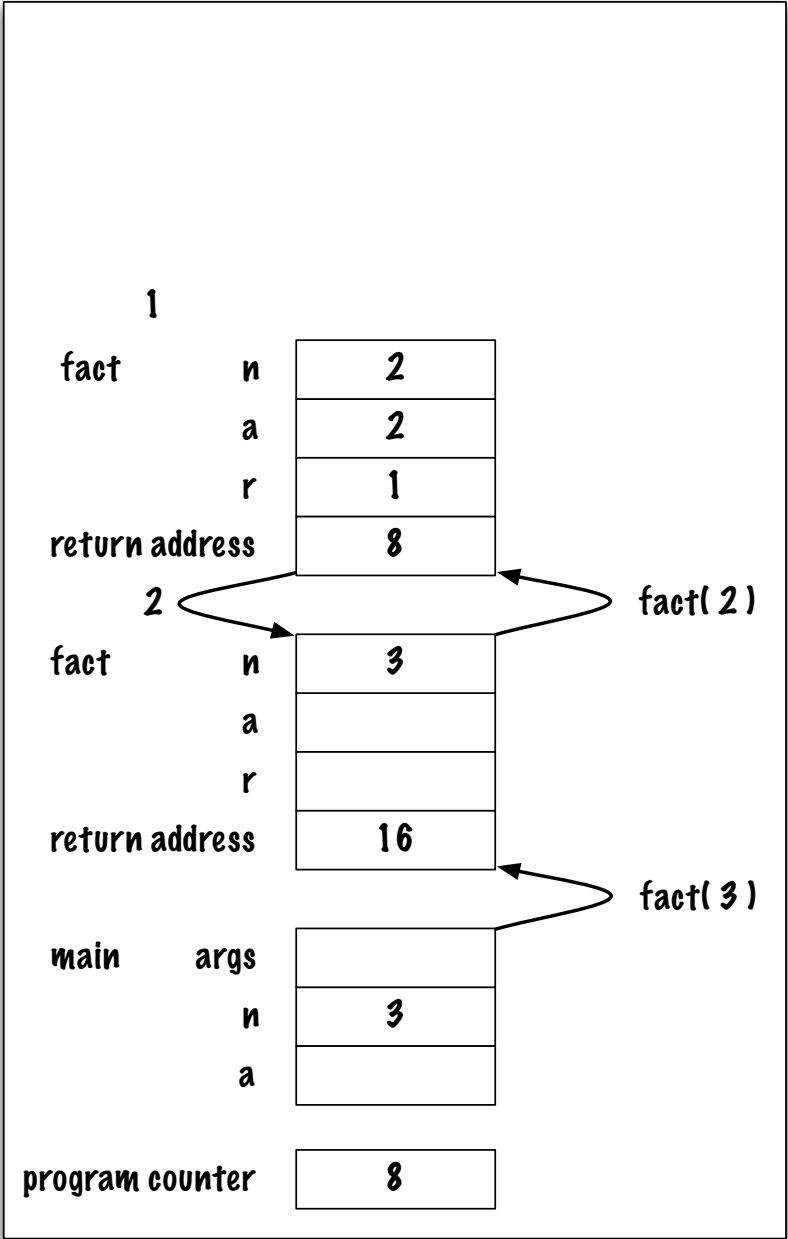


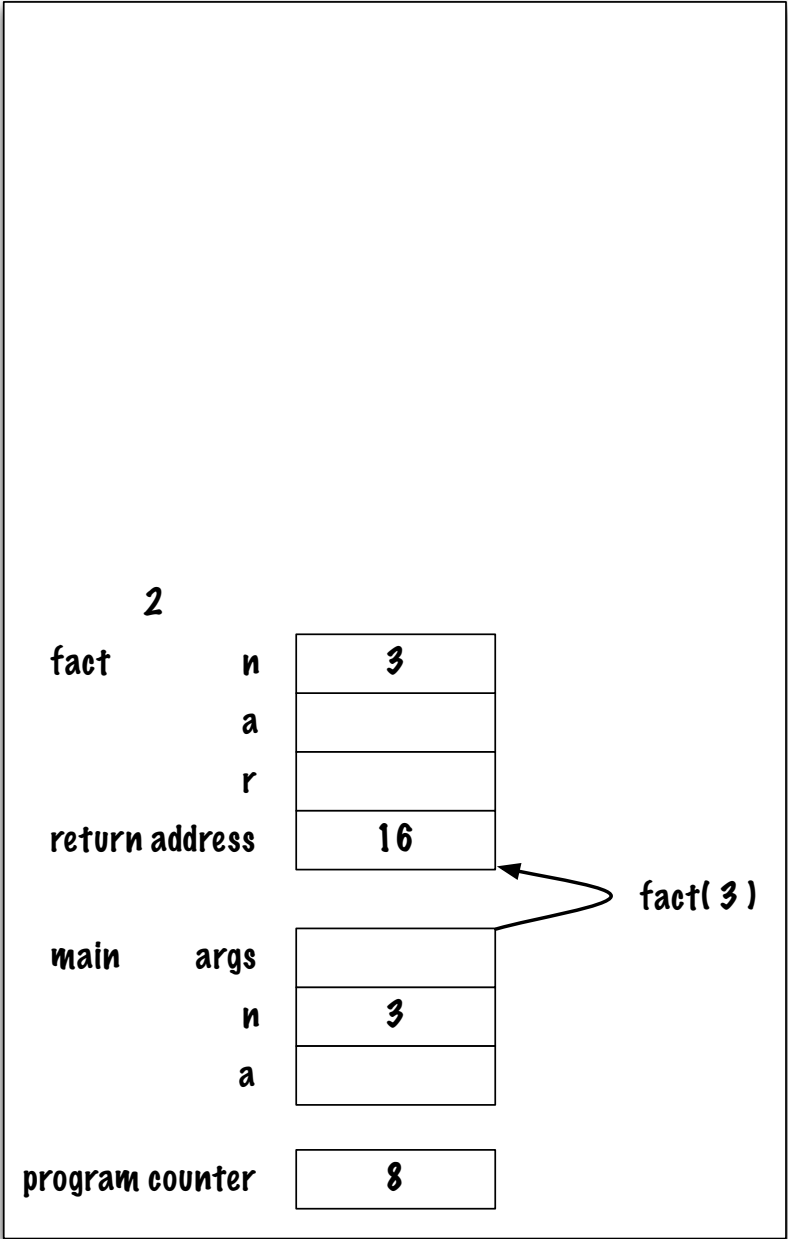


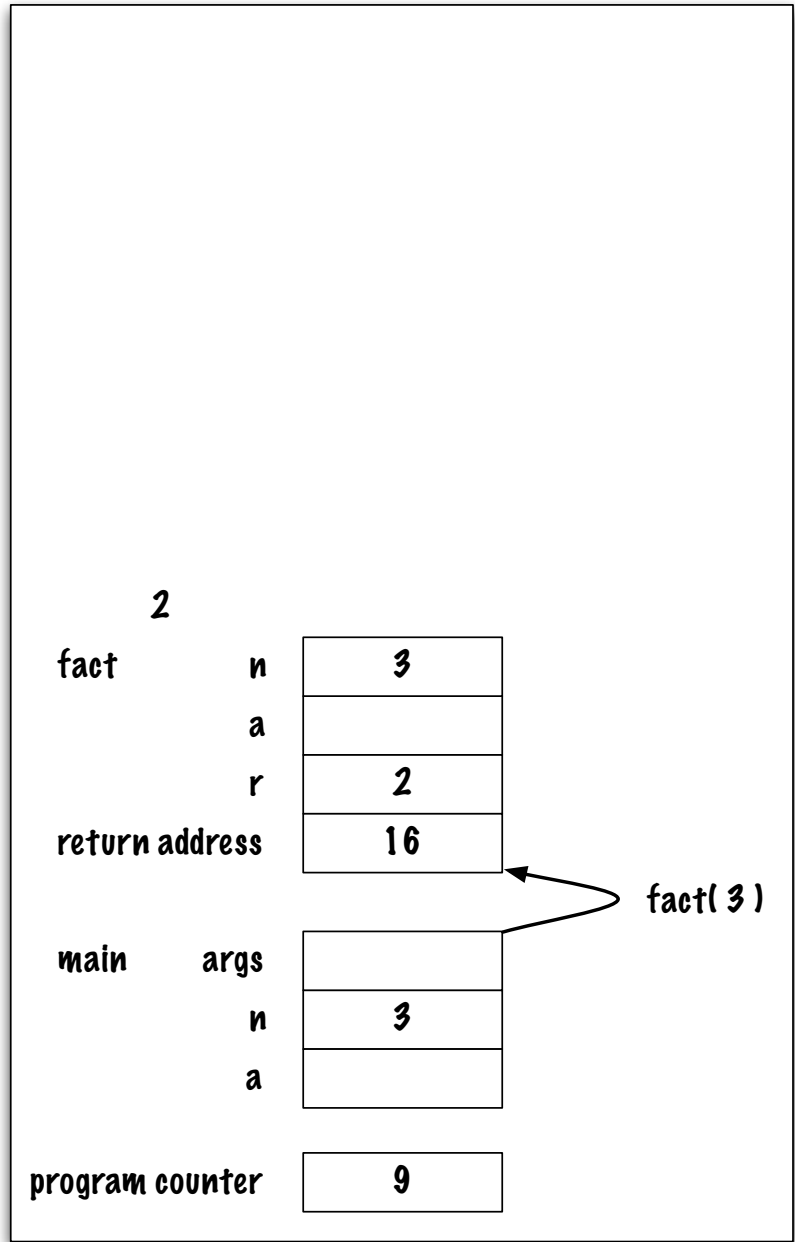


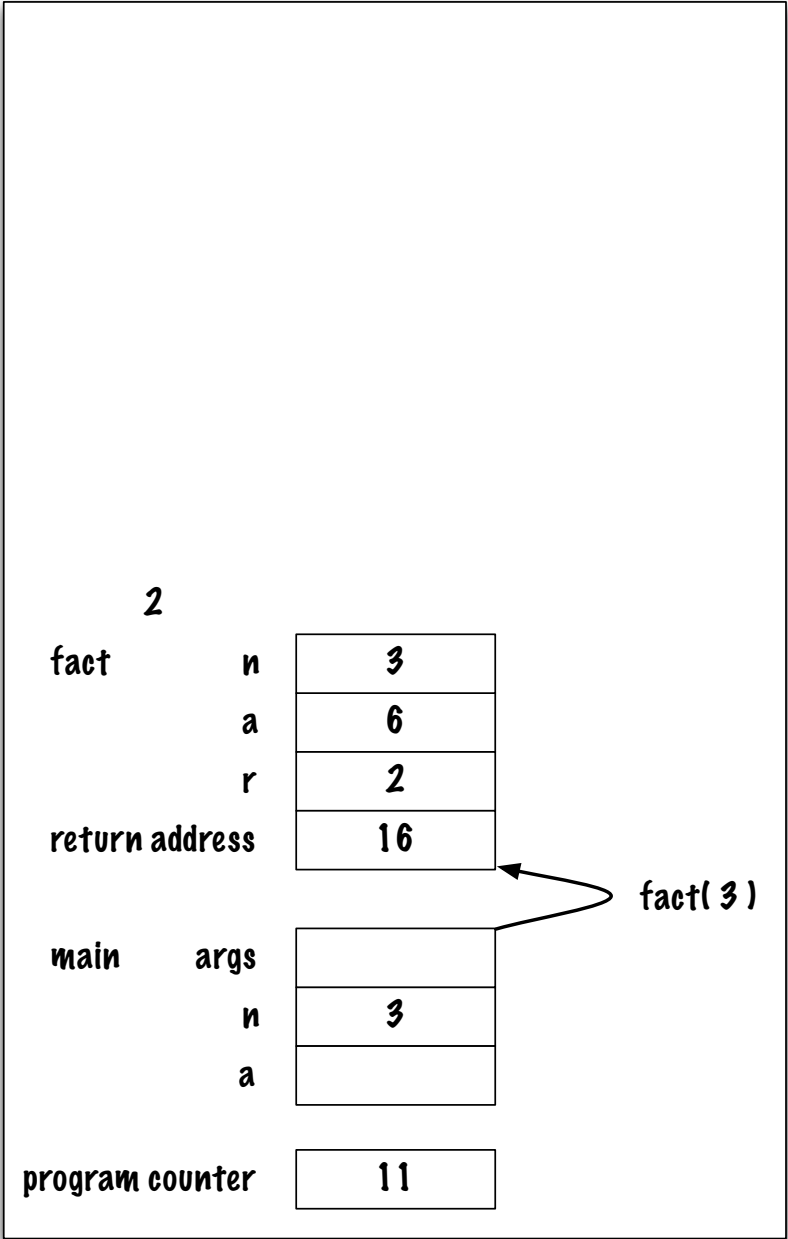


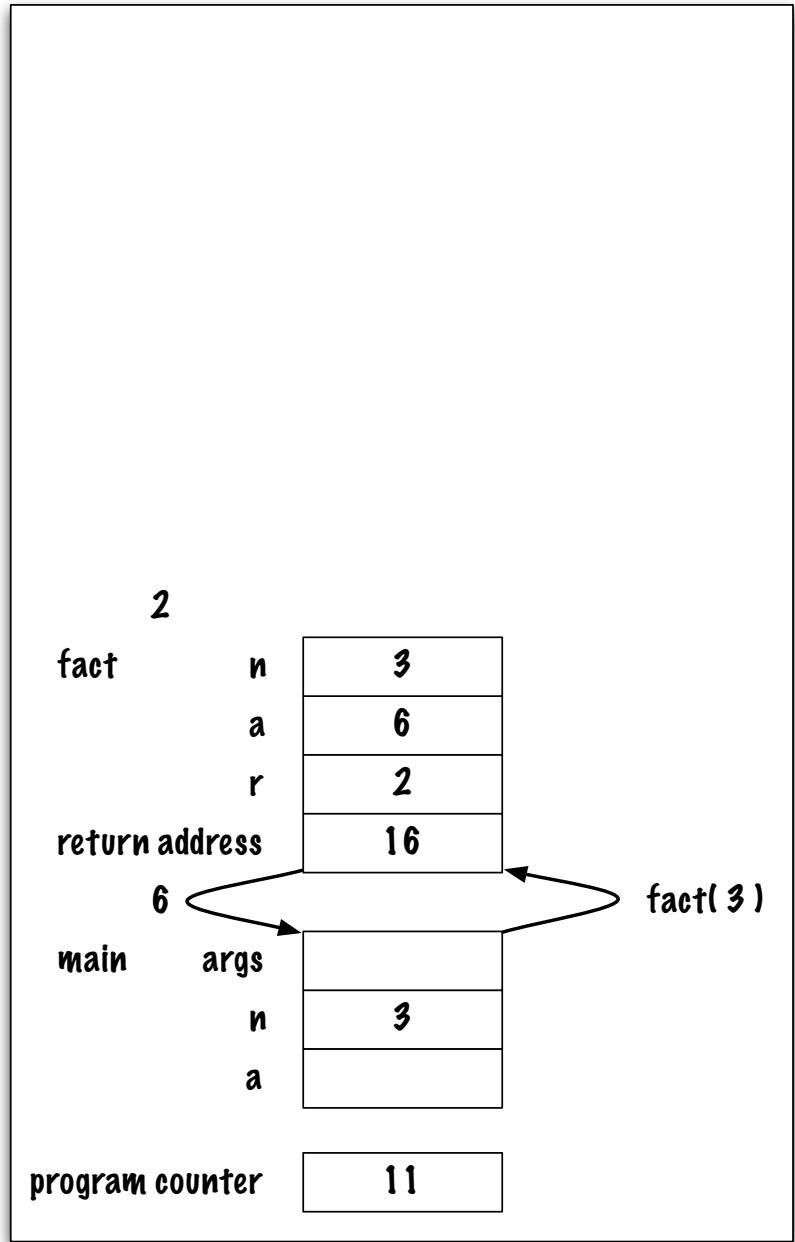


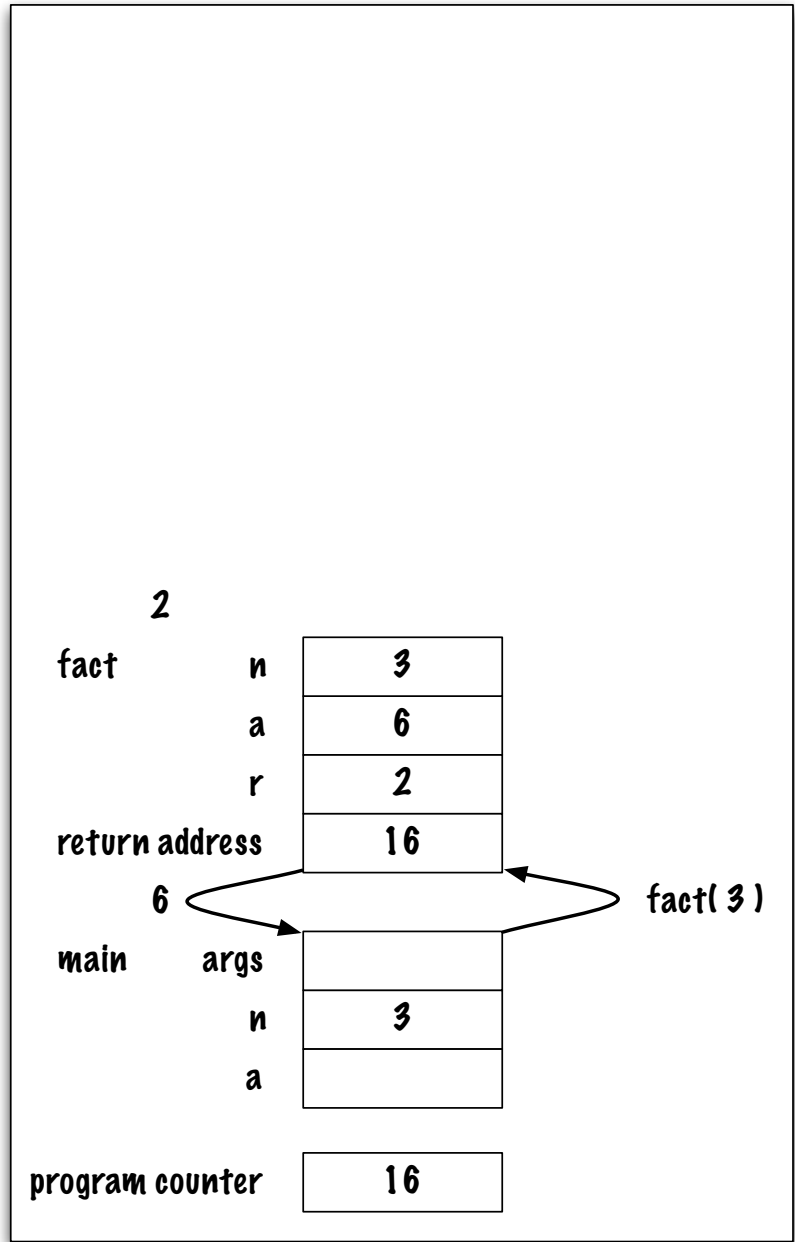












6

main

args

n

a

| |
|---|
| |
| 3 |
| |

program counter

| |
|----|
| 16 |
|----|

6

main

args

n

a

| |
|----------|
| |
| 3 |
| 6 |

program counter

| |
|-----------|
| 17 |
|-----------|