Programmation Comparée

Cours Introductif

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Plan

Principe du cours

Programmation : Quels outils ? Quels problèmes ?

Objectifs et fonctionnement du cours

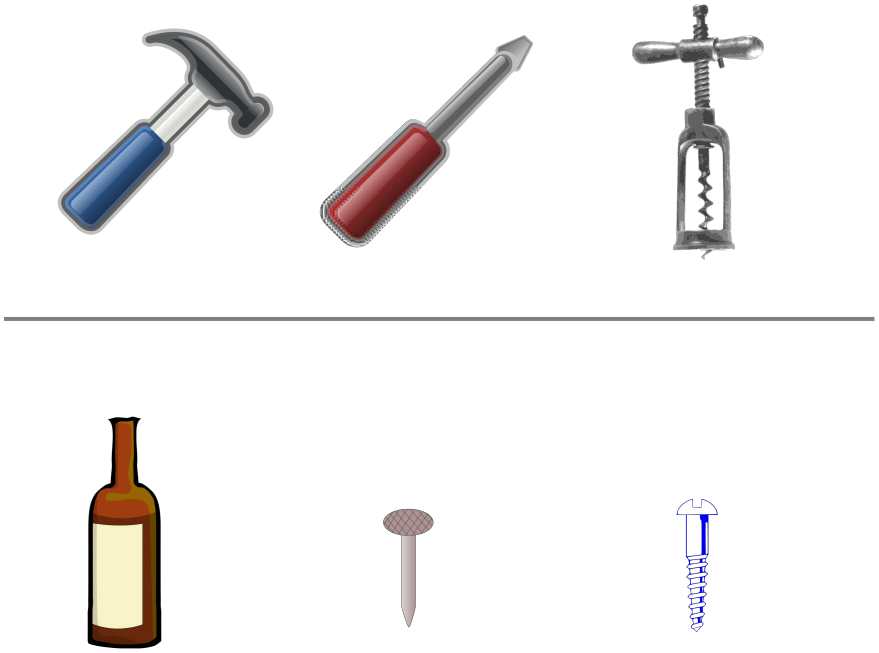
Plan

Principe du cours

Programmation : Quels outils ? Quels problèmes ?

Objectifs et fonctionnement du cours

Qu’est-ce que la programmation ?



(Une définition plus sérieuse vient un peu plus loin.)

Il faut bien comprendre l’outil en lui-même et son utilité.

Comment progresser en programmation ?

Apprentissage centré sur les mécanismes de programmation

Apprentissage linguistique

1. Comprendre un mécanisme de programmation
2. Raisonner sur des programmes écrits à l'aide de ces mécanismes
3. L'intégrer dans sa boîte à outils

Apprentissage méta-linguistique

- Didactique : Apprendre à apprendre un nouveau langage
- Dialectique : Comprendre ce qui se passe (en comprenant ce qui s'est passé)

Le développement logiciel

QUOI?	Prendre conscience d'un problème	*Une distance observée entre ce que doit faire le programme et ce qu'il peut faire.*
	Décomposition	*Plusieurs sous-problèmes et leur articulation*
	Analyse	*Une spécification des problèmes.*
COMMENT?	Conception	*Une spécification des solutions à ces problèmes.*
	Programmation	*Une version exécutable de ces solutions.*
ET ALORS?	Evaluation	*Une nouvelle distance observée entre le résultat effectif et le résultat escompté.*

La programmation moderne

La programmation est l'activité qui consiste à construire une version exécutable de la solution à un problème.

Dans un développement moderne de logiciel, cette activité prend de nombreuses formes car les outils matériels et logiciels ont énormément évolué. Ainsi, on ne construit désormais que très rarement un logiciel à partir de rien. Le travail du programmeur s'appuie sur les mécanismes de langages de programmation différents, sur des environnements préexistants (framework), d'architecture particulière, sur des systèmes d'exploitation variés, ... Tous ces paramètres influent sur la façon de développer et sur la qualité du logiciel final.

Comment faire ces choix ?

Plan

Principe du cours

Programmation : Quels outils ? Quels problèmes ?

Objectifs et fonctionnement du cours

Langage de programmation

Langage de programmation

Un langage de programmation est un langage formel défini par une syntaxe et une sémantique. Plus un langage de programmation fournit de mécanismes de calcul distincts et plus il est dit *expressif*.

Un langage de programmation est une abstraction du code machine concrètement exécuté *en fine*. Son niveau d'abstraction dépend de sa distance à ce code machine. Plus un langage de programmation est proche de la formulation du problème issue de la phase d'analyse et plus il est dit *déclaratif*.

Outils autour des langages de programmation

Un compilateur traduit un programme dans un langage de plus bas niveau pou le rendre plus adapté à une exécution e□cace par la machine. Ainsi, un compilateur peut produire du code natif, c'est-à-dire du code exécutable directement par le processeur de la machine physique. Il peut aussi produire d code machine pour une machine virtuelle, simulée par une ou plusieurs machi concrètes.

Un interprète est un programme qui évalue le code source écrit par le programmeur sans passer par une phase de traduction préalable.

Un analyseur statique est un programme qui, sans exécuter le code source, fournit des informations sur ce dernier. Un typeur est un exemple d'analyseur statique : il fournit la garantie qu'un programme pourra s'exécuter sans erreur grossière et produire une valeur.

En fonction de la présence ou de l'absence de ses outils, un langage est dit interprété, compilé et avec ou sans analyseurs statiques.

Critères de classification

Il existe des milliers de langages de programmation. Le site suivant en référer
pas moins de 8512 :

http://hopl.murdoch.edu.au/

Il en existe de nombreuses taxonomies :

http://hopl.murdoch.edu.au/taxonomy.html
http://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf

La classification que nous allons suivre s'appuiera sur une méthode moins
systématique et rigoureuse que ces deux approches au profit d'un accent mis
les familles de langages les plus utilisées.

Langage assembleur

Les langages d'*assemblage*, ou simplement *assembleur*, sont des langages de bas-niveau liés à un jeu d'instructions d'un processeur particulier. Les assembleurs ont été inventés dans les années 50. Ils étaient alors nommés langages de seconde génération car ils remplacaient les langages machines, de première génération.

Un programme écrit en assembleur est généralement formé à l'aide de trois classes syntaxiques :

- Les *mnémomiques d'instruction* décrivent les opérations effectuées par le processeur.
- Les *directives d'assemblage* donnent des informations externes sur la façon de produire le code exécutable à partir du code assembleur.
- Les *sections de données* contiennent des descriptions des constantes utilisées par le programme.

Exemple de programme assembleur i386

Source : Wikipedia

```
.globl _start
DATA :  .ascii "Hello World"
_start :  mov $4 , %eax
          mov $1 , %ebx
          mov $DATA , %ecx
          mov $8 , %edx
          int $0x80

          mov $1 , %eax
          mov $0 , %ebx
          int $0x80
```

Pour produire un exécutable sous Linux :

```
% gcc foo.S -c -o foo.o
% ld foo.o -o foo
% ./foo
```

Exercice

1. Écrire un programme assembleur qui sort avec un statut de sortie valant 4
2. Écrire le meilleur joueur de Corewar.

Mécanismes principaux du langage assembleur

Par définition, l'assembleur ne fournit pas d'abstraction au dessus du processe
En revanche, tous les mécanismes de calcul du processeur sont accessibles.
Ainsi, du point de vue de l'utilisation optimale des fonctionnalités de la machine, le langage assembleur est très expressif.

Par ailleurs, comme la structure du code machine est totalement libre, certains mécanismes sont réalisables alors qu'ils sont inaccessibles dans (la plupart de les langages de haut niveau :

- Factorisation de code maximale : les conventions d'appel de procédure créé généralement des barrières d'abstraction empêchant la factorisation fragments de code communs entre les procédures.
- Code auto-généré et auto-modifiant : un programme assembleur peut (généralement) écrire dans son segment de code ce qui lui permet de se modifier pour s'optimiser (dérouler une boucle par exemple) ou encore décrypter une partie de son code.

Utilisations courantes de l'assembleur

De nos jours, l'assembleur est peu utilisé car les compilateurs des langages d haut-niveau produisent en général du code d'e□cacité très proche de ce que l pourrait écrire à la main. En particulier, les algorithmes d'allocation des registr modernes peuvent produire des résultats qu'un programmeur aurait du mal à obtenir.

De plus, un programme écrit en assembleur est di□cile à écrire et à maintenir car il est généralement moins concis et moins modulaire qu'un programme éc dans un langage de haut niveau. De plus, il n'est pas portable.

Il reste néanmoins quelques applications où l'assembleur est utilisé :

- Certaines instructions très spécifiques des microprocesseurs ne sont pas gérées par les compilateurs. Pour en tirer parti, une routine assembleur e nécessaire.
- Les drivers s'appuient sur une interaction très précise avec les ports d'entrées/sorties et les registres spéciaux des processeurs que l'on ne pe pas forcément contrôler dans un langage de haut-niveau.

Exemple de routine assembleur en ligne

```
for (number = 0 ; number≤ max ; ++ number) {
   for (i = (number ≫ 1), position = 0 ; i != 0 ; ++ position)
      i ≫= 1 ;
   result = position ;
}
```

Que fait ce programme C ?

Exemple de routine assembleur en ligne

```
for (number = 0 ; number≤ max ; ++ number) {
   asm ("bsr1 %1, %0": "=r" (position) : "r" (number)) ;
   result = position ;
}
```

Ce programme s'exécute 6 fois plus rapidement.

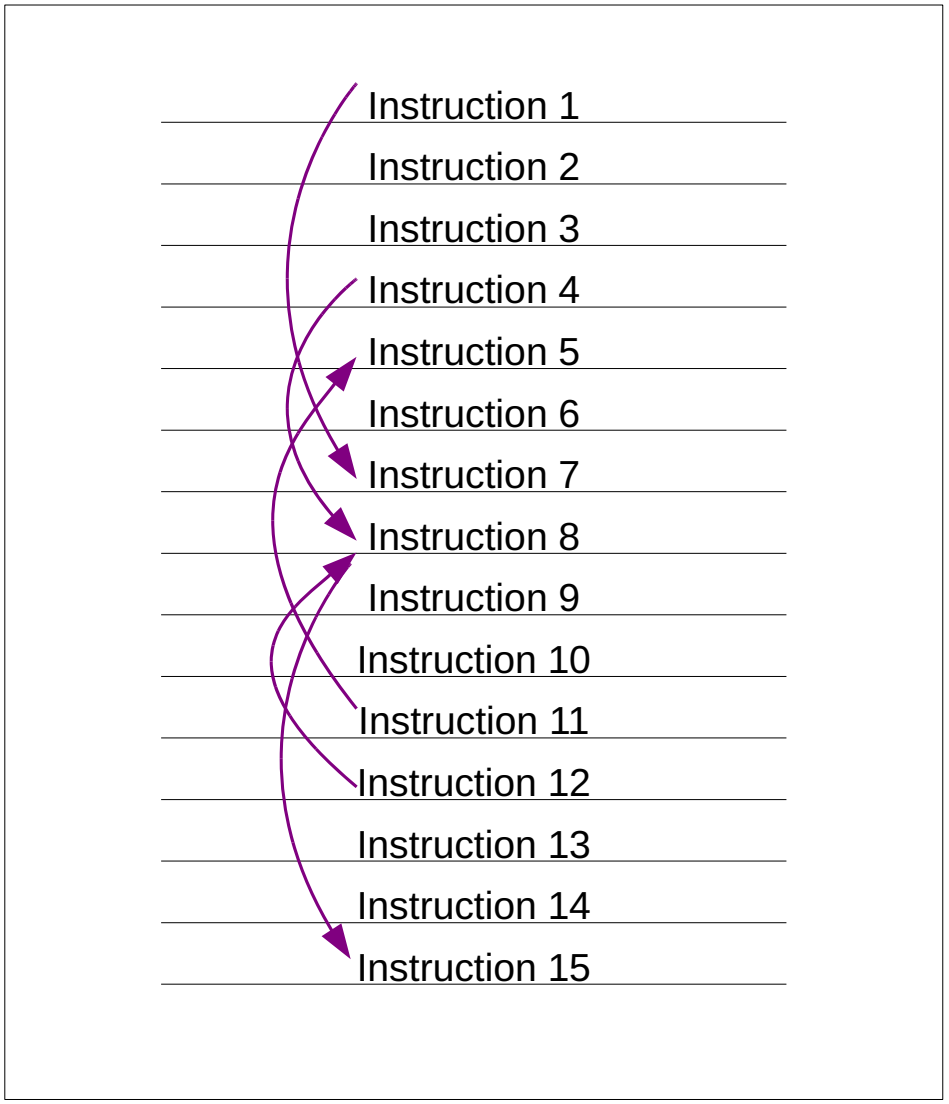
Conclusion sur l'assembleur

Nous n'étudierons pas la programmation assembleur dans ce cours de programmation comparée car son utilisation est devenue rare. Néanmoins, il e sans doute enrichissant d'avoir écrit une routine assembleur une fois dans sa de programmeur.

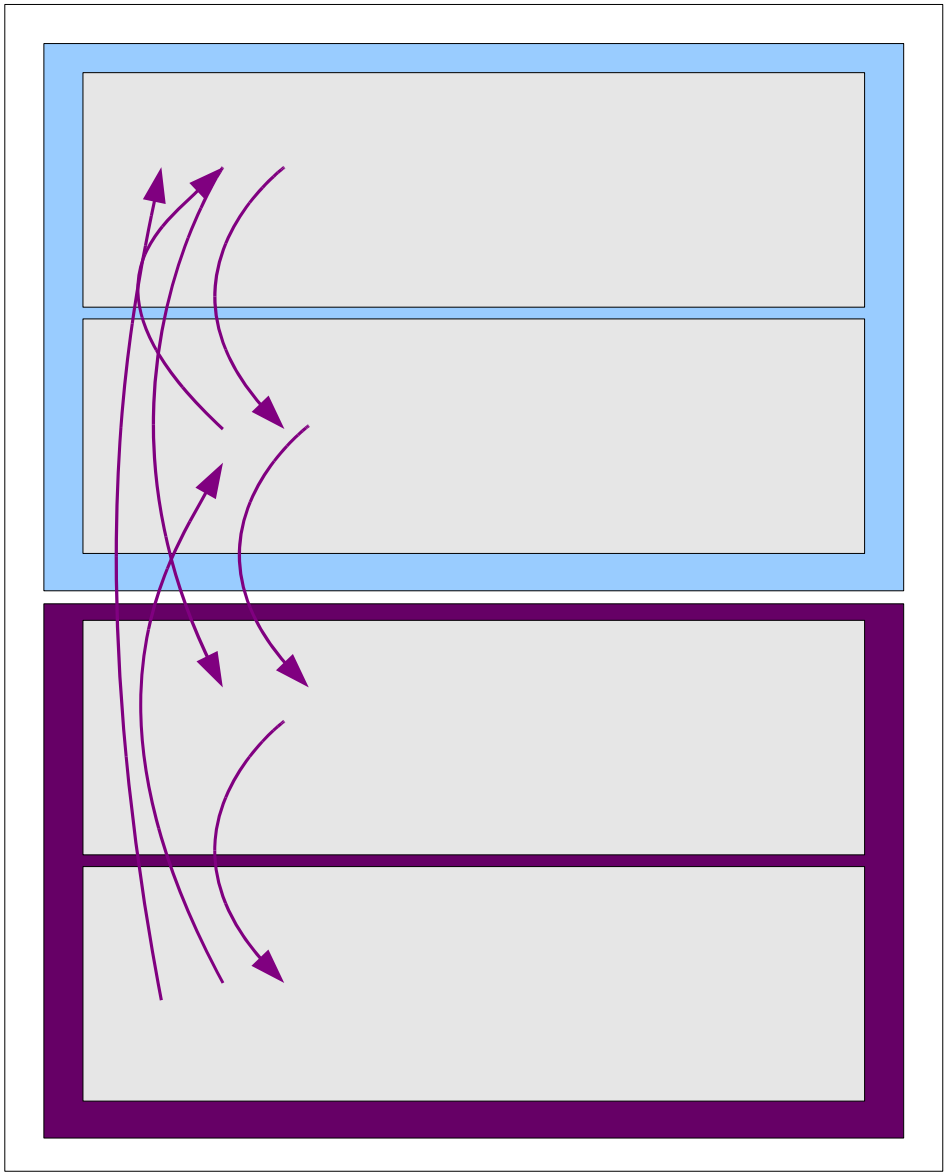
Pour aller plus loin...

- Programming from the Ground Up, Jonathan Bartlett
- Les sources Prince of Persia
- Le code source du module de guidage d'atterissage d'APOLLO 11.

De la programmation non structurée . . .



. . . à la programmation structurée.



Langages procéduraux du premier ordre

Dans les années 60, la crise du logiciel a conduit à une réflexion autour de la méthodologie de la construction des programmes informatiques, aboutissant à l'émergence du génie logiciel. Du point de vue des langages de programmatio des mécanismes pour structurer la programmation et les programmes sont apparus :

- Structures de contrôle de haut-niveau : while, if-then-else, ...
- Procédures et fonctions (de seconde classe).
- Suppression du goto
  (Un point unique de sortie et d'entrée pour les boucles.)
- Minimisation des variables globales au profit de variables locales.

Un article de recherche de l'époque explique bien pourquoi ces restrictions de l'expressivité des langages de programmation sont bénéfiques :

Go To Statement Considered Harmful — Edsger Dijkstra
http://www.ifi.uzh.ch/req/courses/kvse/uebungen/Dijkstra_Goto.pdf

Exemple de programme procédural en ADA me me me

La programmation structurée

En plus de ces langages de programmation procéduraux, cette programmation plus structurée est également caractérisée par une méthologie de conception programmes par ra□nements successifs descendants.

Nous étudierons cette méthode dans le prochain cours et la comparerons aux méthodes de modélisation que vous connaissez déjà.

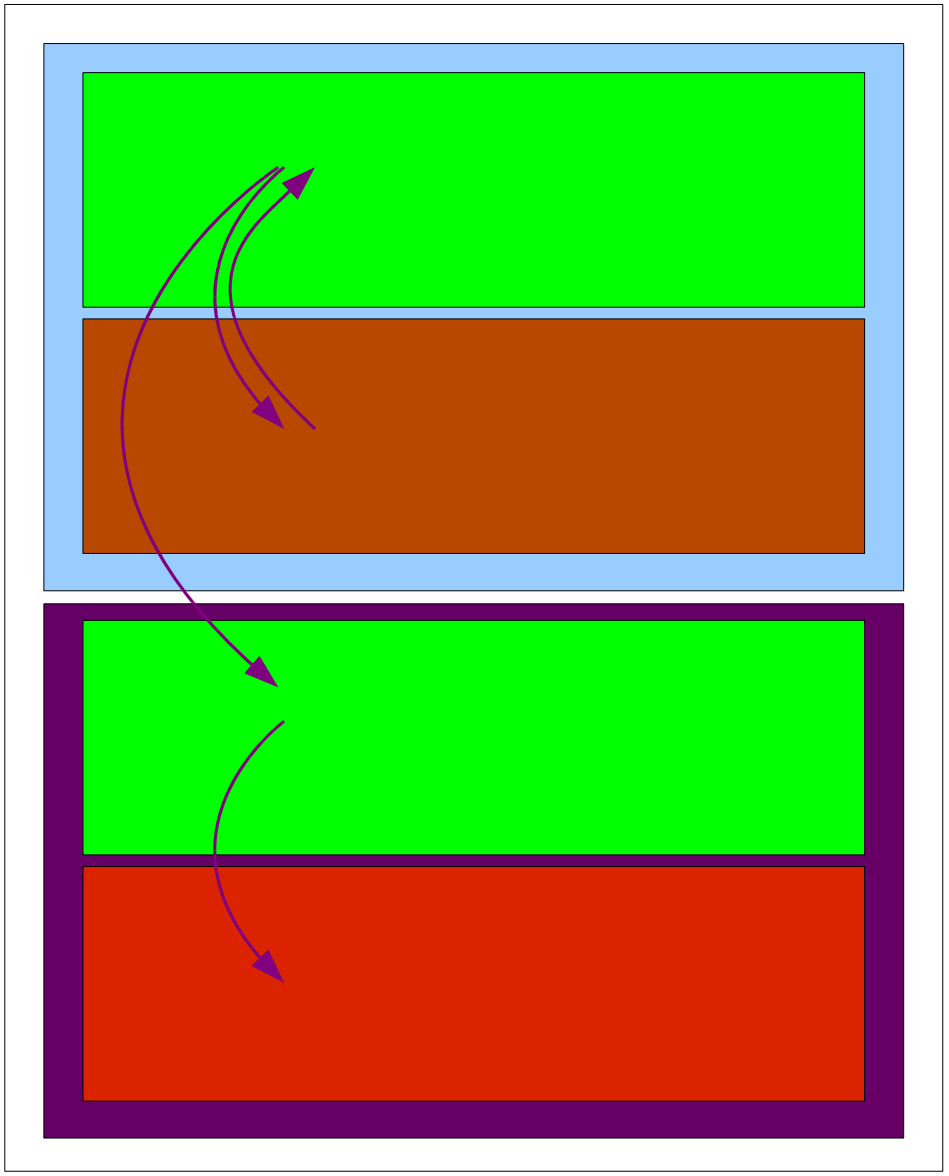
La programmation en grand

Les logiciels devenant de plus en plus complexes, les besoins de fragmentatio de leur développement se sont sentis de plus en plus forts. Des outils linguistiques pour mener une “programmation en grand” (Programming in the large) ont alors été proposés. Ils visaient à faciliter l’interconnection de composants logiciels :

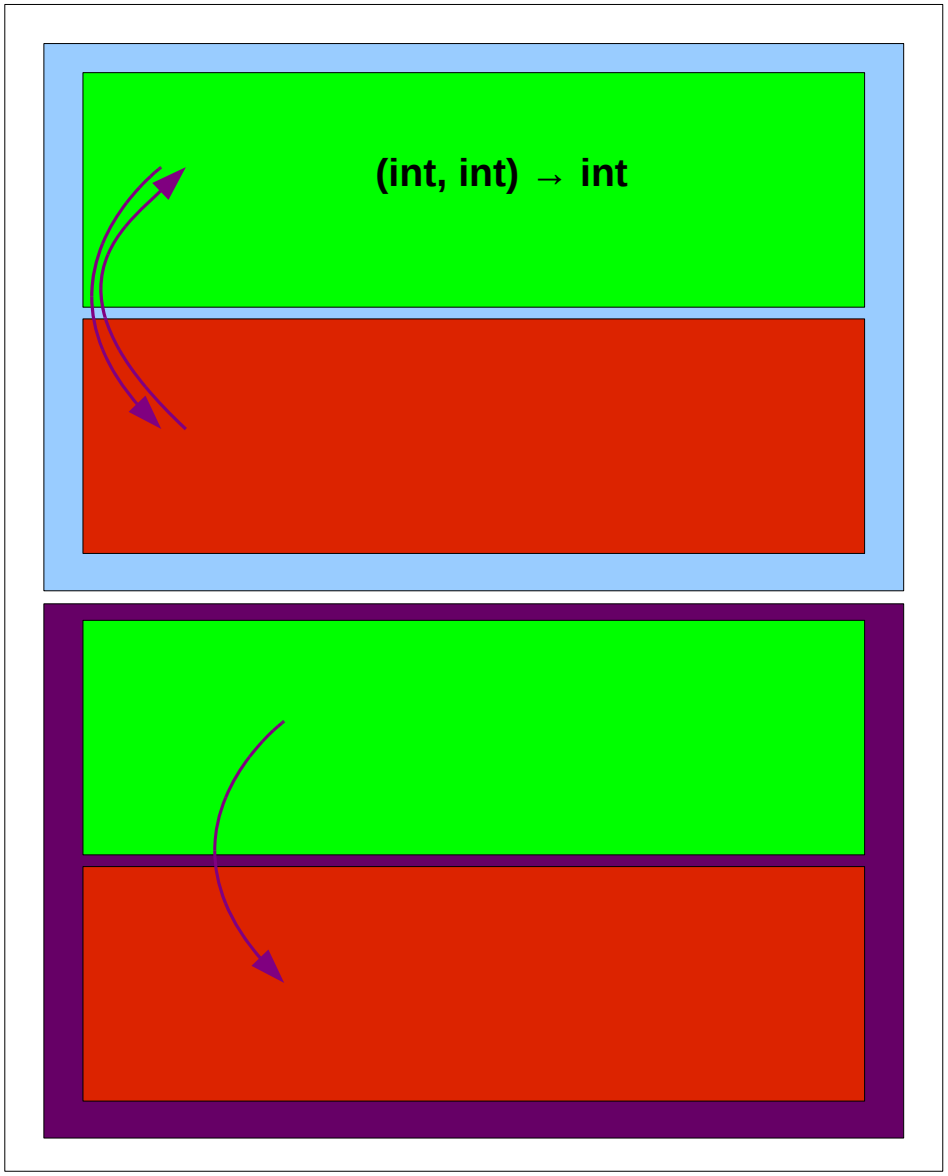
> By large programs we mean systems consisting of many small programs (modules), possibly written by di□erent people. We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a “module interconnection language” for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler. We explore the software reliability aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines.

Programming-in-the-large versus programming-in-the-smal
DeRemer, Frank ; Kron, Hans (1975)

Des mécanismes pour …



. . . limiter les interdépendances



La modularité

Nous allons étendre la réflexion déjà entamée en POCA sur les propriétés de modularité des composants logiciels en étudiant et comparant les mécanisme suivants :

- Les types abstraits.
- Les fonctions d'ordre supérieur.
- Les modules, les classes, les mixins et les classes de type.
- Les messages de première classe.

Nous verrons aussi comment la modularité peut être obtenue par di□érentes formes de décomposition du logiciel en composants et composition des composants en logiciel. En particulier, nous aborderons et comparerons :

- La programmation orientée objet.
- La programmation orientée aspect.
- La programmation orientée sujet.
- La programmation orientée rôle.
- La programmation fonctionnelle.

Les langages d'ordre supérieur

Une idée au cœur de la programmation modulaire est de considérer le code exécutable comme un objet de première classe, c'est-à-dire comme pouvant être l'issue d'un calcul ou bien un de ses paramètres.

Nous comparerons les différentes réalisations de cette idée, qu'ils s'agissent d S-expressions de LISP ou bien des fermetures des langages objets et fonction

Pouvoir écrire une fonction d'ordre supérieur, c'est-à-dire une fonction qui manipule d'autres fonctions, induit un gain très important en expressivité puisque le programmeur peut par exemple définir lui-même ses propres opérateurs de contrôle.

Par ailleurs, nous verrons comment le paradigme de la programmation fonctionnel apporte une augmentation de la déclarativité, facilitant le raisonnement sur les programmes. On peut pousser plus loin encore cette déclarativité en contrôlant les effets de bord comme le font les langages de programmation fonctionnelle pure.

Exemple de programme fonctionnel

```
mkRands = mapM (randomRIO.(,)0 ). enumFromTo 1. pred

replaceAt :: Int → a → [a] → [a]
replaceAt i c = let (a,b) = splitAt i l in a++x : (drop 1 b)

swapElems:: (Int, Int) → [a] → [a]
swapElems (i,j) xs| i ≡ j = xs
                  | _ = replaceAt j (xs!!i) $ replaceAt i (xs!!j) xs

knuthShu□e :: [a] → IO [a]
knuthShu□e xs =
  liftM (foldr swapElems xs. zip [1..]) (mkRands (length xs))
```

Langages de programmation générique et de métaprogrammation

Certes, les langages d'ordre supérieur manipulent du code exécutable mais d'façon assez limitée : il n'est par exemple pas possible d'écrire une fonction d'ordre supérieur qui inspecte une fonction qui lui est passée en argument pour en produire une autre plus e□cace ou en l'adaptant pour traiter une nouvelle forme de données.

La méta-programmation consiste à écrire des programmes qui manipulent le code source d'autres programmes. De nombreuses transformations automatiqu de programmes sont utiles pour éviter au programmeur l'écriture de code répétitif.

La distinction entre *run-time* et *compile-time* est souvent supprimée dans un langage de méta-programmation. On peut ainsi calculer et recompiler certains fragments d'un programme pendant l'exécution.

Exemple de spécialisation de programmes

```
let lift x = .<x>.
let rec member l y =
  match l with
  j [] !  .< false>.
  j x : : xs !  .< if . (lift x) = . y then true else . (member xs y)>.

let spec_member = .< fun x ! . (member [1 ;2 ;3] x) >.
```

$spec_{member}$ est une version spécialisée de la fonction member.