ITI 1521. Introduction à l'informatique II

Pile : applications

by

Marcel Turcotte

Version du 7 février 2020

Préambule

Préambule

Aperçu

Aperçu

Pile : applications

Nous examinons plusieurs exemples de leur utilisation, notamment l'évaluation d'expressions arithmétiques, la sauvegarde de l'historique des commandes, et l'exécution de programmes Java.

Objectif général :

- Cette semaine, vous serez en mesure d'appliquer les piles pour la conception d'algorithmes.

Préambule

Objectifs d’apprentissage

Objectifs d'apprentissage

- **Justifier** le rôle d'une pile dans la résolution d'un problème informatique.
- **Concevoir** un programme informatique nécessitant l'utilisation d'une pile.

Lectures :

- Pages 159-176 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

1 Préambule

2 Applications

3 Prologue

Applications

Applications

Évaluer une expression arithmétique

Application : Évaluer une expression arithmétique

Applications

Objectifs d'apprentissage :

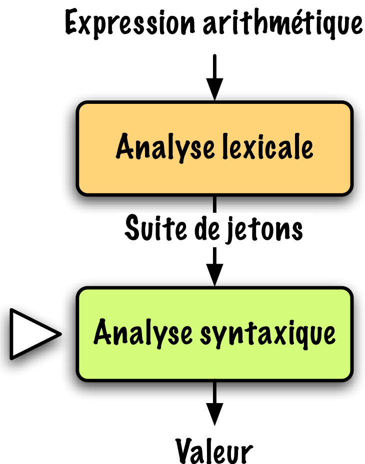
- **Justifier** le rôle d'une pile dans la résolution d'un problème informatique.
- **Concevoir** un programme informatique nécessitant l'utilisation d'une pile.

Lectures :

- Pages 159-176 de E. Koffman et P. Wolfgang.

Architecture de notre application

- Séparation claire des problèmes : **analyse lexicale** et **analyse syntaxique**
- L'**analyse lexicale** prend en entrée une **chaîne de caractères** et la découpe en morceaux appelés **jetons**
  - **Entrée** : $\cdot 1 \cdot + \cdot \cdot 2 \times 33 \cdots -4 \cdot$
  - **Sortie** : $[1, +, 2, \times, 33, -, 4]$
- Notre **analyse syntaxique** prend en entrée une suite de **jetons** et retourne la **valeur** de l'expression.
  - **Entrée** : $[1, +, 2, \times, 33, -, 4]$
  - **Sortie** : 63



StringTokenizer

Java possède un **analyseur lexical** !

```
StringTokenizer st;
st = new StringTokenizer(" 1 +  2    * 33 - 4");

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

1

+

2

*

33

-

4

StreamTokenizer est plus polyvalent !

Scan

Prenez quelques minutes pour analyser cet exemple. **Qu'en pensez-vous ?**

```
public static int scan(String expression) {

    StringTokenizer st; String op; int l,r;

    st = new StringTokenizer(expression);

    l = Integer.parseInt(st.nextToken());

    while (st.hasMoreTokens()) {
        op = st.nextToken();
        r = Integer.parseInt(st.nextToken());
        l = eval(l,op,r);
    }

    return l;
}
```

```java
    private static int eval(int l, String op, int r) {
        int result;
        switch (op) {
            case "+":
                result = l + r;
                break;
            case "-":
                result = l - r;
                break;
            case "/":
                result = l / r;
                break;
            case "*":
                result = l * r;
                break;
            default:
                System.exit(-1);
        }
        return result;
    }
```

Exercices

- Que retourne l'appel **scan(" 3 * 12 + 4")** ?
- Que retourne l'appel **scan(" 3 + 12 * 4")** ?
- Qu'en pensez-vous ?

Discussion

- L'algorithme **scan** évalue les opérations de gauche à droite, sans tenir compte de la **priorité des opérations**.
- L'algorithme **scan** ne traite pas les **parenthèses**.

Il y a deux solutions :

- Utiliser une nouvelle représentation pour les expressions
- Utiliser un algorithme plus complexe

$\Rightarrow$ Ces deux solutions nécessitent l'utilisation, implicite ou explicite, d'une **pile** !

Notations. Il y a trois façons de représenter une expression : $l \diamond r$, où $\diamond$ est un opérateur.

infixe : La notation infixe correspond à la notation habituelle, l'opérateur est mis en sandwich entre ses opérandes : $l \diamond r$.

postfixe : En notation postfixe, les opérandes sont placés devant l'opérateur, $l \;\; r \;\; \diamond$. On appelle 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*) et les langages **PostScript** et **PDF**.

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

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

*www.calculator.org/rpn.html

De l’infixe au postfixe

- 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 $l \ \ r \ \ \diamond$,
  où l et r sont elles même des sous-expressions et $\diamond$ est un opérateur.

Évaluer une expression 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 (2) opérandes par le résultat.

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

Évaluer une expression postfixe (mentalement)

Quelques **exercices** :

- $9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$
- $9 \quad 2 \quad 4 \quad * \quad 5 \quad - \quad /$

$9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$

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**.

- $2 + (3 * 4) \rightarrow 2 \quad 3 \quad 4 \quad * \quad +$
- $(2 + 3) * 4 \rightarrow 2 \quad 3 \quad + \quad 4 \quad *$

Pour évaluer une expression infixe, il faut tenir compte de la **priorité des opérateurs** ainsi que des **parenthèses**.

- Dans le cas de la notation postfixe, **ces concepts sont représentés à même la notation**.

$9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$

9 2 4 * 5 − /

Exercices

- Donnez le **contenu de la pile** à chaque itération de l'algorithme :
  - $9 \quad 3 \quad / \quad 10 \quad 2 \quad 3 \quad * \quad - \quad +$
  - $9 \quad 2 \quad 4 \quad * \quad 5 \quad - \quad /$
- Modifiez l'algorithme afin qu'il construise une expression **infixe** à partir d'une expression **postfixe** donnée en entrée

Applications

Discussion sur l'utilité des types abstraits de données

Discussion

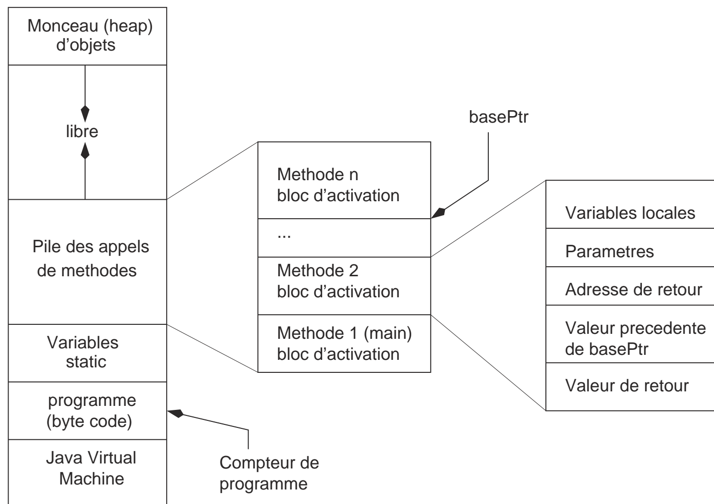
- Veuillez maintenant répondre à la question posée plus tôt : « l'une des implémentations proposées utilise un tableau, **pourquoi n'utilise-t-on pas tout simplement un tableau** pour la conception d'algorithmes ? **Quels sont les avantages** ? »

Applications

Gestion de la mémoire

Application : Gestion de la mémoire lors de l'exécution d'un programme

Représentation de la mémoire et interprétation d'un programme



La machine virtuelle de Java (**JVM**) doit :

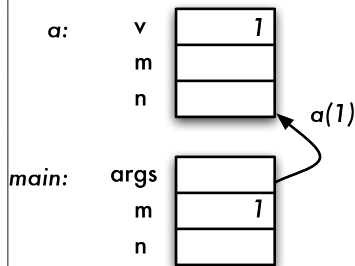
1. Créer un nouveau **bloc d'activation** (la valeur de retour, valeur précédente de basePtr et l'adresse de retour ont une taille fixe, la taille des variables locales et des paramètres dépend de la méthode) ;
2. **Sauver** la valeur courante de basePtr, à l'espace « valeur précédente de basePtr », 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

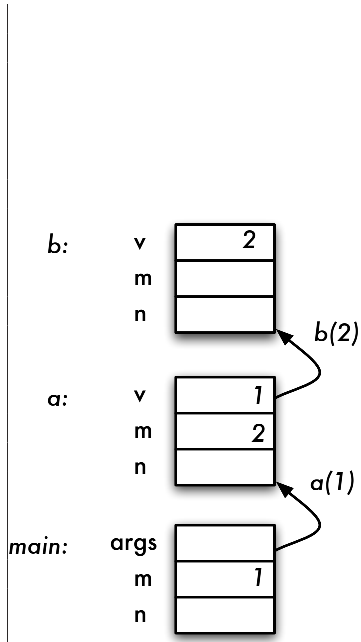
1. La méthode **sauve la valeur de retour** à l'endroit indiqué par « valeur de retour » ;
2. **Retourne le contrôle** à la méthode appelante, c.-à-d. remet en place les valeurs de locationCounter et basePtr ;
3. **Retire le bloc d'activation** courant ;
4. **Reprend** l'exécution à l'endroit désigné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

```java
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[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```

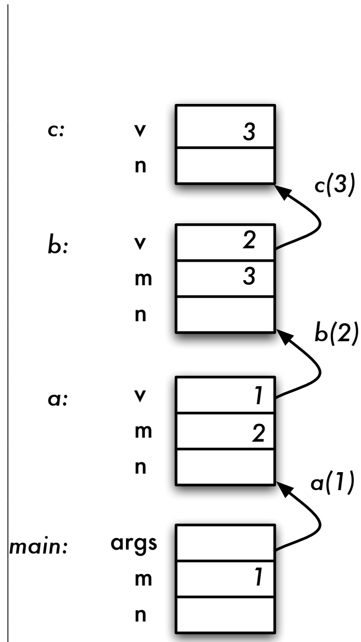
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



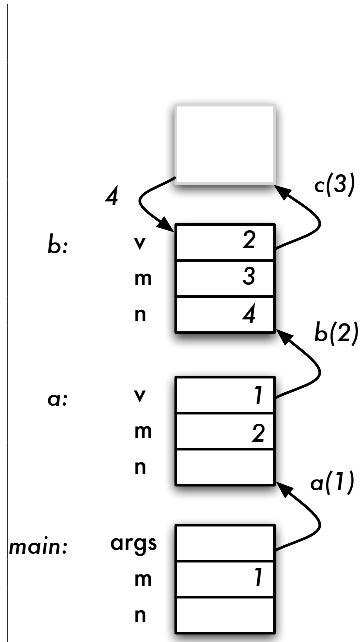
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



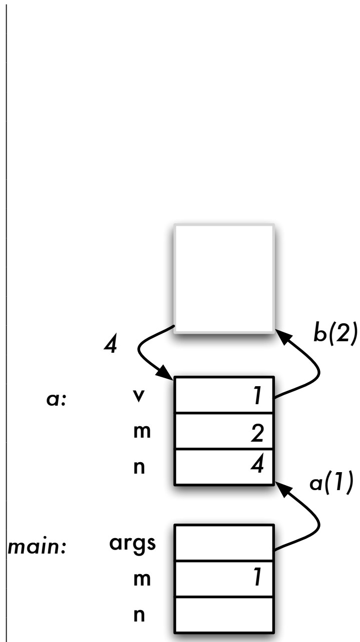
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



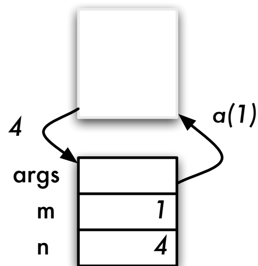
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



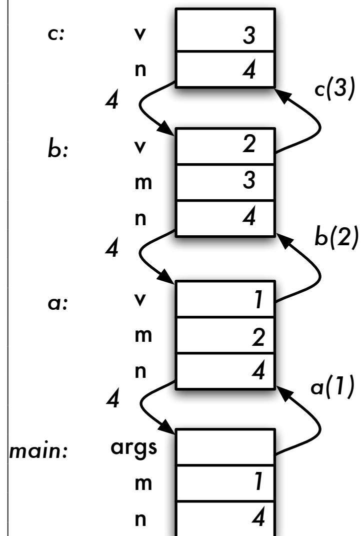
```java
public static int c(int v) {
    int n;
    n = v + 1;
    return n;
}
public static int b(int v) {
    int m,n;
    m = v + 1;
    n = c(m);
    return n;
}
public static int a(int v) {
    int m,n;
    m = v + 1;
    n = b(m);
    return n;
}
public static void main(String[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



```java
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[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



```java
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[] p) {
    int m = 1,n;
    n = a(m);
    System.out.println(n);
}
```



Prologue

Résumé

- On utilise **une pile** lorsqu'on souhaite traiter les éléments **dans l'ordre inverse**.

Prochain module

- **Pile** : éléments chaînés

References I

E. B. Koffman and Wolfgang P. A. T.
***Data Structures : Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa