ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 5 février 2011

Résumé

– Type abstrait de données : pile
  – Application des piles

∗. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Évaluation d'expressions arithmétiques

L'utilisation principale des piles dans le cours sera pour l'implémentation d'algorithmes d'**analyse syntaxique** (*parsing*).

Par exemple afin d'évaluer une expression telle que :

$$1 + 2 * 3 - 4$$

De tels algorithmes à base de piles sont utilisés par les compilateurs afin de vérifier la syntaxe de vos programmes (*parsing*) avant de générer les instructions machines (exécutable).

Vérifier que les parenthèses d'une expression sont bien balancées : '([])' est ok, mais pas '([)]' ou ')((())('.

Les 2 premières étapes de l'analyse d'un programme par un compilateur sont l'**analyse lexicale** et l'**analyse syntaxique**.

Lors de l'**analyse lexicale** (*scanning*) le programme source est lu de gauche à droite et les caractères sont regroupés en jetons (*tokens*), qui sont simplement une suite de caractères, qui par exemple forment un nombre ou un identificateur de variable. Normalement, l'analyse lexicale élimine les caractères blancs (espaces) de l'entrée. Exemple : l'expression

$$\cdot 10 \cdot + \cdot \cdot 2 + \cdot \cdot \cdot 300$$

où $\ll\cdot\gg$ représente un caractère blanc, est transformée en une liste de jetons,

$$[10,+,2,+,300]$$

L'étape suivante est l'**analyse syntaxique** (*parsing*) et consiste à regrouper les jetons en unités grammaticales, par exemple les sous expressions des expressions RPNs (vues en classe cette semaine).

```
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 * 5   " );
    }
}
// > java Test
// INTEGER: 3
// SYMBOL: +
// INTEGER: 4
// SYMBOL: *
// INTEGER: 5
```

```
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; }
}
```

Reader

```
import java.util.StringTokenizer;

public class Reader {

    private StringTokenizer st;

    public Reader( String s ) {
        st = new StringTokenizer( s );
    }

    public boolean hasMoreTokens() {
        return st.hasMoreTokens();
    }

    // ...
```

Reader (suite)

```
    // ...

    public Token nextToken() {
        String t = st.nextToken();
        if ( "true".equals( t ) ) {
            return new Token( true );
        }
        if ( "false".equals( t ) ) {
            return new Token( false );
        }
        try {
            return new Token( Integer.parseInt( t ) );
        } catch ( NumberFormatException e ) {
            return new Token( t );
        }
    }
}
```

Algorithme : 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() ) {
        t = reader.nextToken();
        if ( t.isSymbol() ) {
            op = t; // opérateur
        } else {
            r = t.iValue(); // opérande
            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;
}
```

Évaluation d'expressions arithmétiques

Algorithme LR Scan :

```
Declare L, R, OP

Read L
While not end-of-expression
do:
  Read OP
  Read R
  Evaluate L OP R
  Store result in L
```

À la fin de la boucle, le résultat final se trouve dans L.

```
Declare L, R, OP

Read L
While not end-of-expression
do:
  Read OP
  Read R
  Evaluate L OP R
  Store result 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
>
```

$\Rightarrow$ Fin de l'expression et sortie de boucle, L contient le résultat.

Qu'en pensez-vous ?

Sans les **parenthèses** on ne peut évaluer correctement l'expression :

$\Rightarrow$ 7 - (3 - 2)

Puisque le résultat de l'analyse de gauche-à-droite correspond à l'expression :

$\Rightarrow$ (7 - 3) - 2

De façon similaire, mais pour une raison différente, on ne peut évaluer correctement l'expression :

$\Rightarrow$ 7 - 3 * 2

Puisque l'analyse de gauche-à-droite correspond à évaluer :

$\Rightarrow$ (7 - 3) * 2

Alors que la **priorité des opérateurs** nous dit qu'il faut évaluer :

$\Rightarrow$ 7 - (3 * 2)

Remarques

L'algorithme de gauche-à-droite :

– Ne traite pas les parenthèses ;
– Ne traite pas la priorité des opérateurs.

Solutions :

1. Utiliser une notation différente ;
2. Développer un algorithme plus complexe.

$\Rightarrow$ Les 2 utilisations nécessitent l'utilisation d'une pile.

Notations

Il y a trois façons de représenter l'expression : **L OP R**.

infixe : La notation infixe correspond à la notation habituelle, l'opérateur est mis en sandwich entre ses opérandes : L OP R.

postfixe : En notation postfixe, les opérandes sont placés devant l'opérateur, L R OP. On appel aussi cette notation *Reverse Polish Notation* ou **RPN**, c'est la notation utilisée par certaines calculettes scientifiques (telles que HP-35 de Hewlett-Packard ou Texas Instruments TI-89 à l'aide de RPN Interface par Lars Frederiksen [1]) et le langage PostScript.

```
 7  - (3  -  2)  =  7 3 2 - -
(7  -  3) -  2   =  7 3 - 2 -
```

préfixe : La troisième notation consiste à placer l'opérateur d'abord suivi de ses opérandes, OP L R. Le langage de programmation Lisp utilise une combinaison de parenthèses et de notation préfixe, `(- 7 (* 3 2))`.

$\Rightarrow$ L et R peuvent-elles même être des sous expressions complexes.

1. www.calculator.org/rpn.html

Infixe $\rightarrow$ postfixe (mentalement)

Transformez successivement, une à une, chaque sous expression en suivant l'ordre normal d'évaluation d'une expression infixe.

Une sous expression infixe $l \diamond r$ devient $l\ r \diamond$ (ou l et r sont elles même des sous expressions et $\diamond$ est un opérateur).

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

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

$$9\ /\ (\ \overbrace{2}^{l}\ \overbrace{4}^{r}\ \overbrace{\times}^{\diamond}\ -\ 5\ )$$

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

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

$$9\ /\ [\ \overbrace{[\ 2\ \ 4\ \times\ ]}^{l}\ \overbrace{5}^{r}\ \overbrace{-}^{\diamond}\ ]$$

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

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

$$\overbrace{9}^{l}\ \overbrace{[\ 2\ \ 4\ \times\ 5\ -\ ]}^{r}\ \overbrace{/}^{\diamond}$$

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

Évaluer une expression postfixe (mentalement)

Parcourir l'expression de gauche à droite. Lorsque l'élément courant est un opérateur, appliquer le à ses deux opérandes, c'est-à-dire, remplacez $l\ r\ \diamond$ par le résultat de l'évaluation de l'expression $l\ \diamond\ r$.

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

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

$$9\ \overbrace{8}^{l \diamond r}\ 5\ -\ /$$

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

$$9\ \overbrace{3}^{l\ \diamond\ r}\ /$$

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


l ◇ r
3

Expressions postfixe (mentalement)

Jusqu'à ce que la fin de l'expression soit atteinte :

1. Lire de gauche à droite jusqu'au premier opérateur ;
2. Appliquer l'opérateur aux 2 opérandes qui le précèdent ;
3. Remplacer l'opérateur et ses opérandes par le résultat.

Lorsque la fin de l'expression est atteinte, nous avons le résultat.

9 3 / 10 2 3 * - +

```
9 2 4 * 5 - /
```

Remarques

L'ordre des opérandes est le même pour les deux notations, postfixe et infixe, cependant les endroits où sont insérés les opérateurs diffèrent.

```
2 + (3 * 4) = 2 3 4 * +

(2 + 3) * 4  = 2 3 + 4 *
```

Pour évaluer une expression infixe, il faut tenir compte de la précédence des opérateurs ainsi que des parenthèses. Dans le cas de la notation postfixe, ces concepts sont représentés à même la notation.

Algorithme : Eval Infix

Quel sera le rôle de la pile ?

```
operands = ≪new stack≫;

while ( ≪plus de jetons≫ ) {
  t = ≪prochain≫;
  if ( ≪t est un opérande≫ ) {
    operands.push( ≪valeur entière de t≫ );
  } else { // c'est un opérateur
    op = ≪opérateur t≫;
    r = operands.pop();
    l = operands.pop();
    operands.push( ≪eval( l, op, r )≫ );
  }
}
return operands.pop();
```

Évaluer une expression postfixe

L'algorithme nécessite une pile (Numbers), une variable qui contient le dernier «élément» lu (X), ainsi que L et R (dont la fonction est la même que tout à l'heure).

```
Numbers = [

While not end-of-expression
do:
  Read X
  If X isNumber, PUSH X onto Numbers
  If X isOperator,
    R = POP Numbers (droite avant gauche?)
    L = POP Numbers
    Evaluate L X R; PUSH result onto Numbers
```

Pour obtenir le résultat final : POP Numbers.

9 3 - 2 /

9 3 / 10 2 3 * - +

```
9 / ((2 * 4) - 5)  =  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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Initialiser une nouvelle pile vide.

```
9 / ((2 * 4) - 5)  =  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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire X.

```
9 / ((2 * 4) - 5)  =  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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

⇒ Empile X.

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

   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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

⇒ Empiler 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ L'élément du dessus est retiré et mis dans 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 (droite avant gauche?)
>      L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ L'élément du dessus est retiré et mis dans L.

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

   Numbers = [9 8
   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 (droite avant gauche?)
       L = POP Numbers
>      Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Empiler le résultat de L X R, $2 \times 4 = 8$, sur la pile.

```
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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Empiler 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Retirer l'élément du dessus de la pile et le mettre dans 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 (droite avant gauche?)
>      L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Retirer l'élément du dessus de la pile et le mettre dans 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 (droite avant gauche?)
       L = POP Numbers
>      Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Empiler le résultat de L X R, $8 - 5 = 3$, sur la pile.

```
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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Lire 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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ 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 (droite avant gauche?)
>      L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ 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 (droite avant gauche?)
       L = POP Numbers
>      Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Évaluer L X R, $9 \div 3 = 3$, empiler le résultat 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 (droite avant gauche?)
       L = POP Numbers
>      Evaluate L X R; PUSH result onto Numbers
```

$\Rightarrow$ Fin de l'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 (droite avant gauche?)
       L = POP Numbers
       Evaluate L X R; PUSH result onto Numbers
>
```

$\Rightarrow$ le résultat est POP Numbers $= 3$; et la pile est maintenant vide.

Problème

Plutôt que d'**évaluer** une expression RPN, on veut maintenant convertir une expression RPN en notation infixe (la notation usuelle).

Qu'est-ce qu'il faut faire ? A-t-on besoin d'un nouvel algorithme ?

Non, la modification est toute simple, il suffit de remplacer l'énoncé ≪Evaluate L OP R≫ par ≪Concatenate (L OP R), c'est-à-dire la représentation infixe de ≪L R OP≫.

Notez que les parenthèses sont essentielles (pas toutes, mais certaines le sont).

Cette fois-ci la pile ne contient plus des nombres, mais des chaînes de caractères qui représentent des nombres et des sous expressions.

9 5 6 3 / - /

Postfixe → infixe

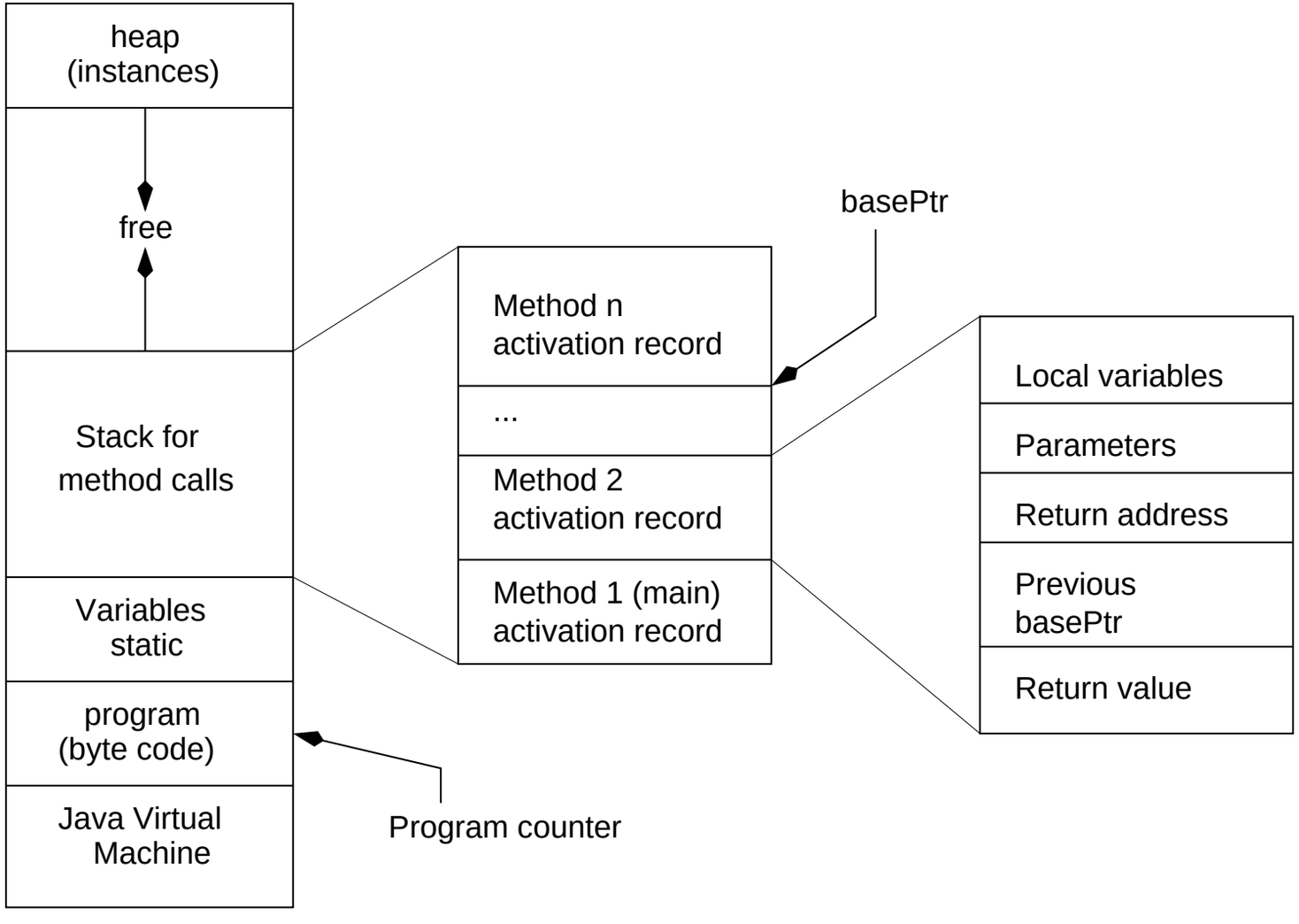
```
Numbers = [
X =
L =
R =

While not end-of-expression
do:
  Read X
  If X isNumber, PUSH X onto Numbers
  If X isOperator,
    R = POP Numbers (droite avant gauche?)
    L = POP Numbers
    Concatenate ( L X R ); PUSH result onto Numbers
```

Infixe $\rightarrow$?

```
While not end-of-expression
do:
  Read X
  If X isNumber, PUSH X onto Numbers
  If X isOperator,
    R = POP Numbers (droite avant gauche?)
    L = POP Numbers
    Process L X R; PUSH result onto Numbers
```

Nous avons vu un exemple ou 'Process = Evaluate', puis un autre exemple ou 'Process = Concatenate', Process pourrait aussi produire du code assembleur (i.e. instructions machines).

Cet exemple démontre donc comment les programmes sont compilés ou traduits.



$\Rightarrow$ Vision schématique et simplifiée de l'interprétation d'un programme Java.

À l'appel d'une méthode

La machine virtuelle de Java (JVM) doit :

1. Créer un nouveau bloc d'activation (la valeur de retour, valeur précédente de basePtr et l'adresse de retour ont une taille fixe, la taille des variables locales et des paramètres dépend de la méthode) ;
2. Sauver la valeur courante de basePtr, à l'espace ≪valeur précédente de basePtr≫, faire pointer basePtr à la base du bloc courant ;
3. Sauver la valeur de locationCounter dans l'espace désigner par ≪adresse de retour≫, faire pointer locationCounter vers la première instruction de la méthode appelée ;
4. Copier les valeurs des paramètres effectifs dans région désignée par ≪Paramètre≫ ;
5. Initialiser les variables locales ;
6. Début d'exécution à l'instruction pointée par locationCounter.

À la fin de l'exécution d'une méthode

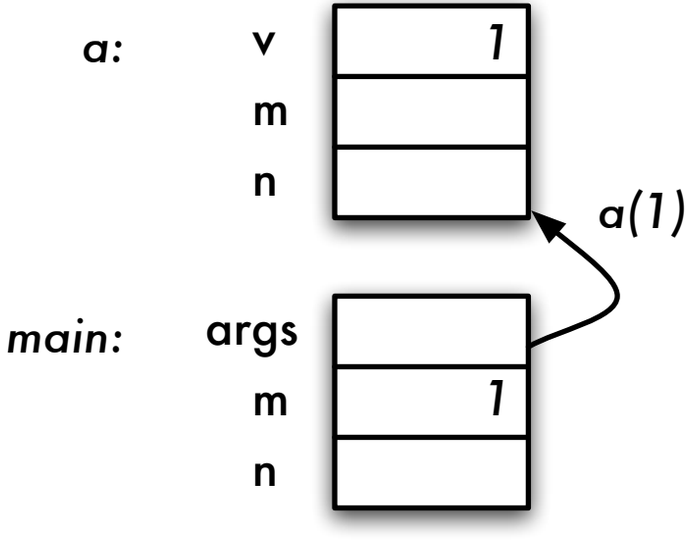
1. La méthode sauve la valeur de retour à l'endroit indiqué par ≪valeur de retour≫ ;
2. Retourne le contrôle à la méthode appelante, i.e. remet en place les valeurs de locationCounter et basePtr ;
3. Retire le bloc d'activation courant ;
4. Reprend l'exécution à l'endroit désigné par locationCounter.

Exemple 1 (simplifié)

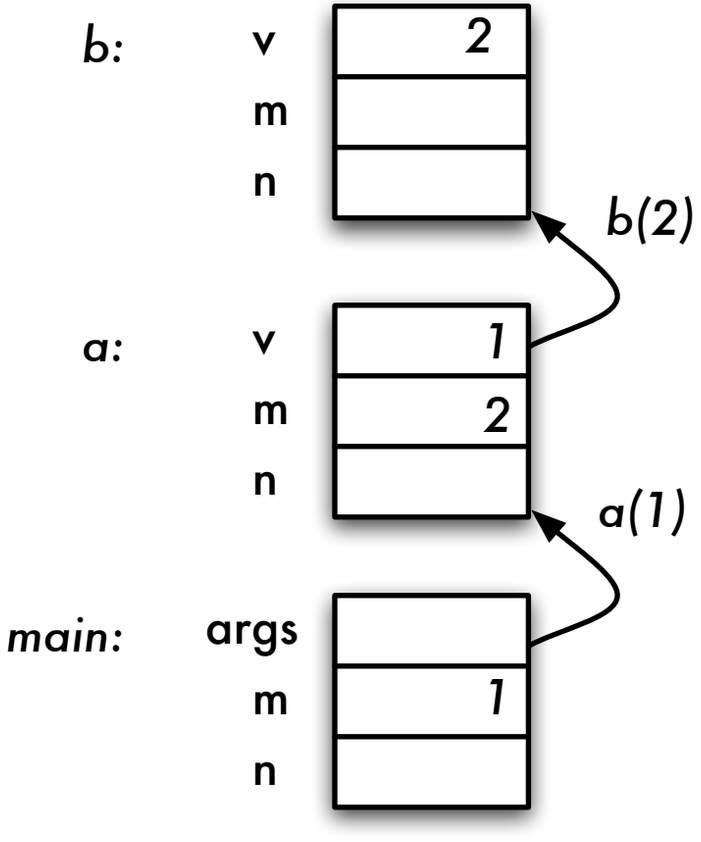
```
public class Calls {
    public static int c( int v ) {
        int n;
        n = v + 1;
        return n;
    }
    public static int b( int v ) {
        int m,n;
        m = v + 1;
        n = c( m );
        return n;
    }
    public static int a( int v ) {
        int m,n;
        m = v +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

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

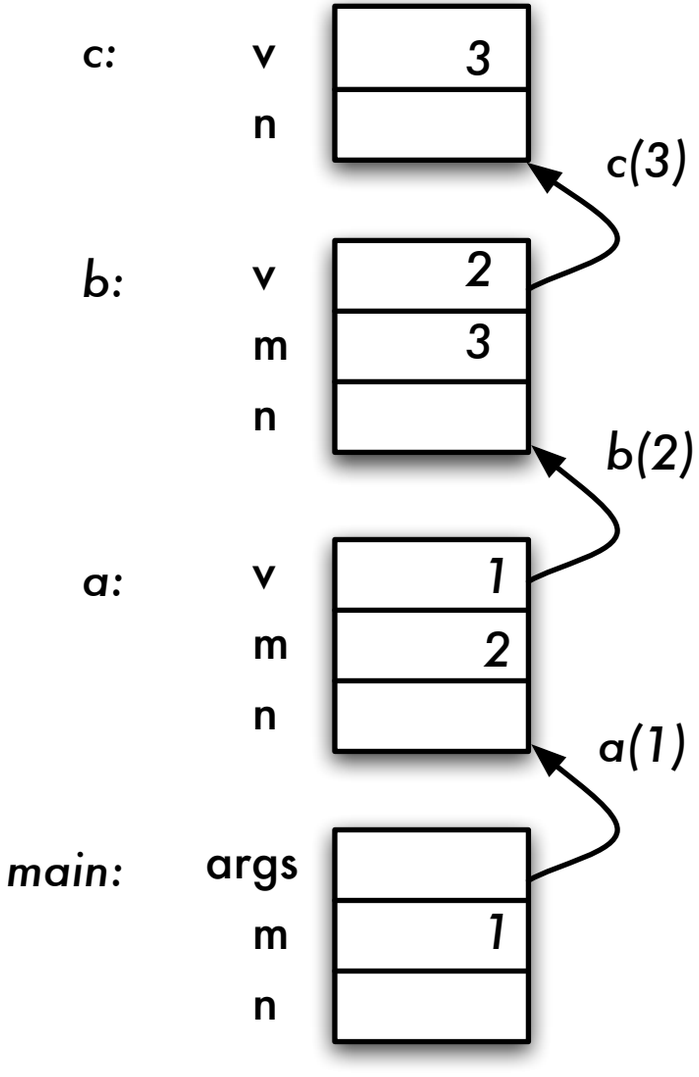
Exemple 1 (simplifié)



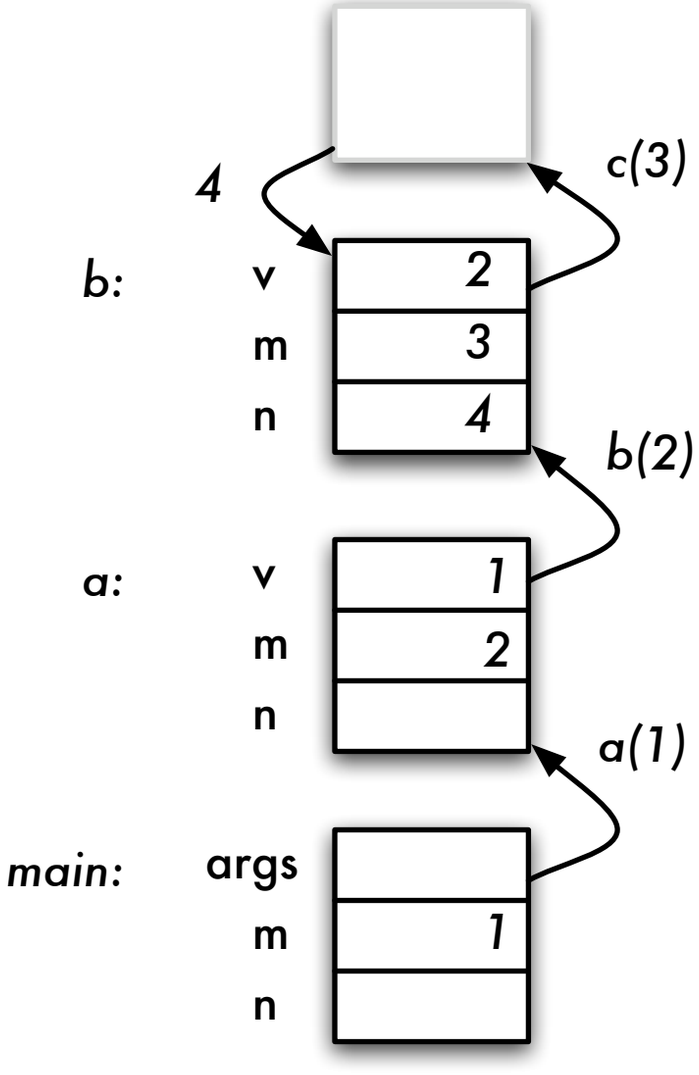
Exemple 1 (simplifié)



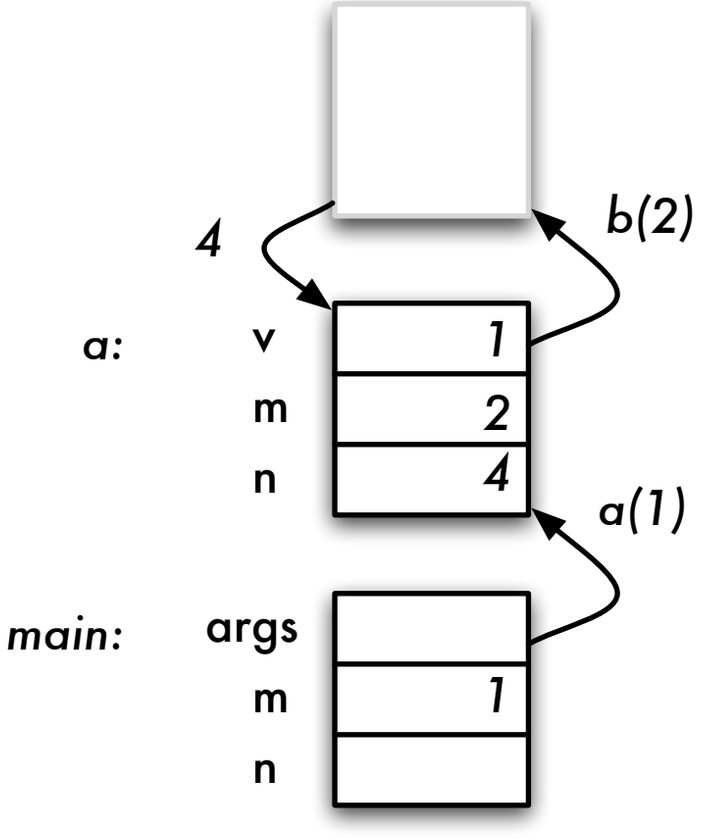
Exemple 1 (simplifié)



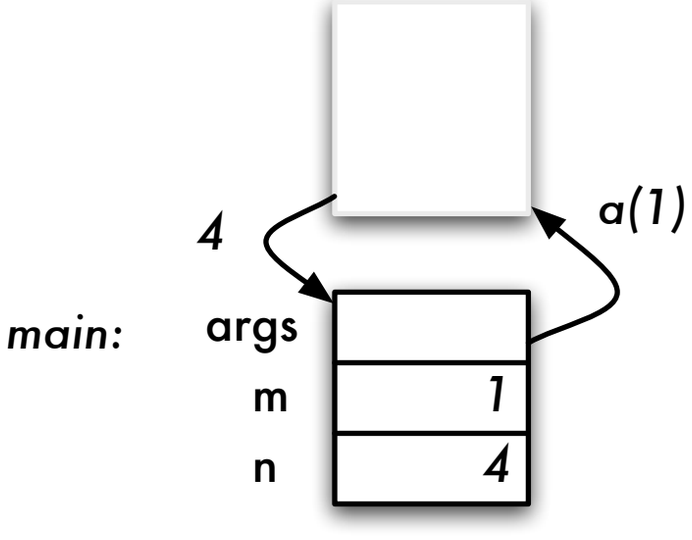
Exemple 1 (simplifié)



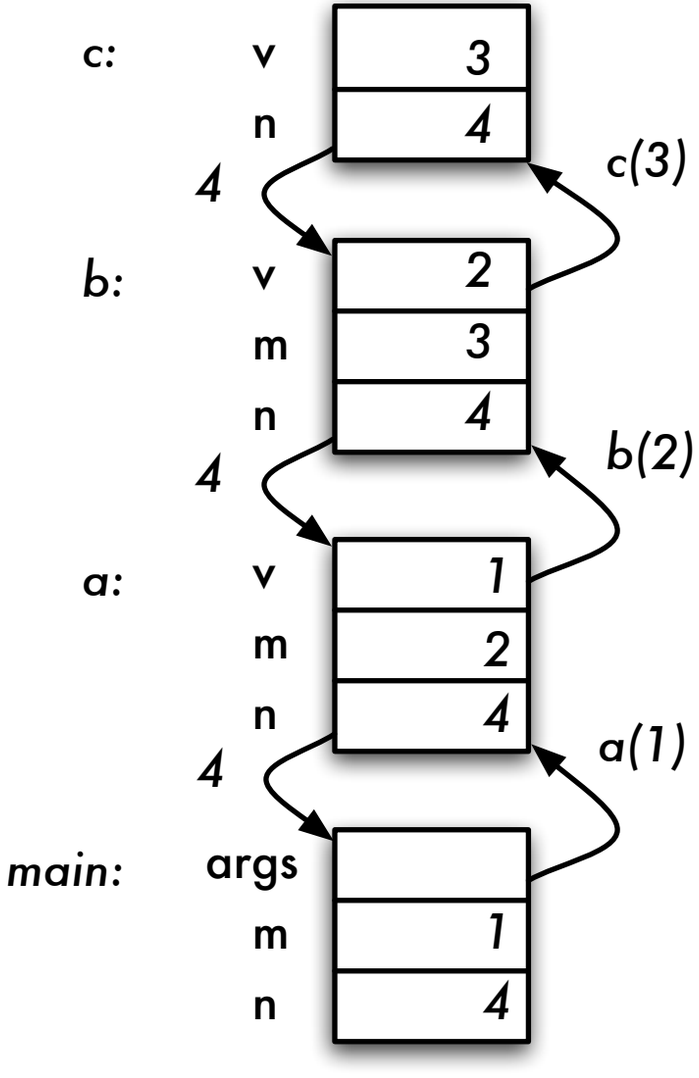
Exemple 1 (simplifié)



Exemple 1 (simplifié)

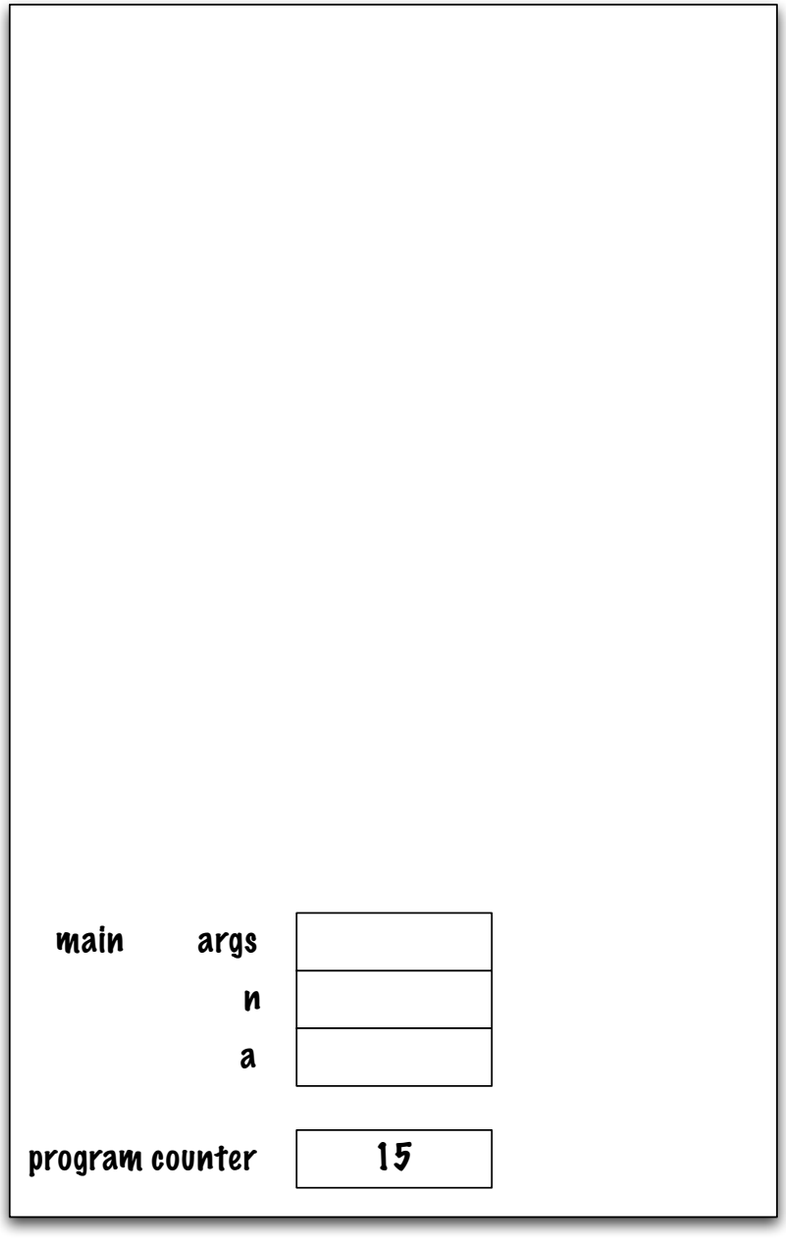


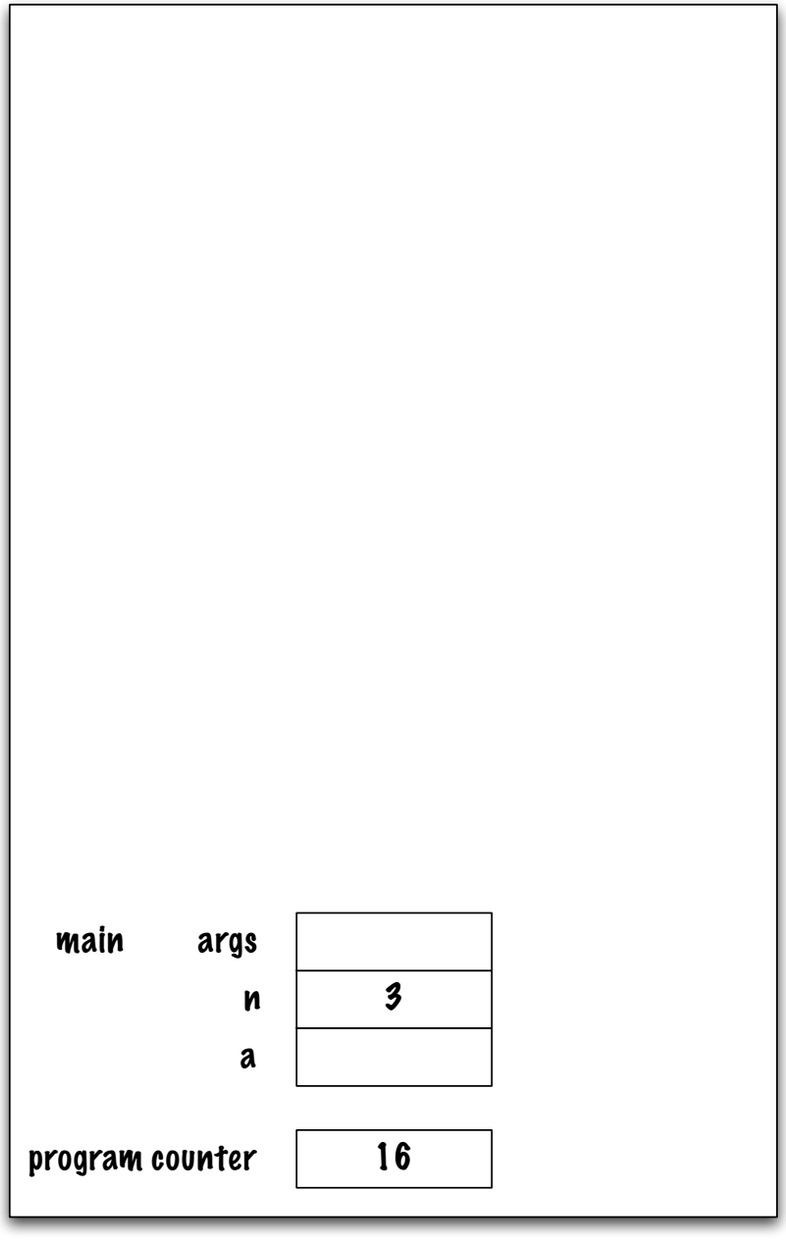
Exemple 1 : résumé

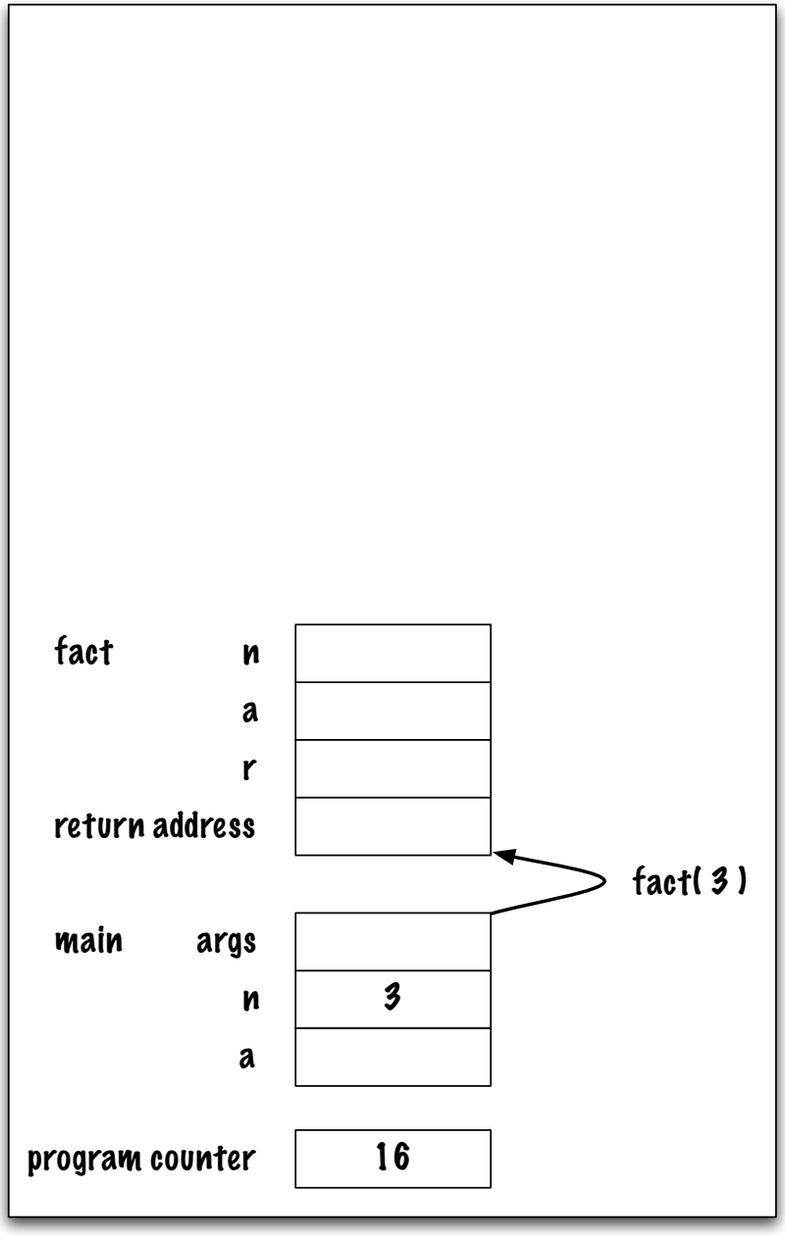


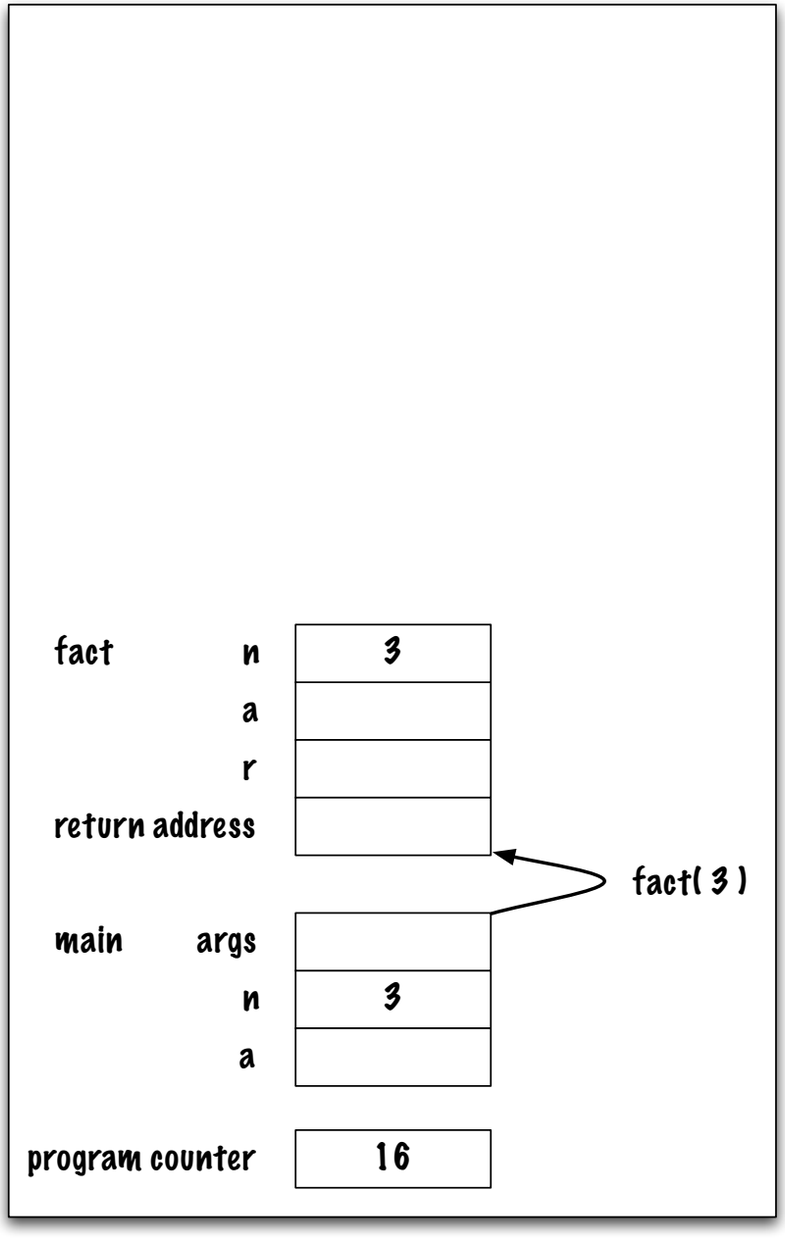
Exemple 2 (avec compteur de programme)

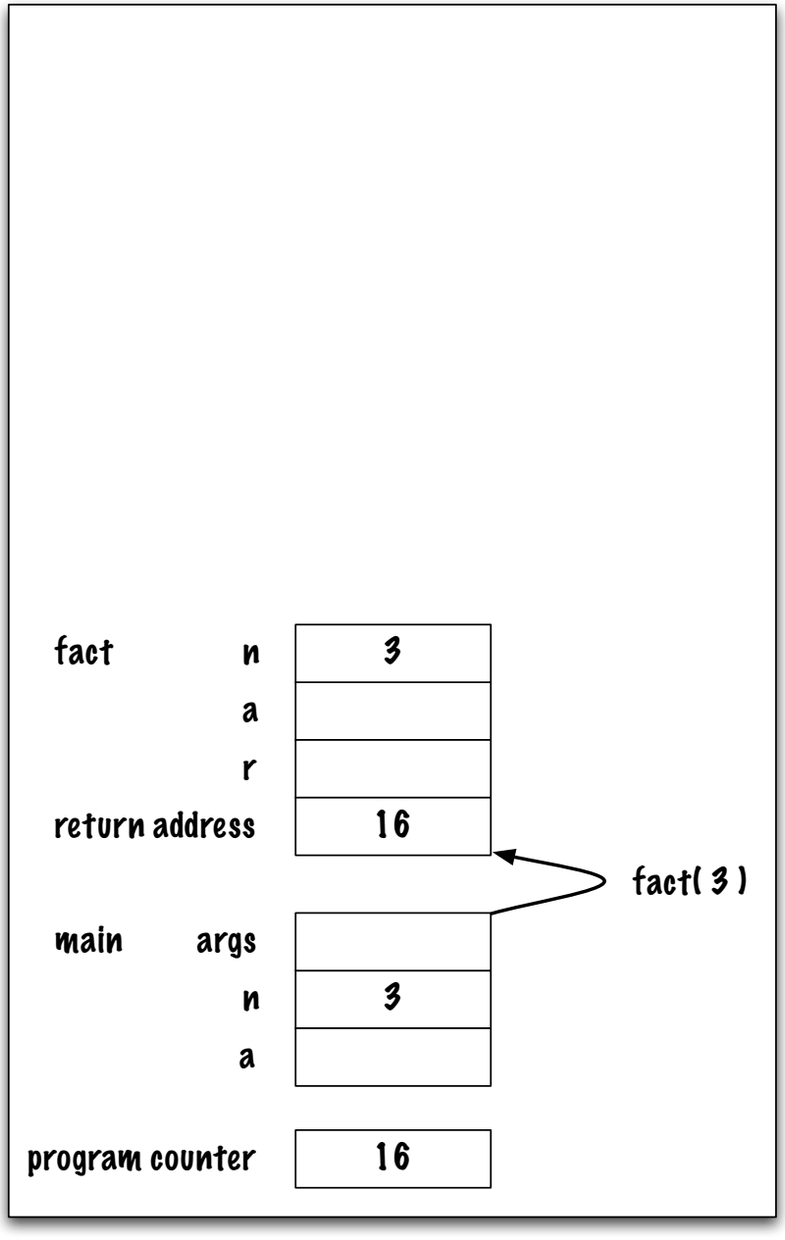
```
01 public class Fact {
02     public static int fact( int n ) {
03      // pre-condition: n >= 0
04      int a, r;
05      if ( n == 0 || n == 1 ) {
06          a = 1;
07      } else {
08          r = fact( n-1 );
09          a = n * r;
10      }
11      return a;
12     }
13     public static void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

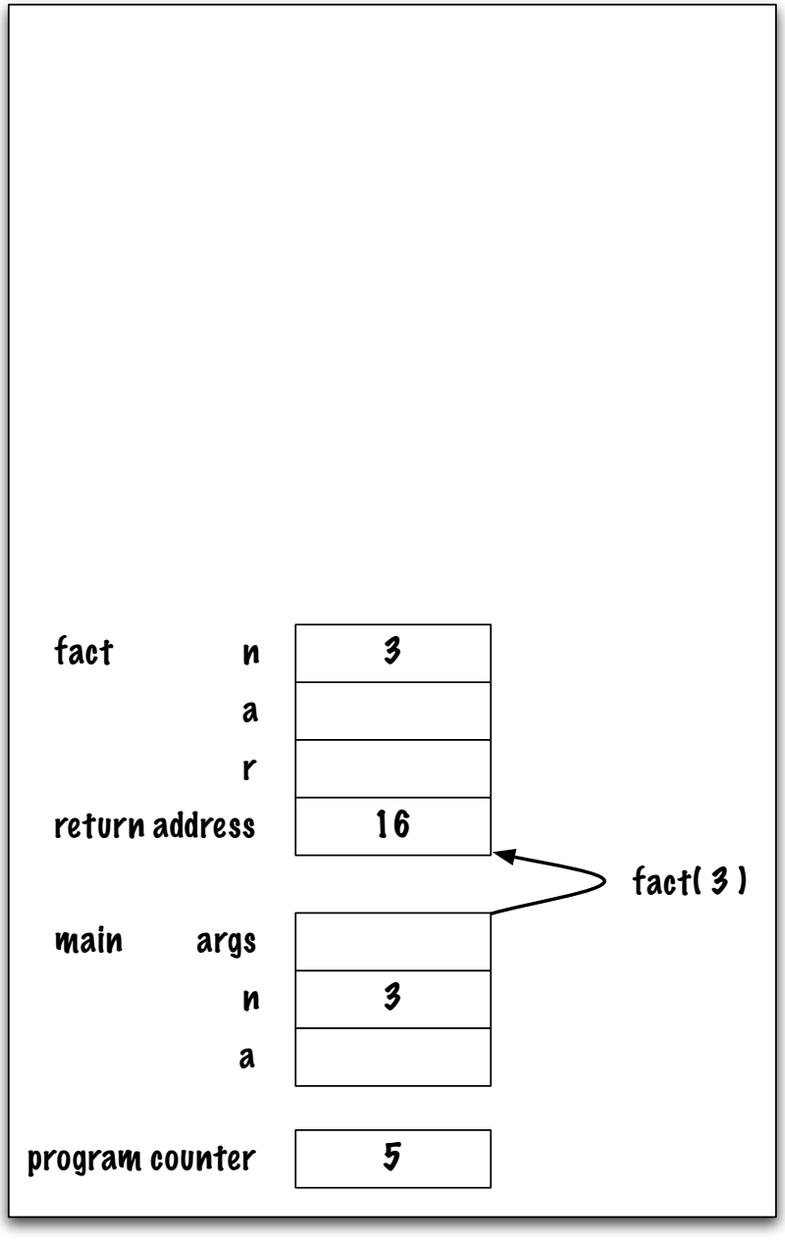

main
args
n
a
program counter
15

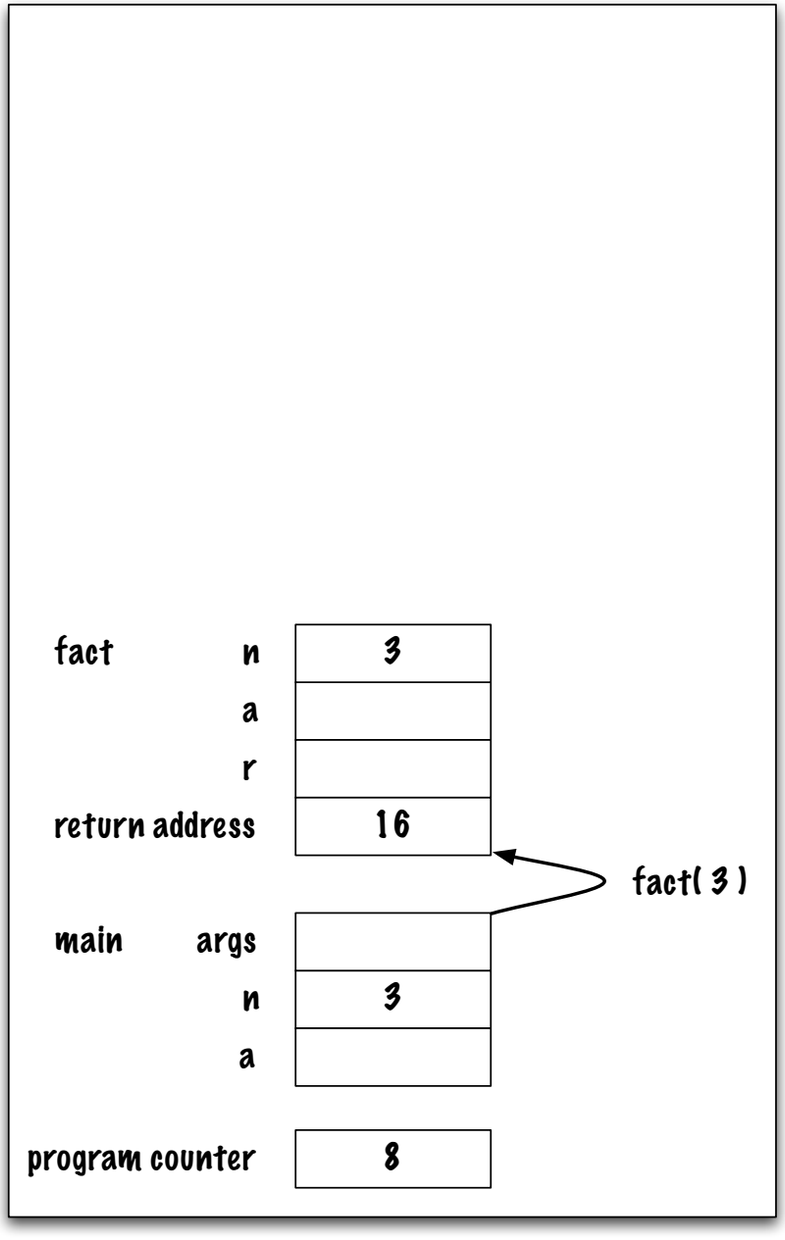

main
args
n
3
a
program counter
16

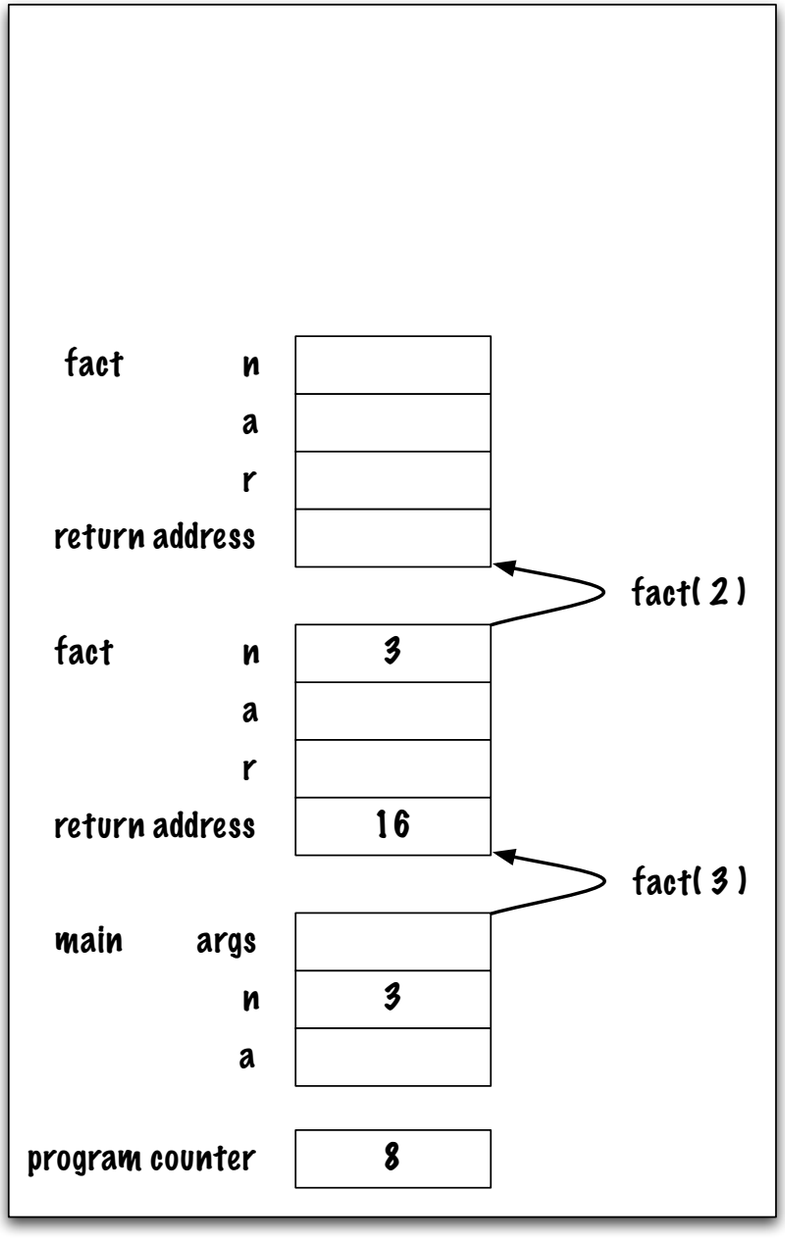

fact
n
a
r
return address
fact(3)
main
args
n
3
a
program counter
16

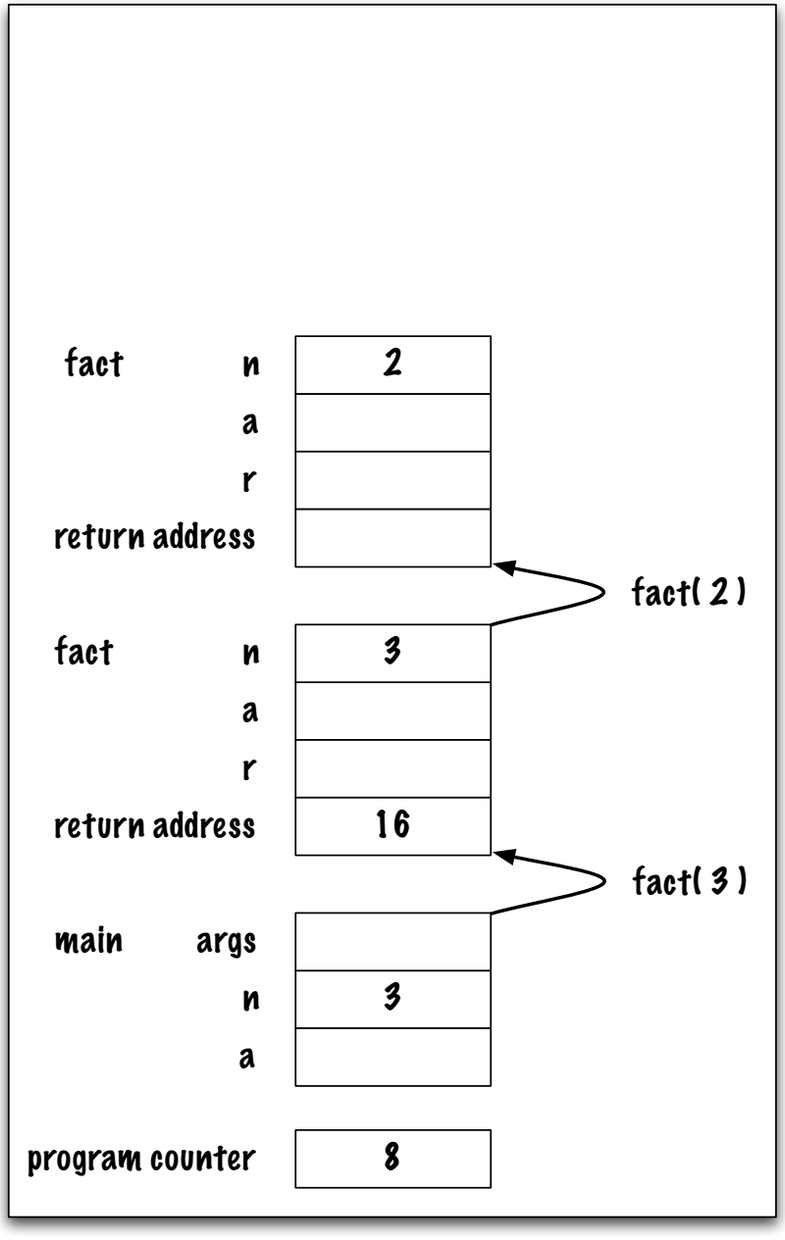

fact
n
3
a
r
return address
fact(3)
main
args
n
3
a
program counter
16

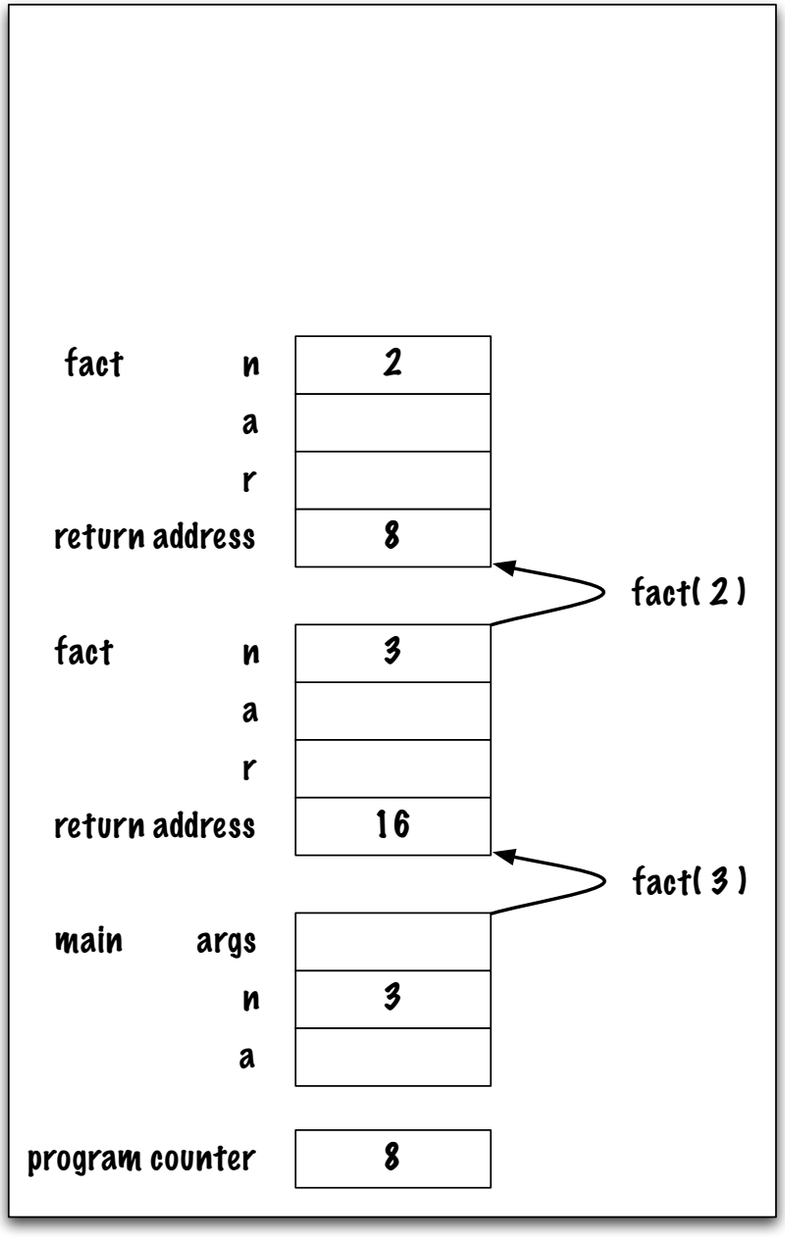

fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
16

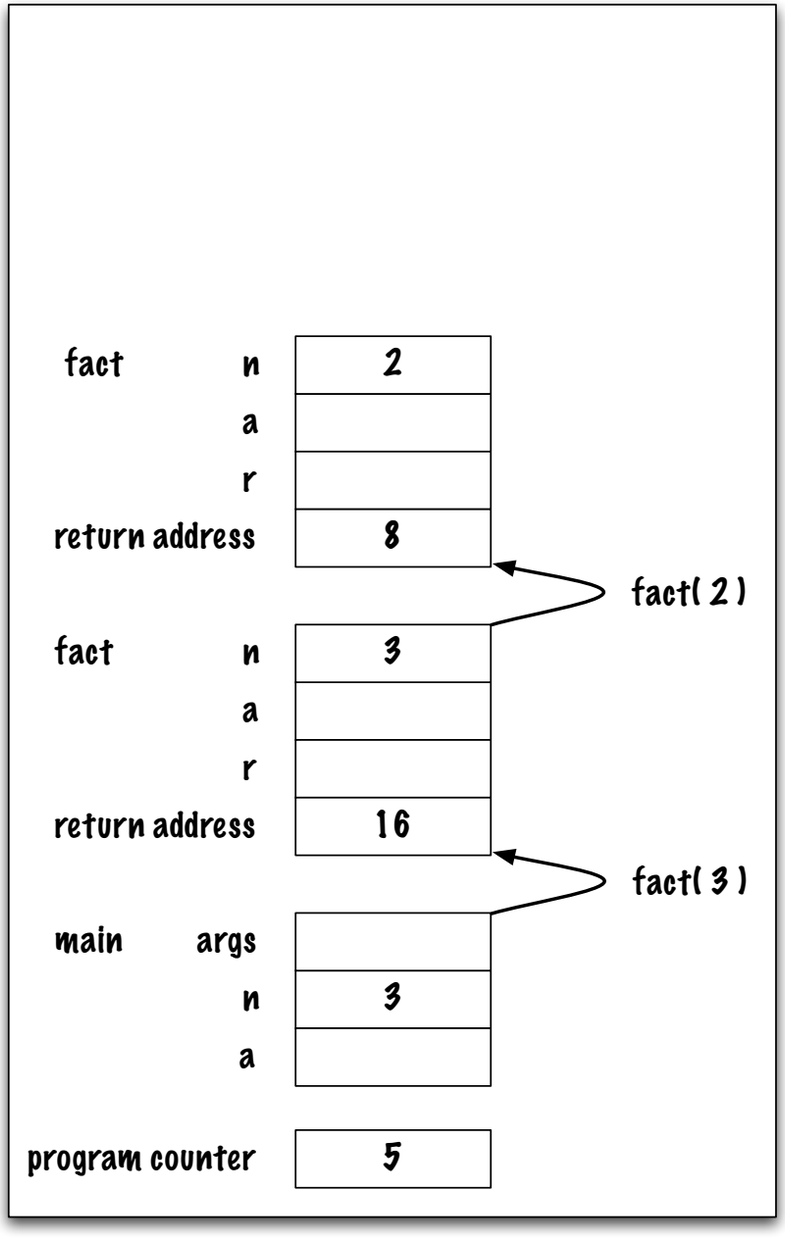

fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
5

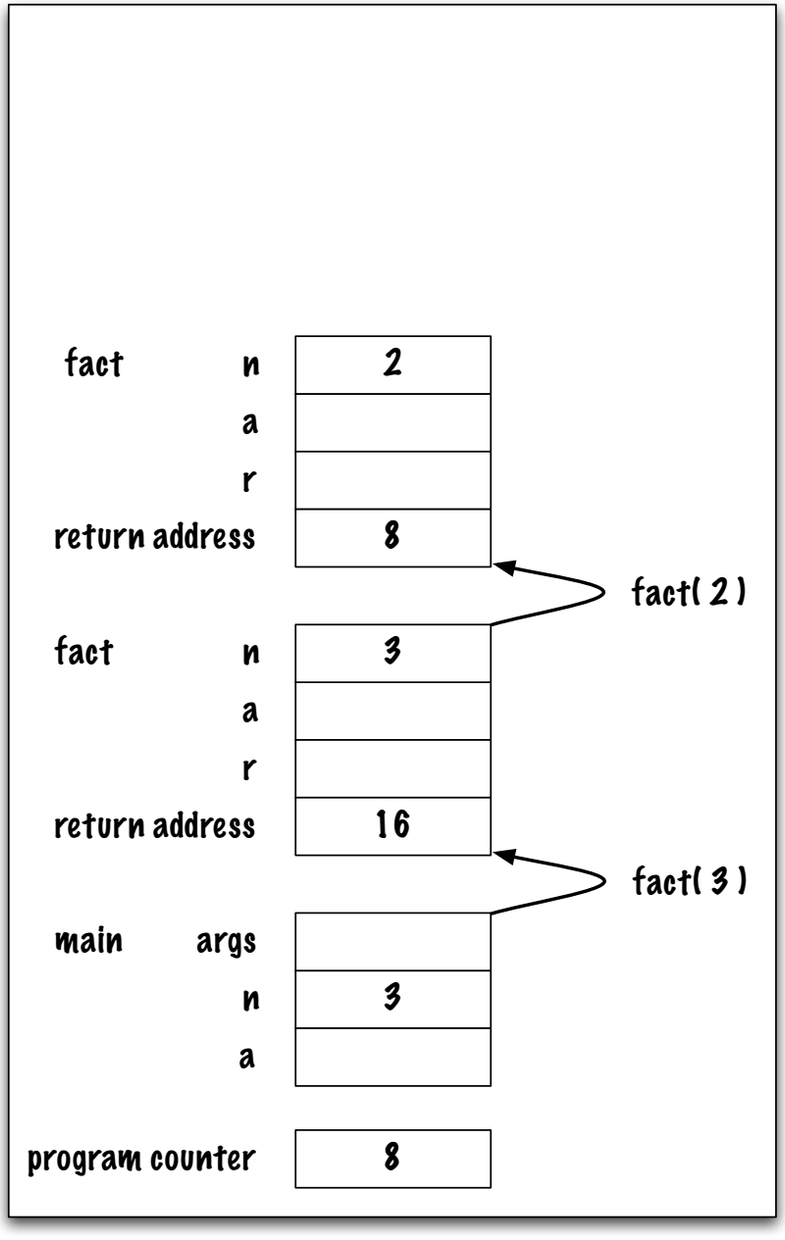

fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

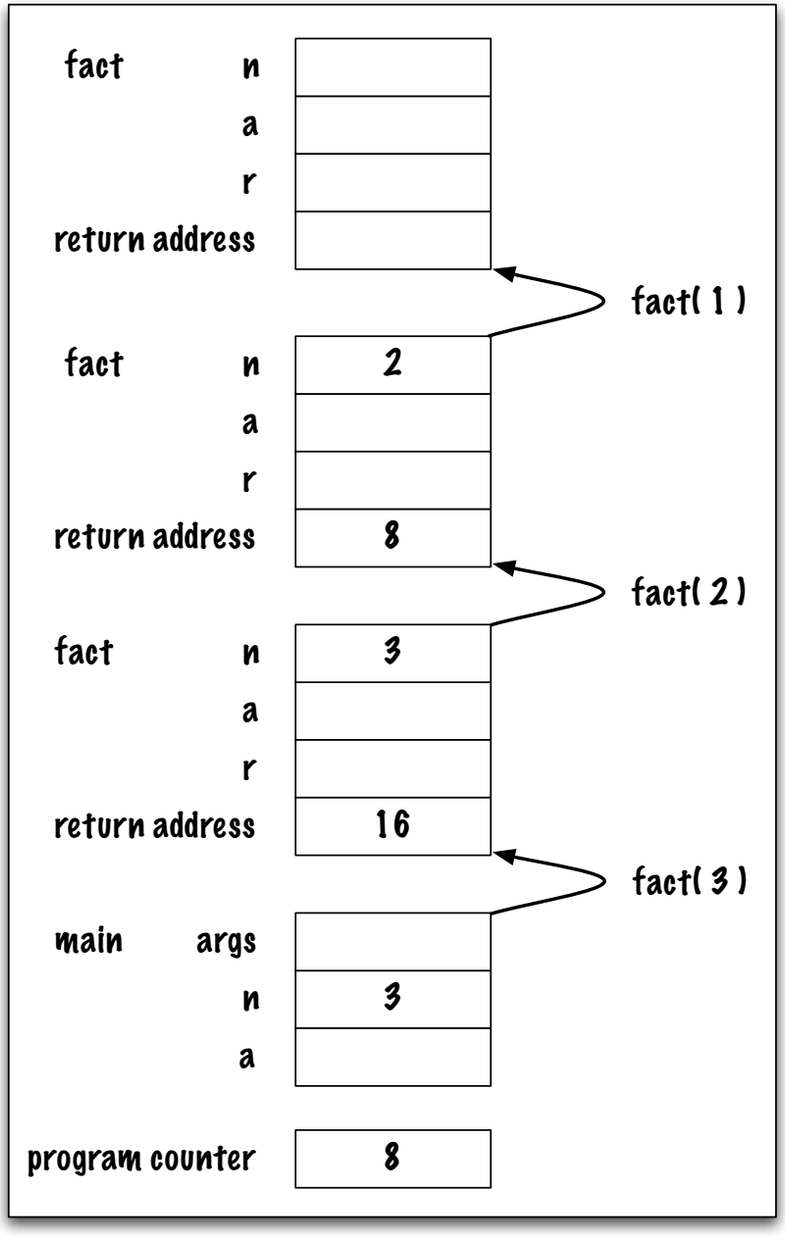

fact
n
a
r
return address
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

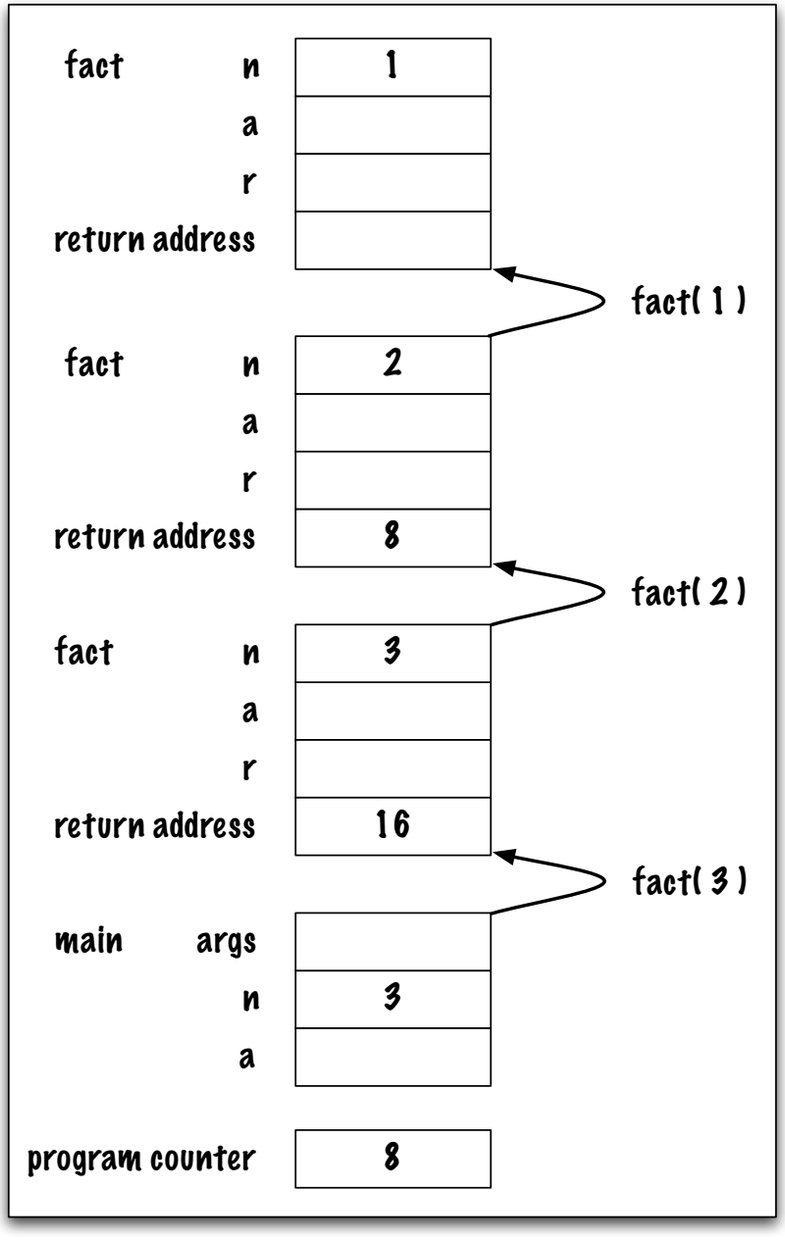

fact
n
2
a
r
return address
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

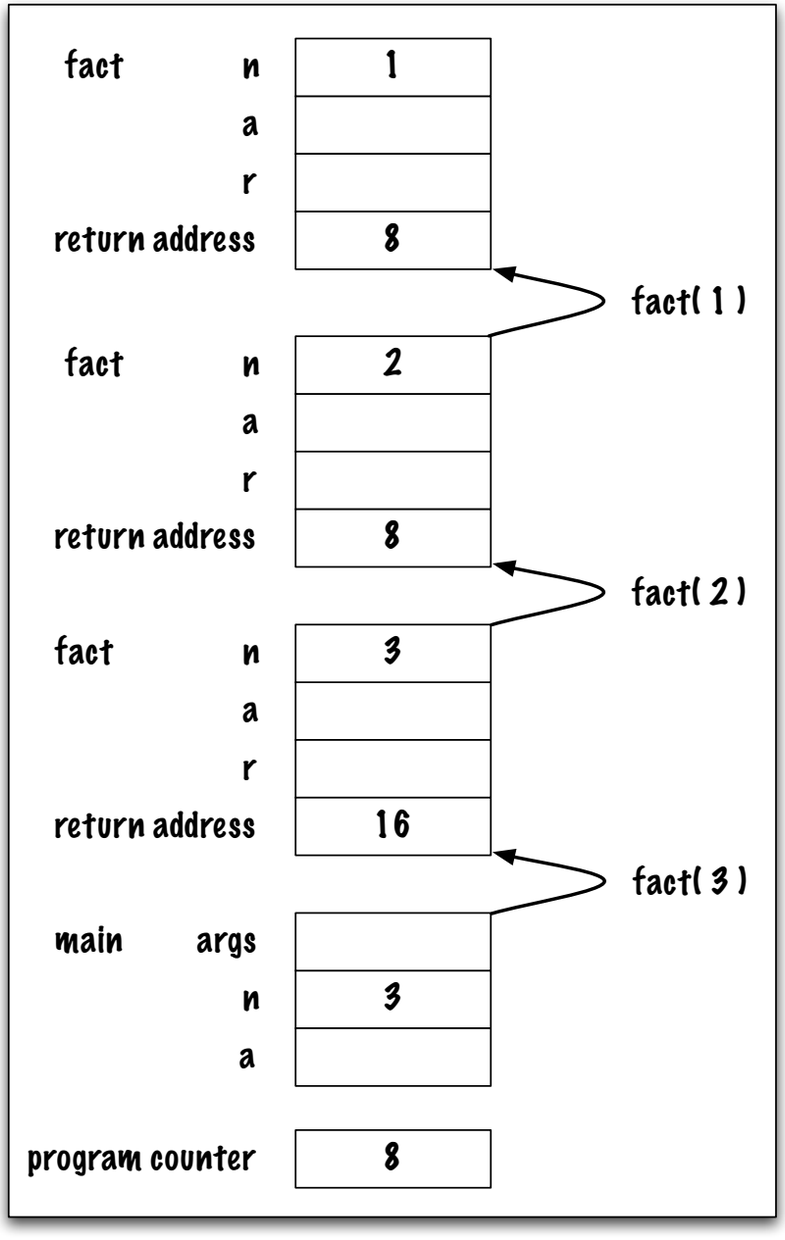

fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

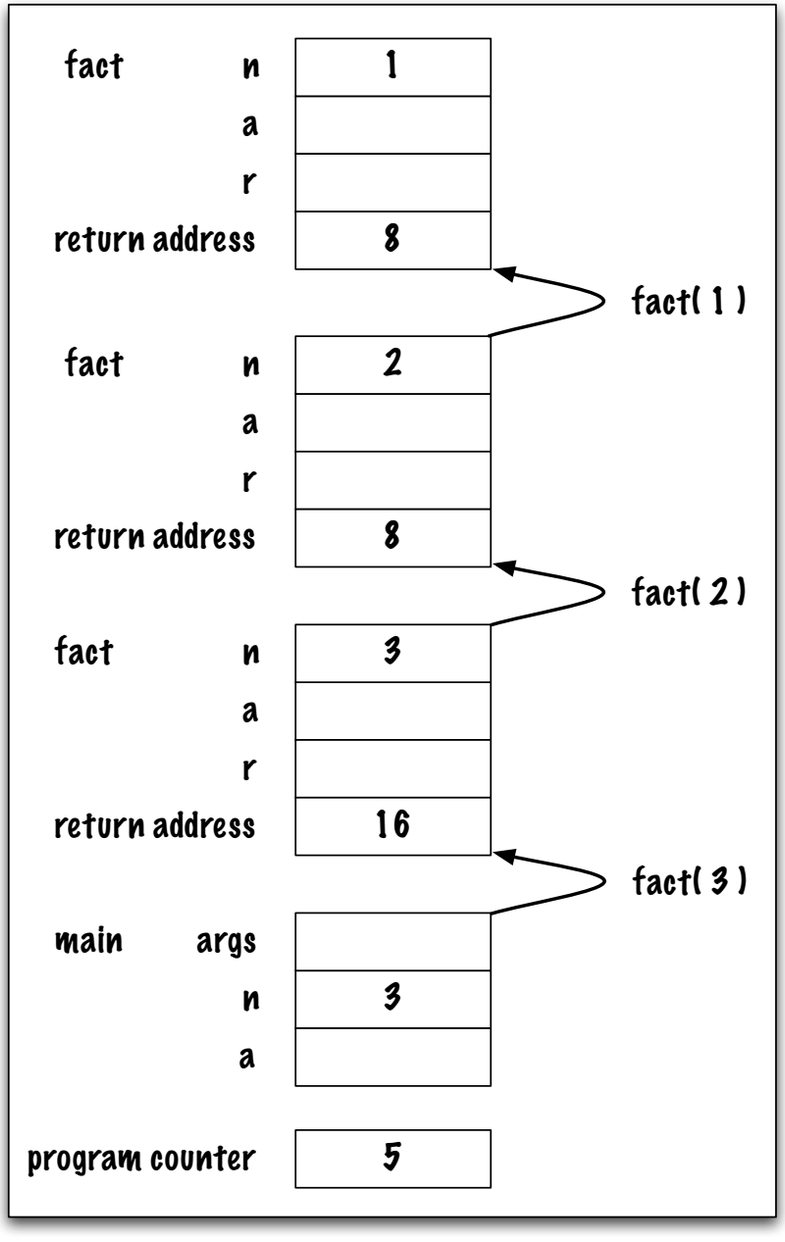

fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
5

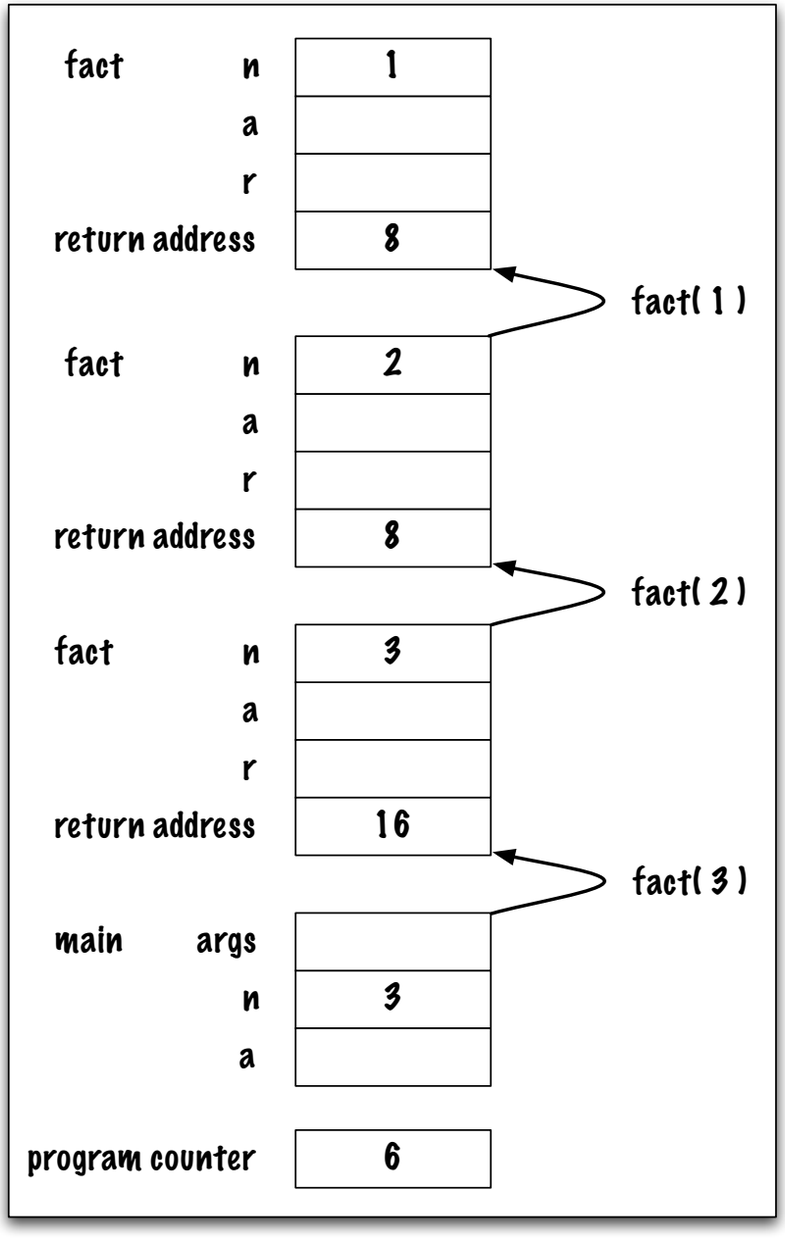

fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

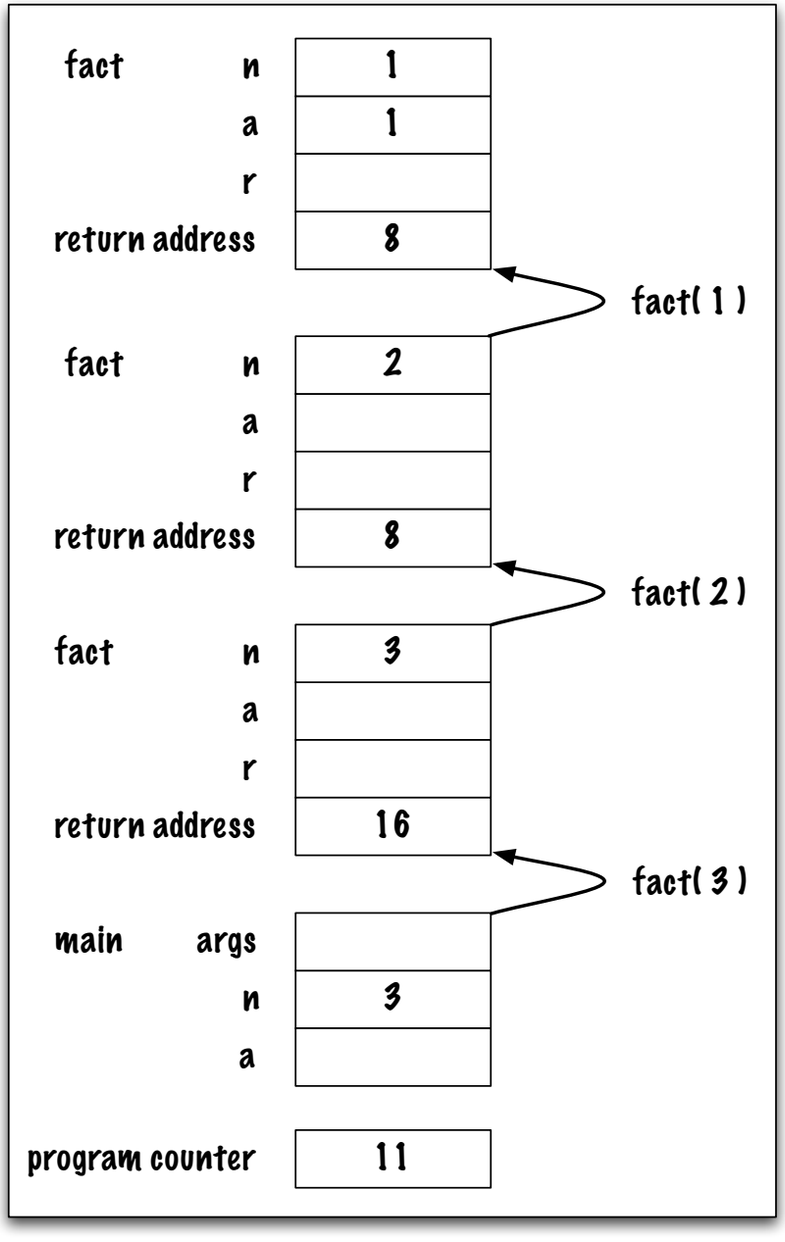

fact
n
a
r
return address
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

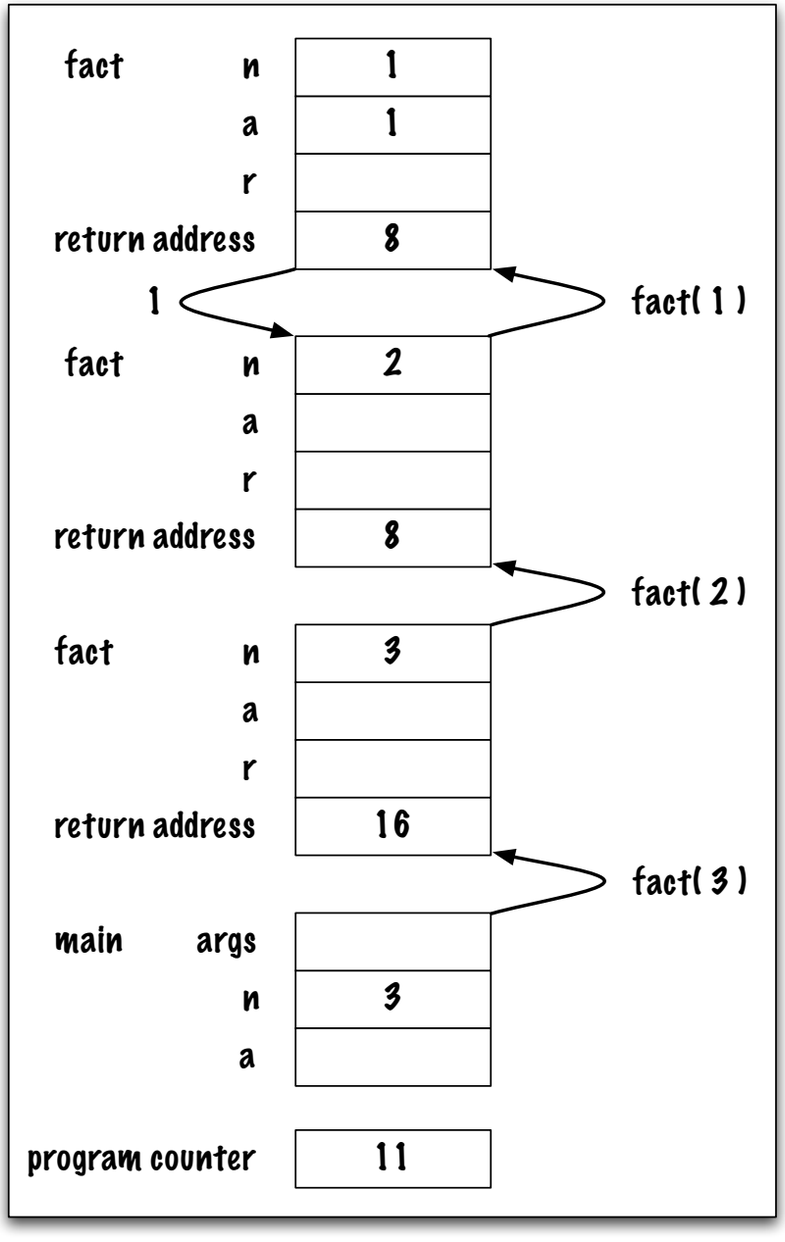

fact
n
1
a
r
return address
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

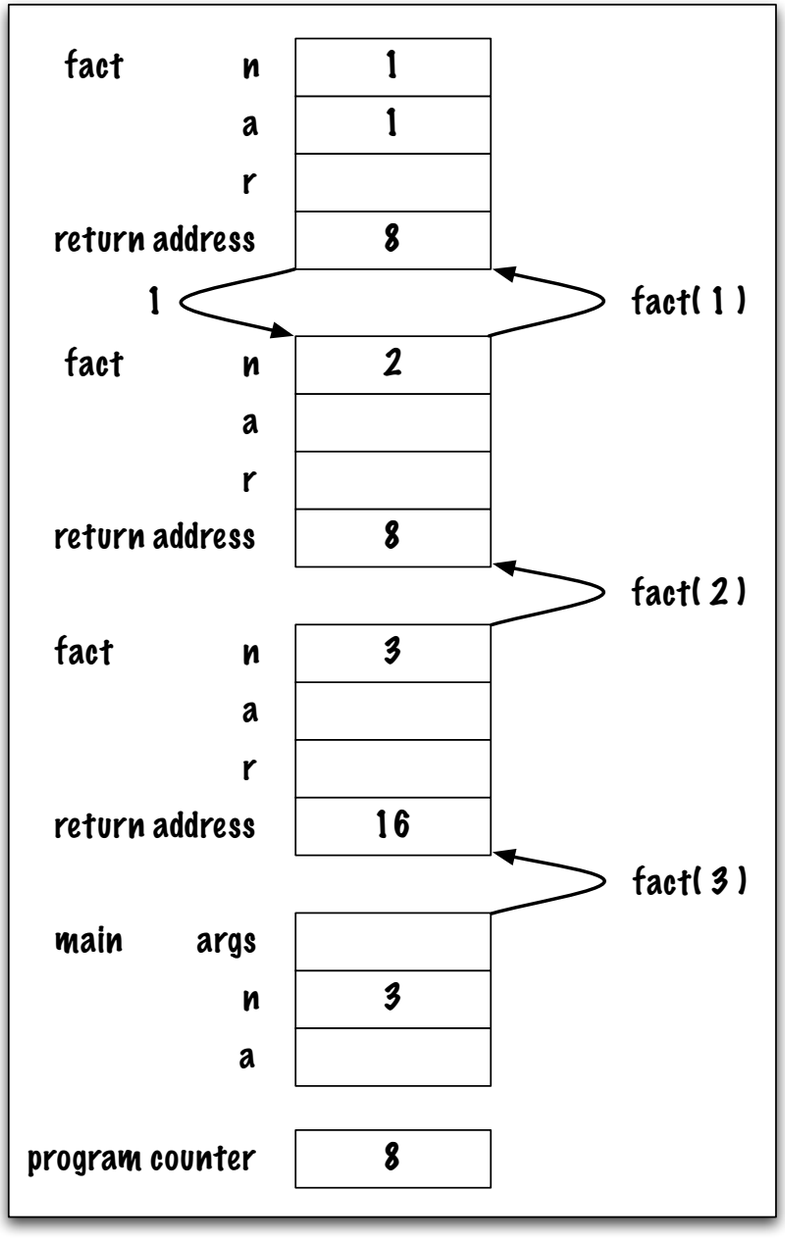

fact
n
1
a
r
return address
8
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

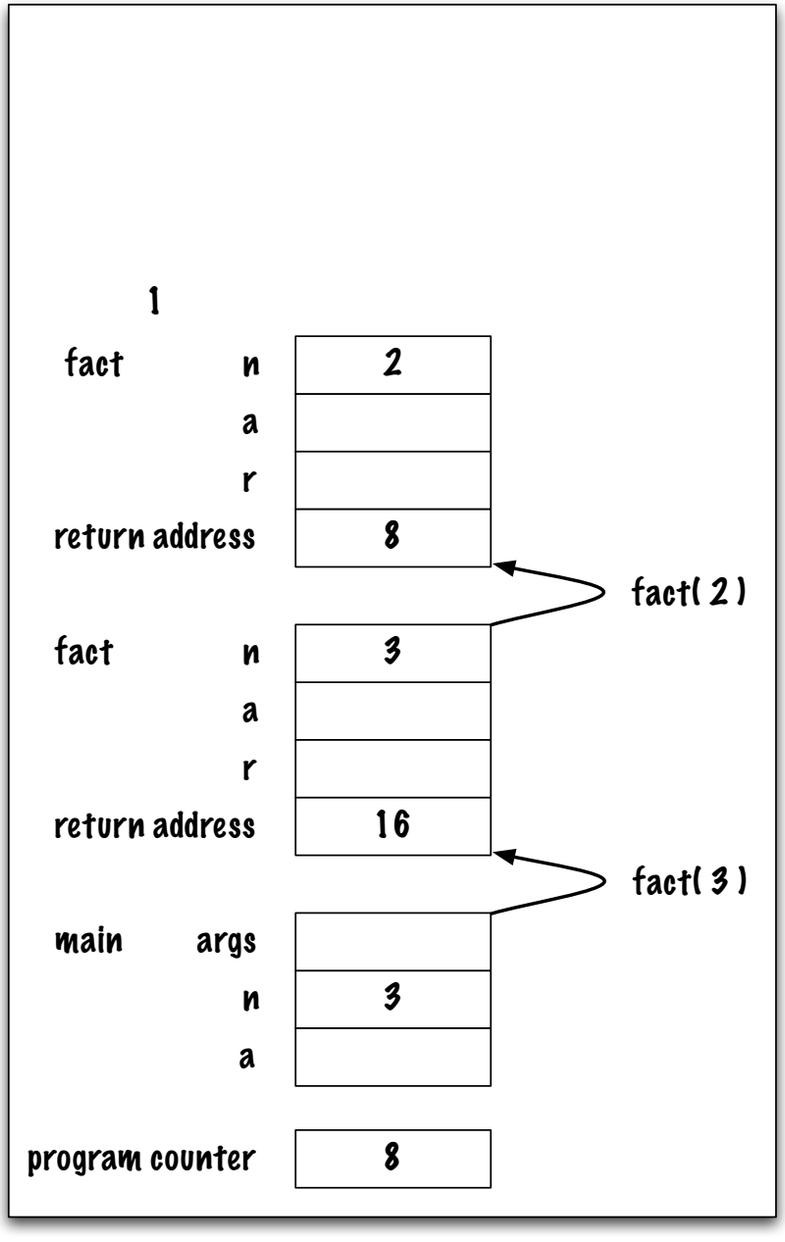

fact
n
1
a
r
return address
8
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
5

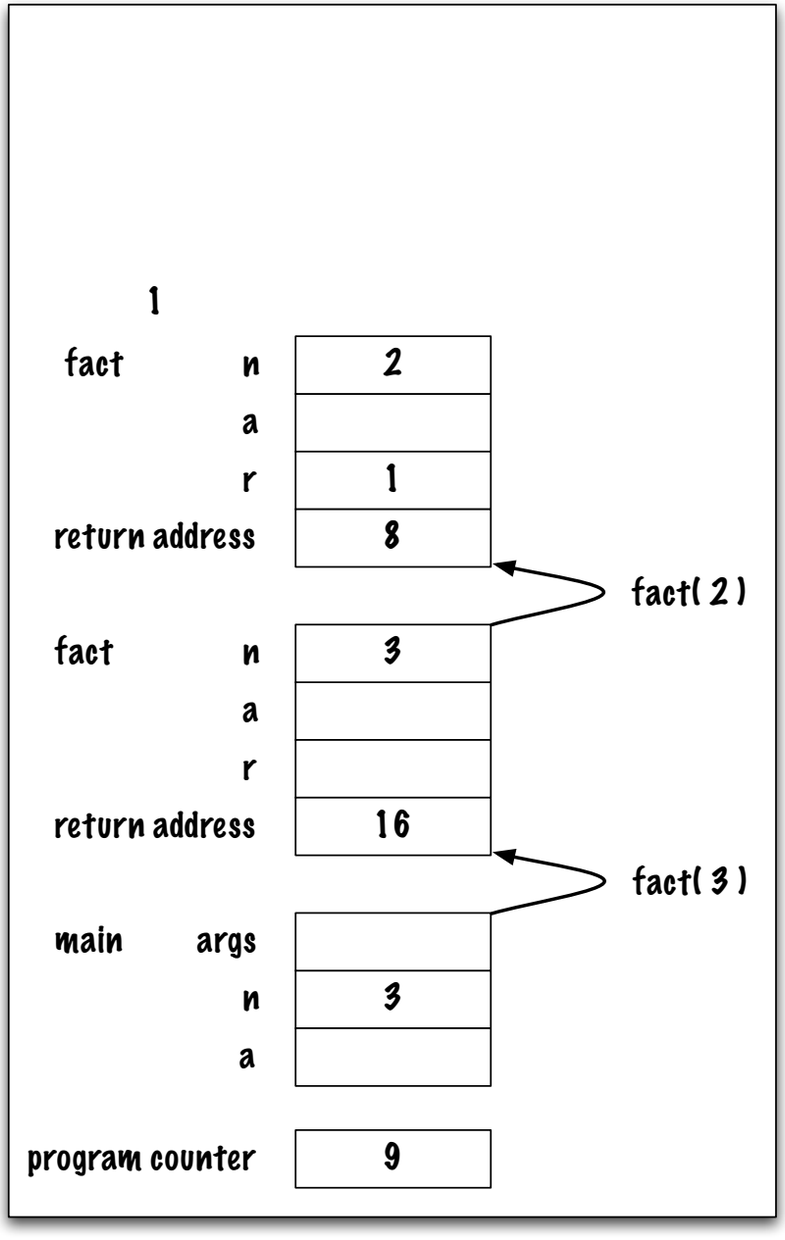

fact
n
1
a
r
return address
8
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
6

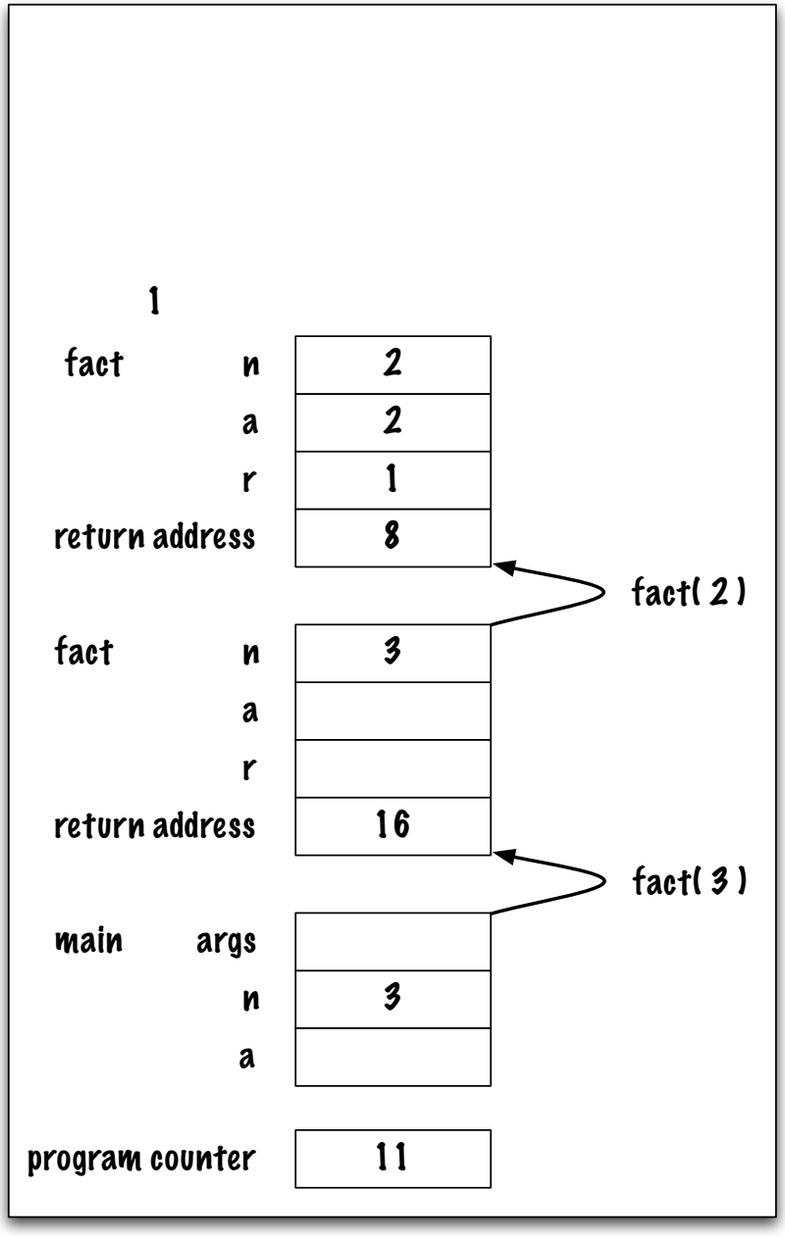

fact
n
1
a
1
r
return address
8
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
11

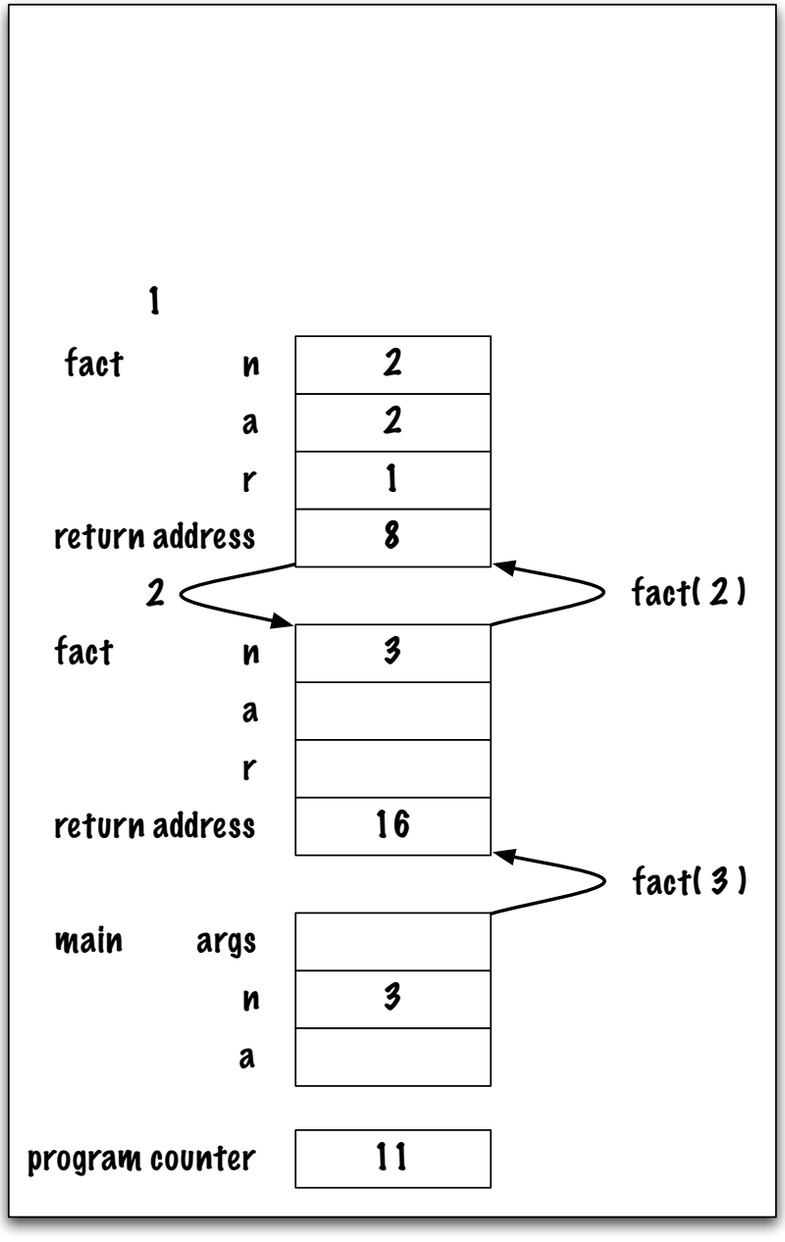

fact
n
1
a
1
r
return address
8
1
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
11

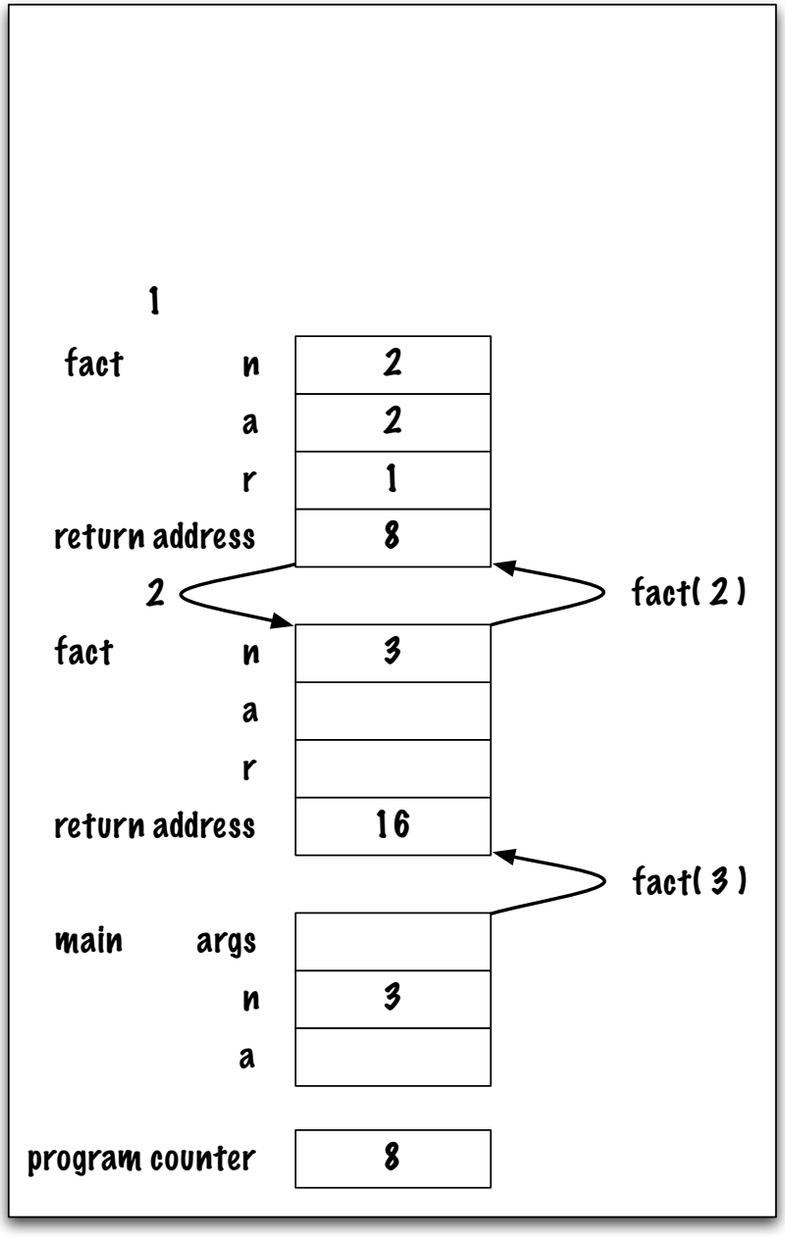

fact
n
1
a
1
r
return address
8
1
fact(1)
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

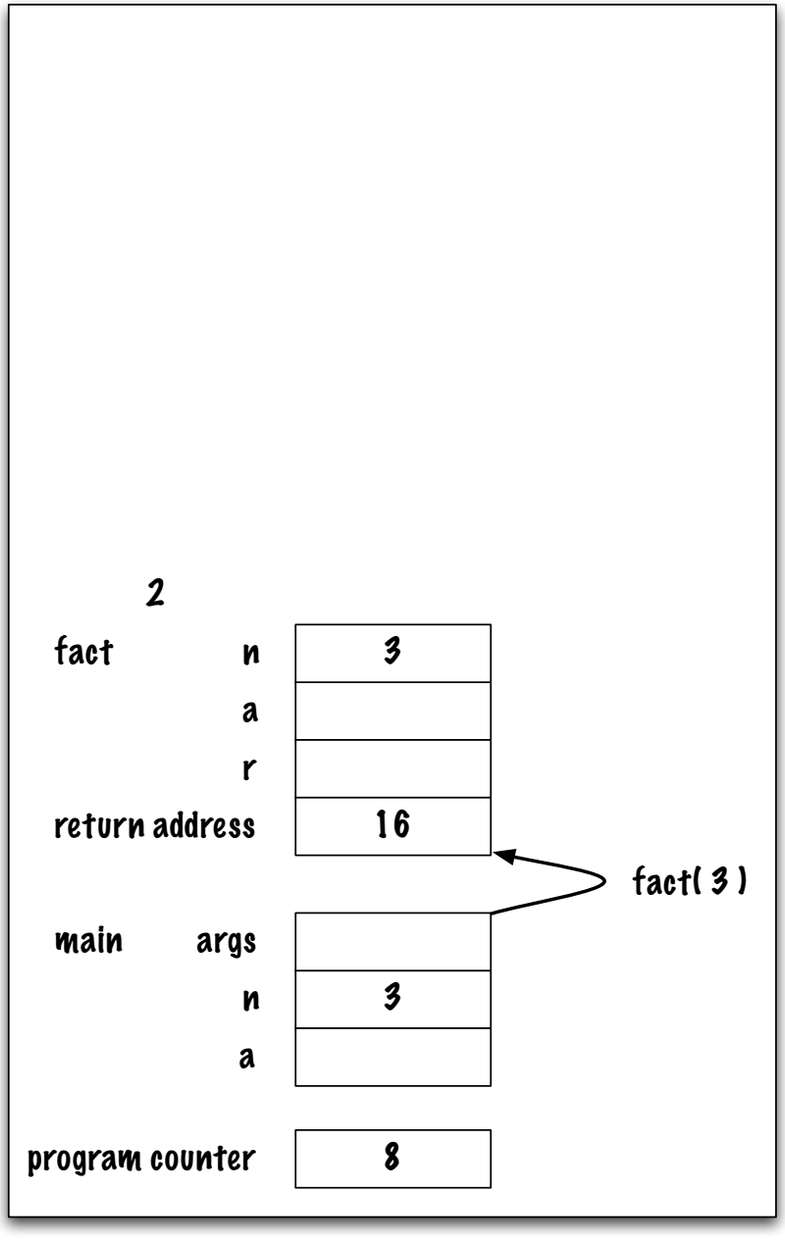

1
fact
n
2
a
r
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

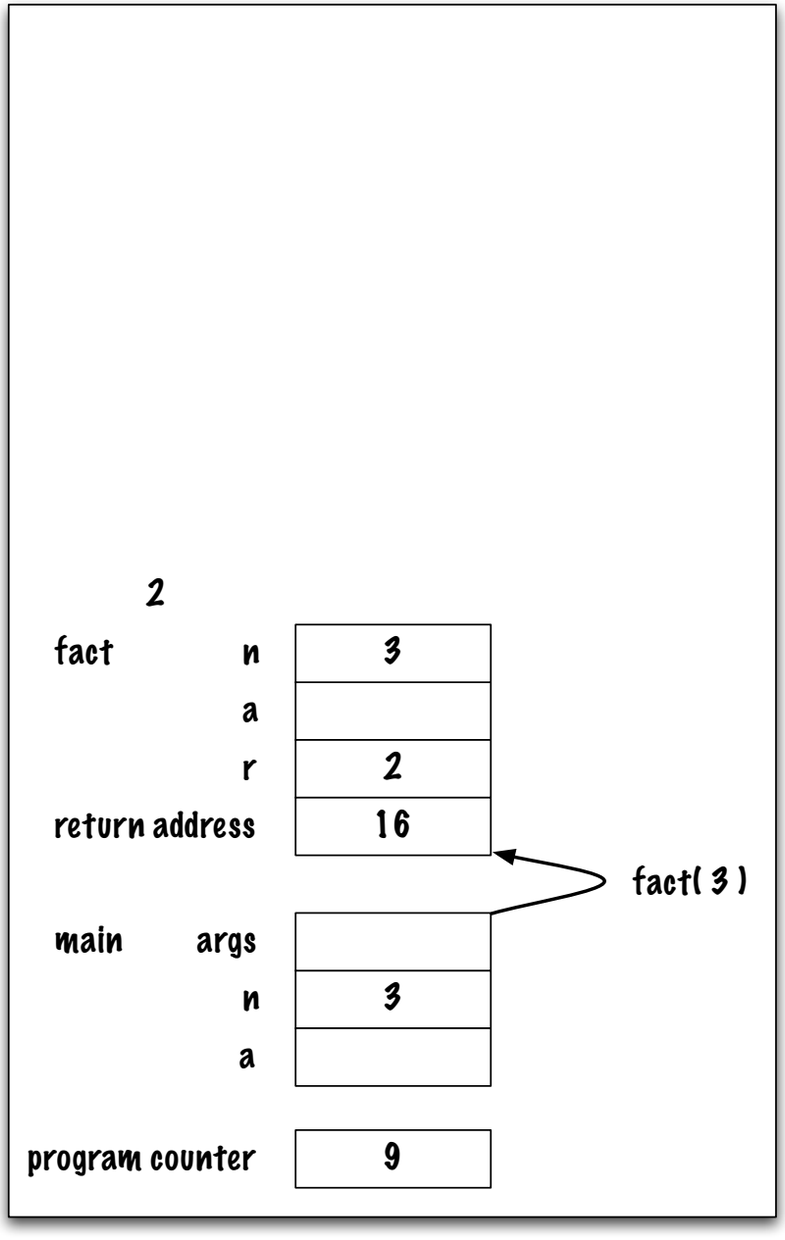

1
fact
n
2
a
r
1
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
9

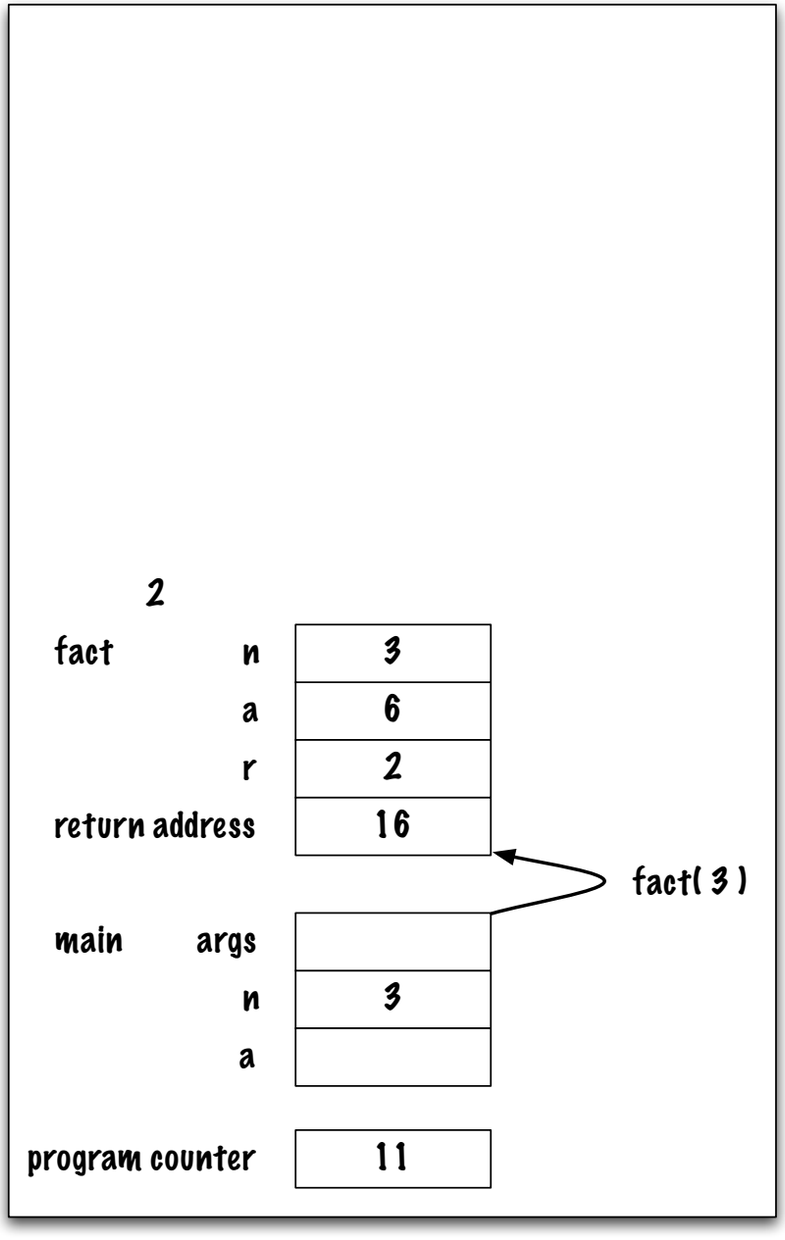

1
fact
n
2
a
2
r
1
return address
8
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
11

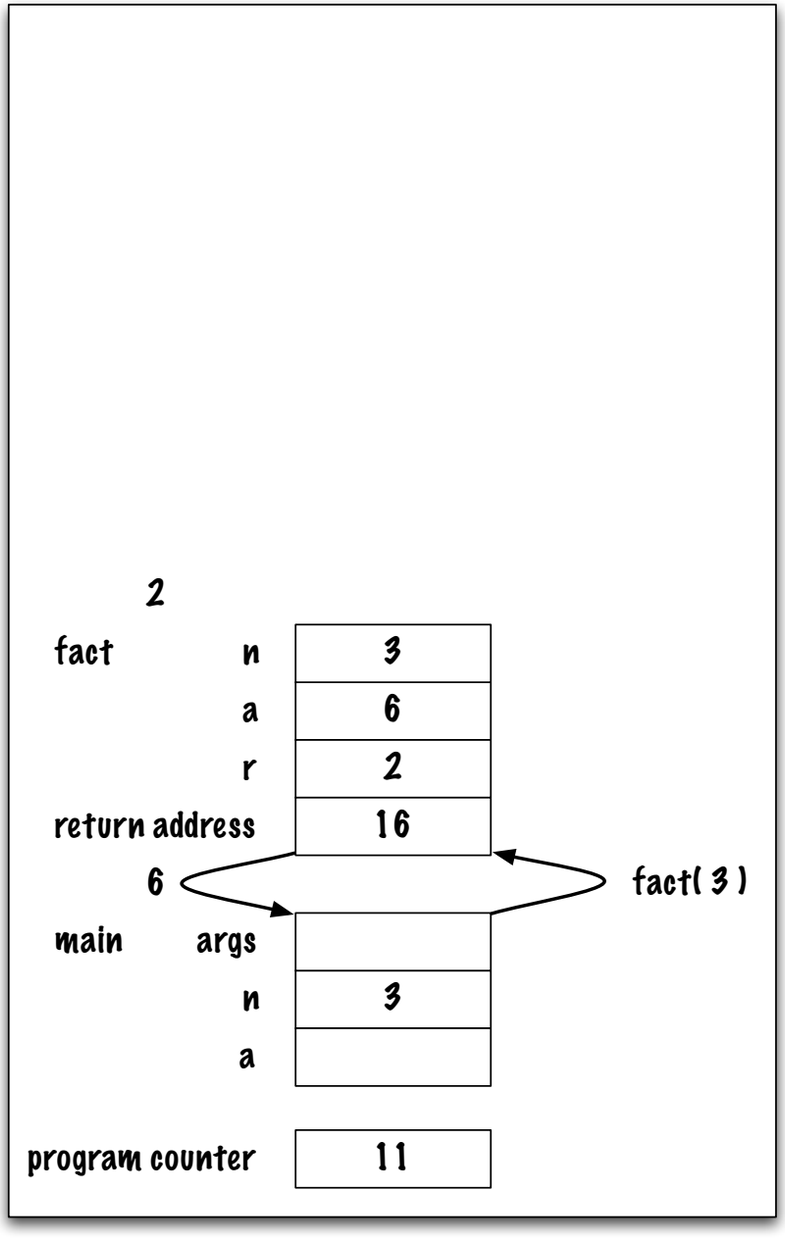

1
fact
n
2
a
2
r
1
return address
8
2
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
11

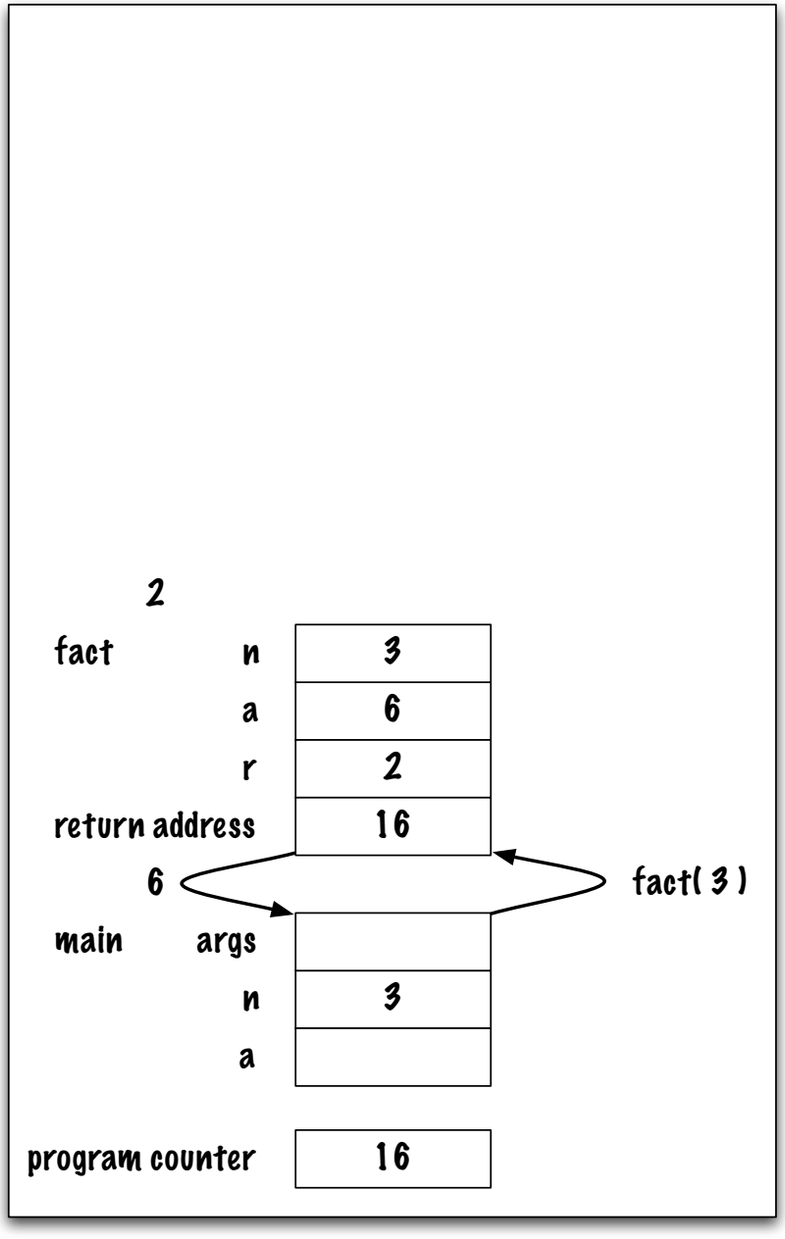

1
fact
n
2
a
2
r
1
return address
8
2
fact(2)
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

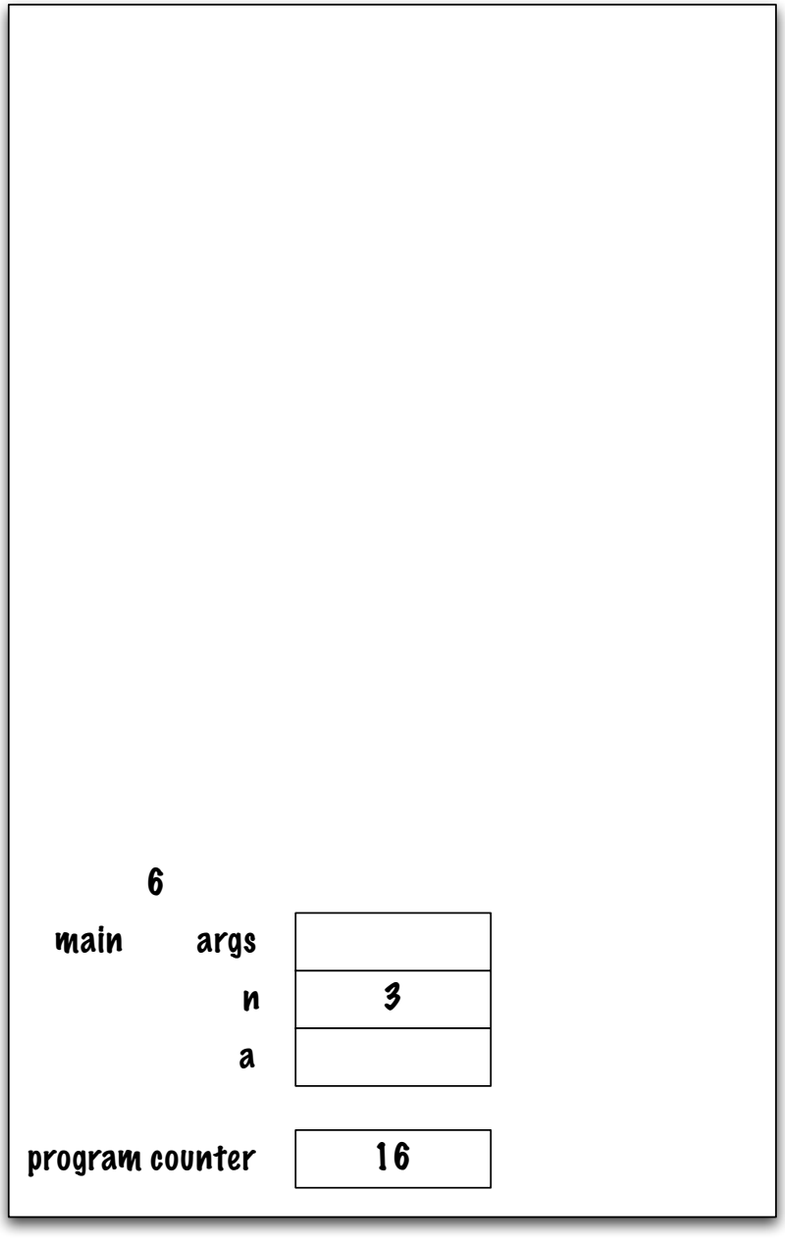

2
fact
n
3
a
r
return address
16
fact(3)
main
args
n
3
a
program counter
8

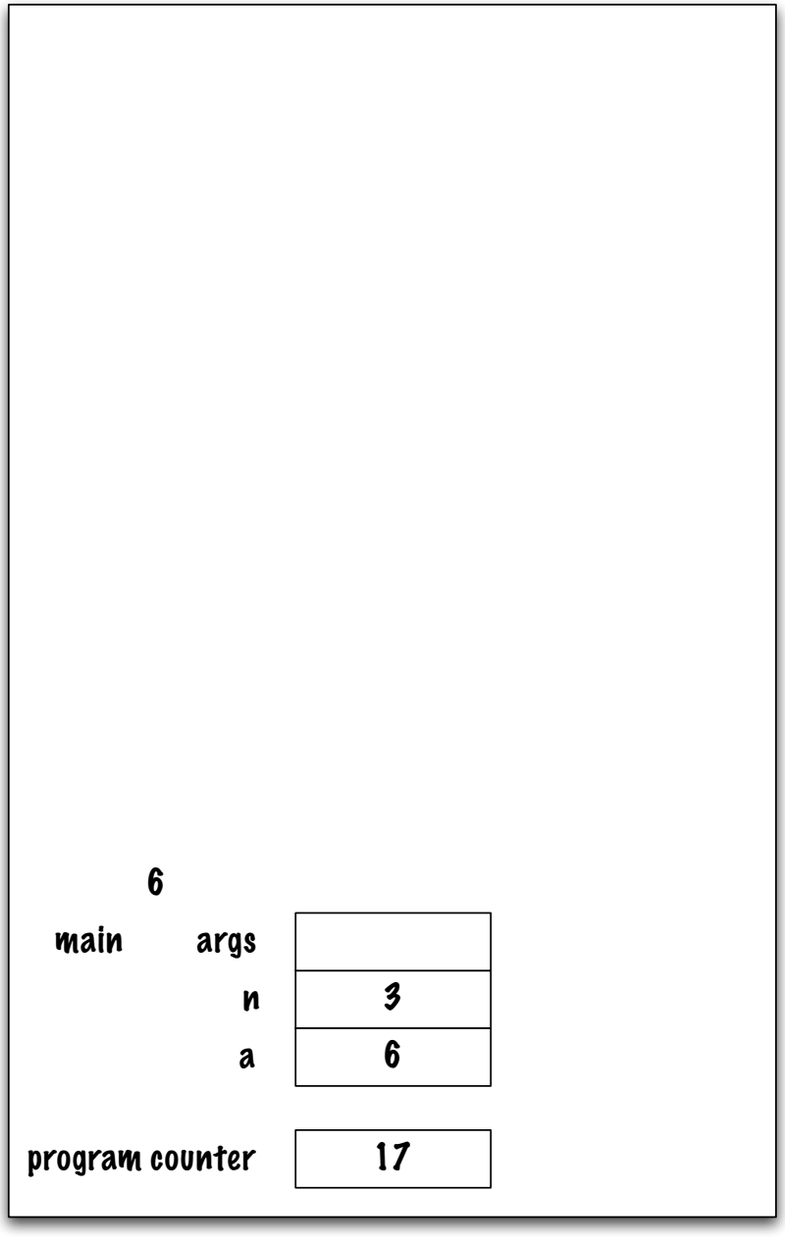

2
fact
n
3
a
r
2
return address
16
fact(3)
main
args
n
3
a
program counter
9


2
fact
n
3
a
6
r
2
return address
16
fact(3)
main
args
n
3
a
program counter
11


2
fact
n
3
a
6
r
2
return address
16
6
fact(3)
main
args
n
3
a
program counter
11


2
fact
n
3
a
6
r
2
return address
16
6
fact(3)
main
args
n
3
a
program counter
16


6
main
args
n
3
a
program counter
16


6
main
args
n
3
a
6
program counter
17