IFT2035 – Concepts des langages de programmation

http://www.iro.umontreal.ca/~feeley/cours/ift2035/

Copyright © 1996-2011 Marc Feeley

Un programme simple

© 2011 Marc Feeley


```
(display "Bienvenue au cours IFT2035!\n")
```

- Quel langage de programmation?
- Que fait ce programme?

Un programme mystérieux

© 2011 Marc Feeley


```
%!PS
/Times-Roman findfont 150 scalefont setfont
180 100 moveto
60 rotate
(Bi) show
1 0 0 setrgbcolor (en) show
0 1 0 setrgbcolor (ve) show
0 0 1 setrgbcolor (nue) show
showpage
```

- Quel langage de programmation?
- Que fait ce programme?

Un programme obscur

© 2011 Marc Feeley


```
#define P(X)j=write(1,X,1)
#define C 21
int M[5000]={2},*u=M,N[5000],R=17,a[4],l[]=
{0,-1,C-1,-1},m[]={1,-C,-1,C},*b=N,*d=N,c,e
,f,g,i,j,k,s;main(){for(M[i=C*R-1]=24;f|d>=
b;){c=M[g=i];i=e;for(s=f=0;s<4;s++)if((k=m[
s]+g)>=0&&k<C*R&&l[s]!=k%C&&(!M[k]||!j&&c>=
16!=M[k]>=16))a[f++]=s;rand();if(f){f=M[e=m
[s=a[rand()%f]]+g];j=j<f?f:j;f+=c&-16*!j;M[
g]=c|1<<s;M[*d++=e]=f|1<<(s+2)%4;}else e=d>
b++?b[-1]:e;}P(" ");P(" ");for(s=C;--s;P("_"
))P("_");for(;P("\n"),R--;P("|"))for(e=C;e--
;P("_"+(*u++/8)%2))P("|_"+(*u/4)%2);}
```

- Quel langage de programmation?
- Que fait ce programme?

Un programme intelligent


```
jesuis(X) :- jepense(X).

jepense(marc).
jepense(line).

?- jesuis(Y).
Y = marc ;
Y = line ;
no
```

- Quel langage de programmation?

Langages de programmation

© 2011 Marc Feeley

- Les **langages de programmation** jouent un rôle fondamental en informatique
- Tout système informatique doit être **réalisé** à l'aide d'un (ou plusieurs) langage(s)
- Peut-on se contenter d'apprendre **un ou deux** langages de programmation? **Non!**
  - L'industrie utilise plusieurs langages, et il y en a de plus en plus (parfois privés)
  - Certains traitements se font mieux avec certains langages
  - Les approches d'un langage peuvent s'appliquer à d'autres

Aperçu du cours

© 2011 Marc Feeley

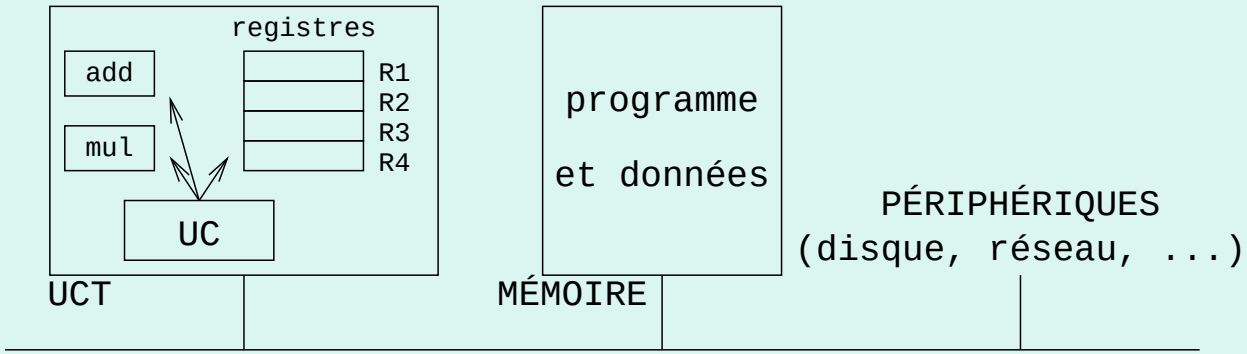
- Il existe un très grand nombre de langages de programmation ($>$ **2000**)
- Les buts du cours sont
  1. de vous permettre d'**utiliser** la majorité de ces langages **efficacement** en peu de temps
  2. d'étudier l'**implantation** des langages de programmation
- C'est possible car les langages partagent beaucoup de **concepts de programmation**
- Des langages représentatifs de ces styles seront utilisés pour étudier les concepts (**Scheme**, **Prolog**, **C**, ...)

Évolution: langage machine (1)

© 2011 Marc Feeley

$\Rightarrow$ Débuts ($<$ 1955): **langage machine**

- Seul langage compris par la machine
- L'unité centrale de traitement (UCT) et la mémoire sont séparées; la mémoire contient le programme et les données à traiter ("architecture de Von Neumann" = même mémoire, "architecture Harvard" = mémoires séparées)



Évolution: langage machine (2)

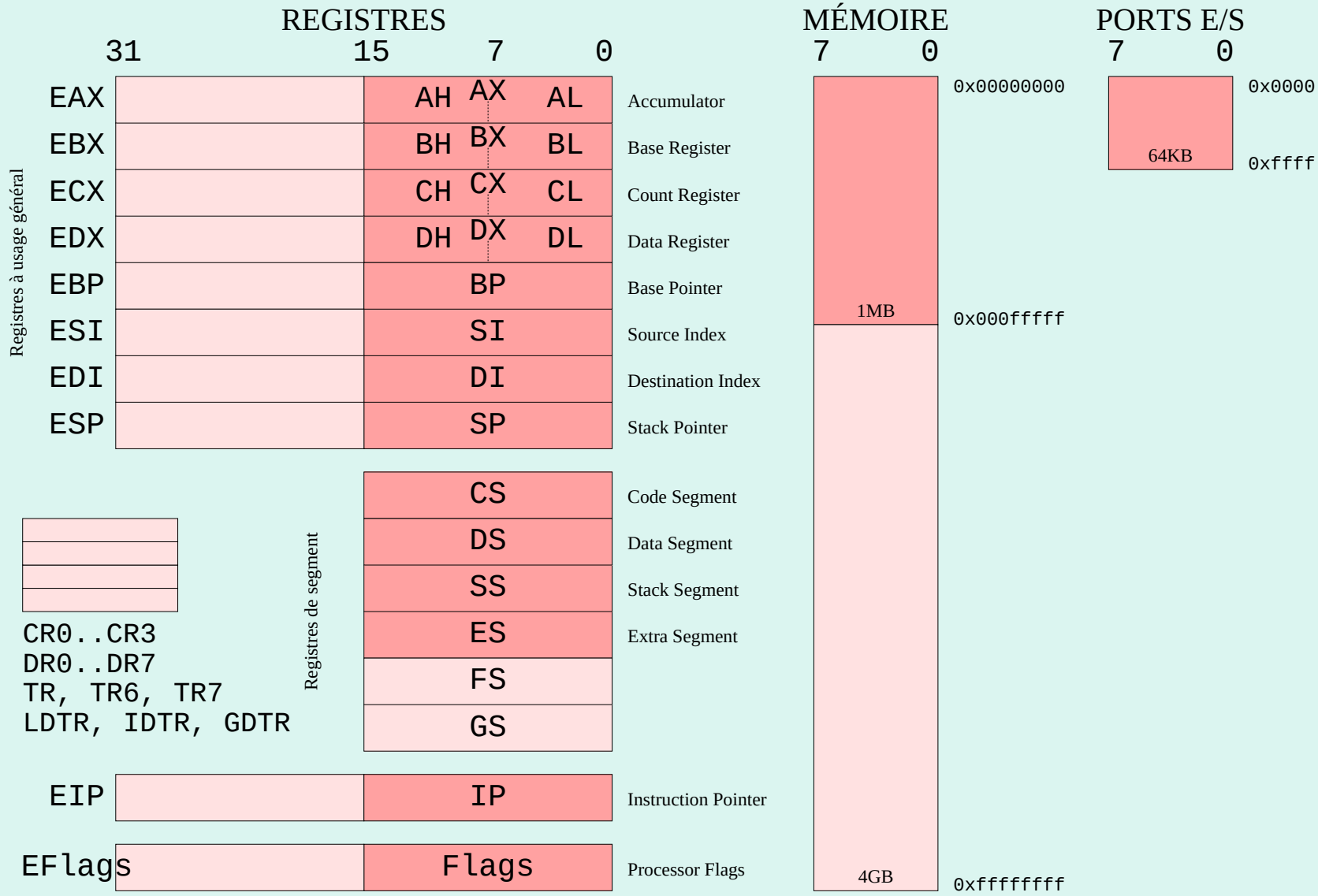
© 2011 Marc Feeley

- Types d'instructions machine:
  - **transfert de donnée** entre registres, mémoire et périphériques
  - **arithmétique** ($+$, $\times$, $/$, ...) sur entiers de taille fixe (8, 16, 32 bits) ou flottants
  - **contrôle** (poursuivre l'exécution à un autre point du programme, conditionnellement ou pas)

Évolution: langage machine (3)

© 2011 Marc Feeley

Exemple concret: famille x86 de Intel



Évolution: langage machine (4)

© 2011 Marc Feeley

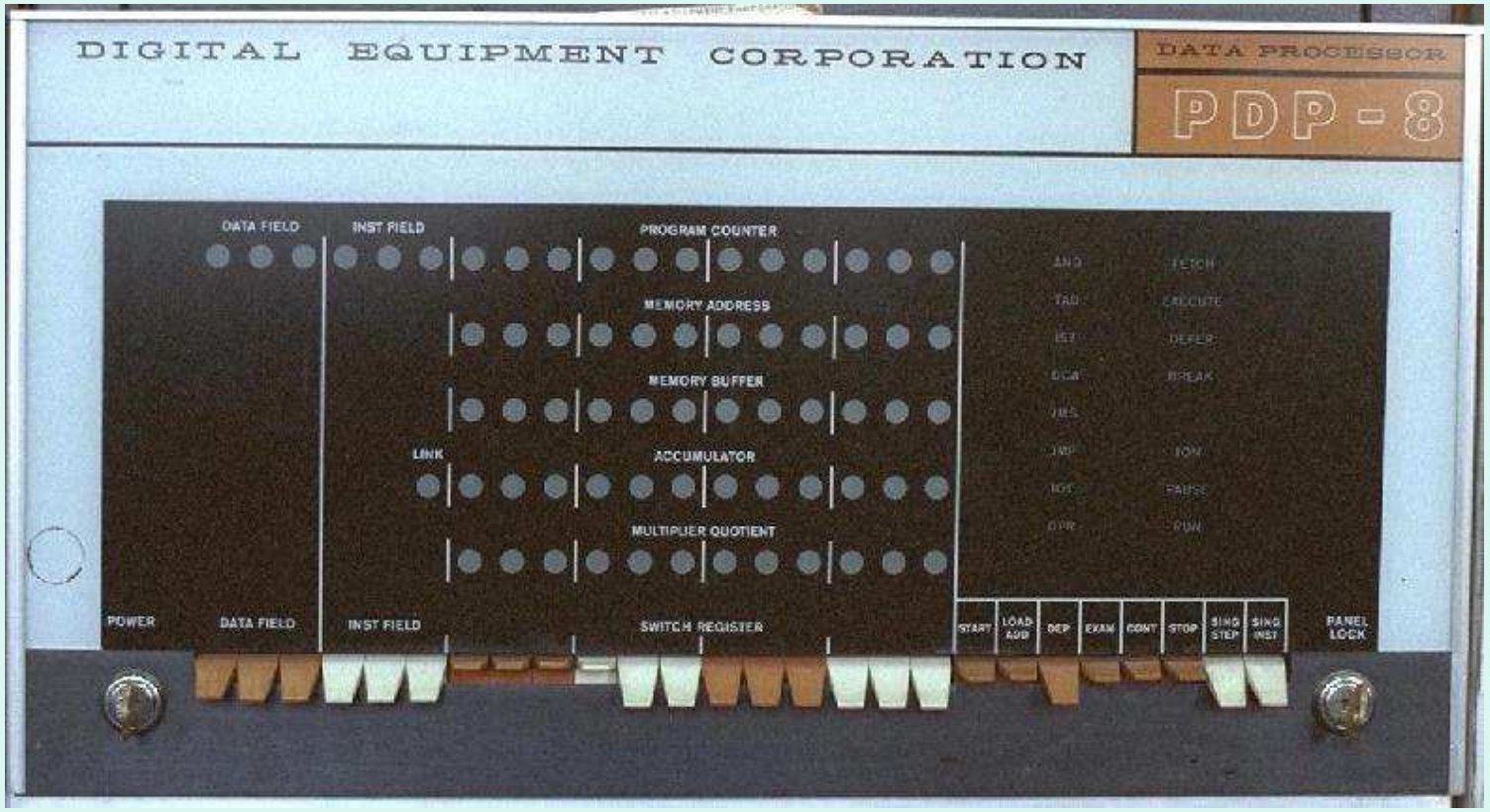
- Les instructions sont **encodées** par des chaînes de bits (typiquement des multiples de 8, 16 ou 32 bits)
- L'UC **lit**, **décode** puis **exécute** les instructions une à une
- Exemple sur famille x86: additionner 2 entiers

```
binaire

10001011   \
01000101    } lire 1er entier dans registre AX
00001010   /
00000011   \
01000101    } lire 2ème entier et ajouter à AX
00010100   /
```

Évolution: langage machine (5)

© 2011 Marc Feeley

- À l'origine il fallait entrer le programme binaire **manuellement** sur le tableau de contrôle:



- Aujourd'hui c'est le rôle du "**bootloader**" du SE

Évolution: langage machine (6)

© 2011 Marc Feeley

- Avantages:
  1. **contrôle total** sur la machine (accès à toutes les possibilités de la machine ce qui est très utile pour l'implantation d'un SE)
  2. possibilité d'**exécution rapide**
- Désavantages:
  1. **inintelligible** pour les humains
  2. le langage **change** d'un type de machine à l'autre (il faut donc récrire le programme en entier pour pouvoir l'utiliser sur un autre type de machine)

Évolution: langage d'assemblage (1)

© 2011 Marc Feeley

=> **langage d'assemblage** ("assembler")

- Essentiellement une **représentation textuelle symbolique du langage machine**
- On utilise des **mnémoniques** pour nommer les instructions ("mov", "add", ...) et on peut **nommer** les emplacements mémoire et constantes (avec des identificateurs)
- Même exemple:

```
mov X(%ebp),%ax  # lire 1er entier dans AX
add Y(%ebp),%ax  # lire et ajouter 2ème entier
```

Évolution: langage d'assemblage (2)

© 2011 Marc Feeley

- Un peu plus lisible mais on a toujours:
  - **codage peu efficace**: beaucoup de code à écrire pour faire peu de choses
  - **surcharge intellectuelle**: le programmeur doit être constamment conscient des caractéristiques de la machine (nb. registres, taille des registres et mémoire, irrégularités du jeu d'instruction, organisation des pipelines, etc)
- Ce sont des **langages de bas-niveau**: où il faut s'occuper de détails qui ont plus rapport avec la machine qu'avec le calcul à faire

Évolution: langages de haut-niveau (1)

© 2011 Marc Feeley

$=>$ **langage de haut-niveau** (“high-level”)

- Langage qui **facilite la tâche de programmation** en cachant les détails se rapportant à la machine
- Les constructions du langage sont **plus proches conceptuellement** de la structure du calcul
- Même exemple en C:

```
X+Y
```

Évolution: langages de haut-niveau (2)

© 2011 Marc Feeley

Exemple en C et assembleur x86

```
int somme()
{
  int s = 0;
  int i;

  for (i=0;i<10;i++)
    s = s + t[i];

  return s;
}
```

```
    .text
    .globl somme
somme:
    xorl %eax,%eax
    xorl %edx,%edx
    movl $t,%ecx
boucle:
    addl (%ecx,%edx,4),%eax
    incl %edx
    cmpl $9,%edx
    jle  boucle
    ret
```

- Note: le code de droite est celui généré par le compilateur C `gcc` (avec `gcc -S somme.c`); il aurait fallu un temps considérable pour un bon programmeur pour l'écrire à la main

Évolution: langages de haut-niveau (3)

© 2011 Marc Feeley

- Avantages:
  1. **notation plus familière** (facilite écriture, analyse, compréhension, modification et entretien du programme)
  2. **fiabilité** (réduit le nombre de bogues)
     - Dijkstra: "*Il sagit d'organiser les calculs de telle manière à ce que nos moyens limités nous permettent d'en garantir l'effet.*"
     - Hoare: "*Il y a deux façons de construire un logiciel: avec un design si simple qu'il n'y a évidemment pas de fautes et avec un design si compliqué qu'il n'y a pas de fautes évidentes. La première méthode est de loin la plus difficile.*"

Évolution: langages de haut-niveau (4)

© 2011 Marc Feeley

- Avantages:

  3. **portabilité** (habileté d'un programme d'être utilisé sur d'autres machines avec peu de changements)
  4. **réutilisation** de librairies de code
  5. **détection automatique d'erreurs**

- Tout cela contribue à augmenter la **productivité** des programmeurs (réduction des coûts pour réaliser/modifier/entretenir un logiciel)

Évolution: langages de haut-niveau (5)

© 2011 Marc Feeley

- **Tout est relatif**: un langage qui en 1960 était de “haut niveau” peut bien paraître de “bas niveau” aujourd’hui:
  machine < **assembleur** < **FORTRAN** < **C** < **Java**
- Le **niveau d’abstraction** est un espace à plusieurs dimensions (un langage peut être de plus haut niveau qu’un autre sur un certain point mais de plus bas niveau sur un autre point):

	gestion mémoire	**point flottants**
Java	automatique	précision fixe
Ada	manuelle	précision variable

Types de langages

© 2011 Marc Feeley

- Les langages de programmation sont conçus pour un certain **domaine d'application**
  - **Usage général ("general purpose"):** Java, C, Pascal
  - **Usage spécialisé:**
    - **Calcul scientifique:** Fortran, MATLAB
    - **Traitement administratif:** COBOL, RPG
    - **Traitement de texte:** Icon, sed
    - **Scripts:** Perl, VisualBASIC, sh
    - **Mise-en-page:** Postscript, TeX
    - **Enseignement:** BASIC, Logo
    - **Description de matériel:** VHDL, Verilog
    - ...

Il y a > 2000 langages

© 2011 Marc Feeley

- **Acronymes:** FORTRAN, Lisp, BASIC, APL, CSP, Prolog, COBOL, Algol, SNOBOL, VHDL, Perl, Tcl, Dylan
- **Lettres:** B, C, D, E, J, Q, Z, Beta
- **Savants et célébrités:** Pascal, Ada, Eiffel, Haskell, Curry, Miranda, Turing, Erlang
- **Pays, animaux, minéraux, végétaux:** Java, Python, Ruby, Oak, Maple
- **Autres:** Scheme, Forth, Postscript, Smalltalk, Actors, Self, SIMULA, MODULA, Oberon, Icon, ABC, Logo, Six, ...

Styles de programmation

© 2011 Marc Feeley

- Déf: le style de programmation c'est la **façon d'exprimer le calcul**

- Exemple: calcul de $n! = 1 \times 2 \times \ldots \times n$ en C

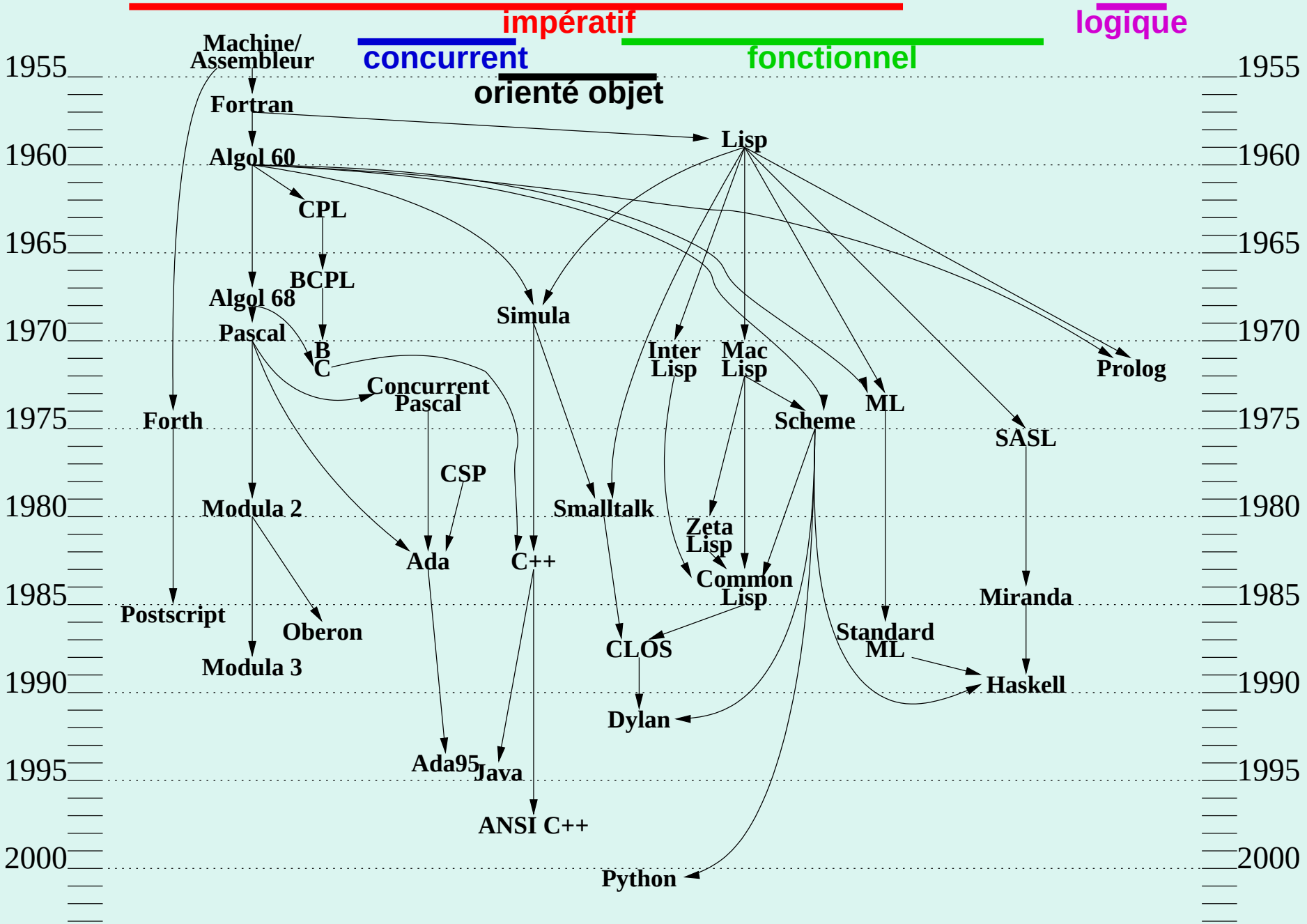
```
/*style impératif*/

int i, f;

f = 1;
i = 2;
while (i <= n)
 {
  f = f*i;
  i = i+1;
 }
```

```
/*style fonctionnel*/

int fact(int n)
{
 if (n == 0)
  return 1;
 else
  return n * fact(n-1);
}
```

- Un langage peut être "pur" ou permettre plus d'un style (et en favoriser certains)

Styles de programmation principaux

© 2011 Marc Feeley



Programmation impérative (1)

© 2011 Marc Feeley

- Calcul = **séquence d'actions**
- Analogie: **recette de cuisine**
- Exemple: lire X, lire Y, calculer X*Y, mettre résultat dans Z
- **1957** : **FORTRAN** (FORmula TRANslation), populaire rapidement car plus facile que l'assembleur et code presqu'aussi rapide
- **1960** : **Algol 60** (ALGOrithmic Language), surtout utilisé pour publier des algorithmes, a donné lieu à la "famille ALGOL"
- **1964** : **BASIC**, premier langage pour l'enseignement, interactif

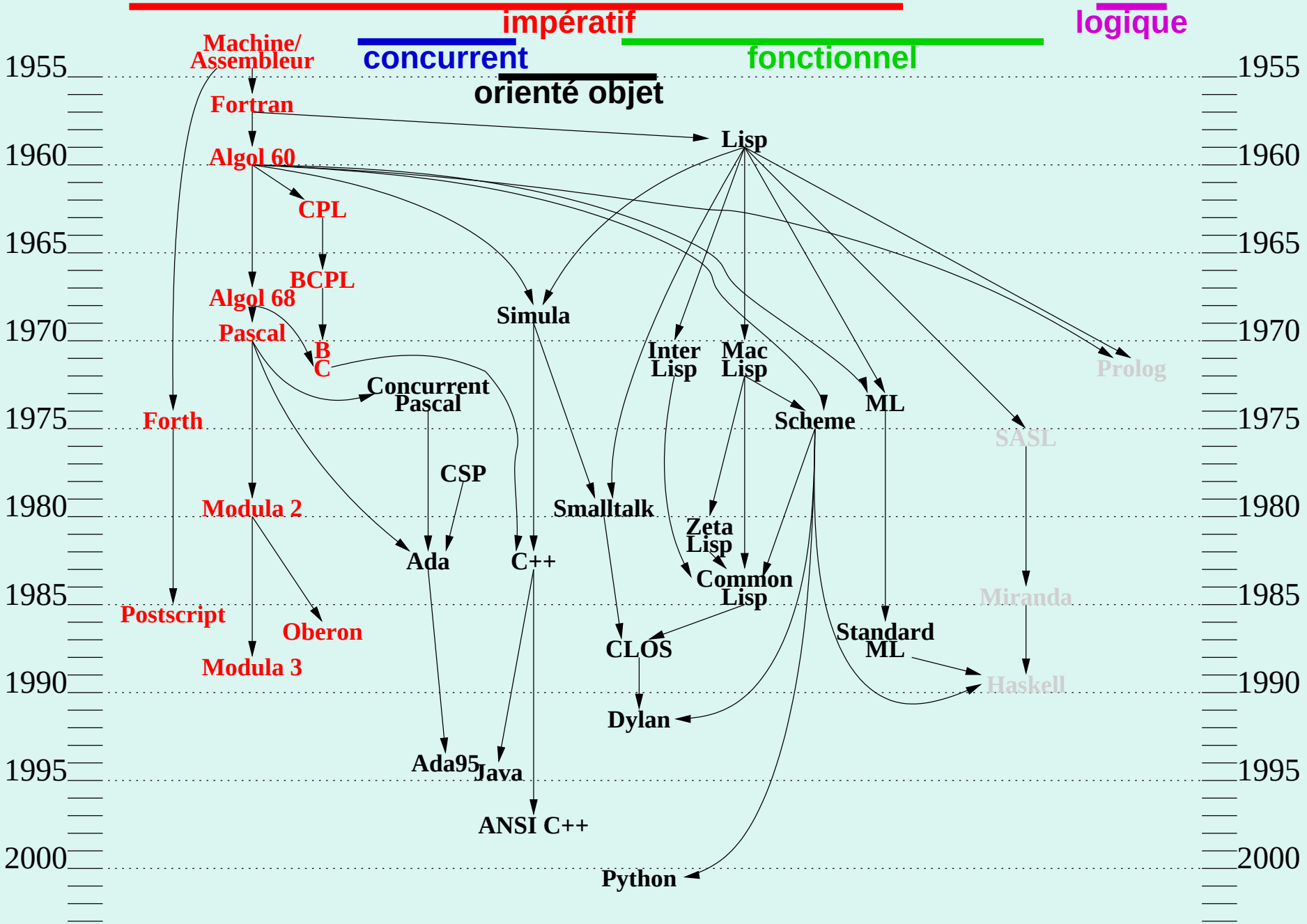
Programmation impérative (2)

© 2011 Marc Feeley

- **1970** : **Pascal**, sous-ensemble de Algol, très facile à compiler, Wirth créa ensuite Modula-2 et Oberon
- **1972** : **B** et **C**, assez bas niveau pour écrire un système d'exploitation (créés pour implanter UNIX)
- **1975** : Bill Gates écrit le premier **interprète BASIC** pour micro-ordinateur et sa compagnie "Traf-O-Data" est renommée "Micro-Soft"
- **1983** : **Ada**, créé pour remplacer tous les langages utilisés par le DoD, fusion de plusieurs langages

Programmation impérative (3)

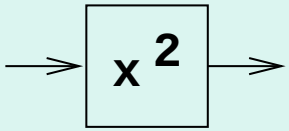
© 2011 Marc Feeley

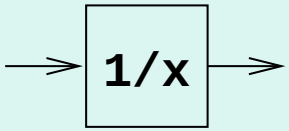


Programmation fonctionnelle (1)

© 2011 Marc Feeley

- Calcul = **fonction (au sens mathématique)**





- Analogie: **définition mathématique**
- Un programme peut être construit en composant des fonctions plus primitives
- **1959** : **Lisp** (LISt Processor), conçu pour le traitement de listes, surtout utilisé en IA, premier langage avec fonctions récursives, fonctions d'ordre supérieur, gestion automatique de la mémoire, expression conditionnelle, ...

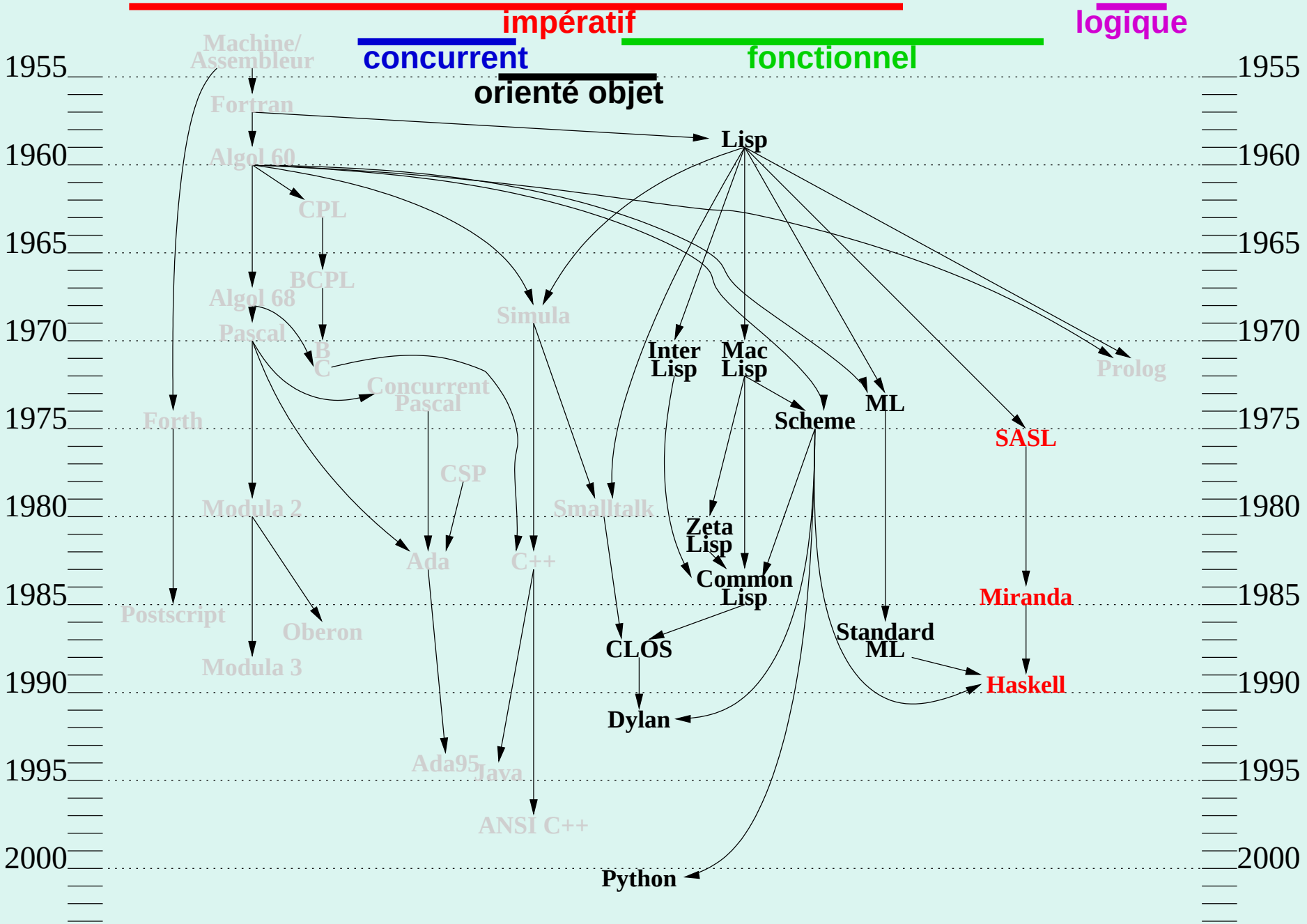
Programmation fonctionnelle (2)

© 2011 Marc Feeley

- **1975** : **Scheme**, variante épurée de Lisp avec liaison lexicale, simple et rapide
- **1984** : **Common Lisp**, fusion de plusieurs Lisps, énorme
- **1974** : **ML**, syntaxe à la Algol + typage statique
- **1990** : **Haskell**, fonctionnel “pur”

Programmation fonctionnelle (3)

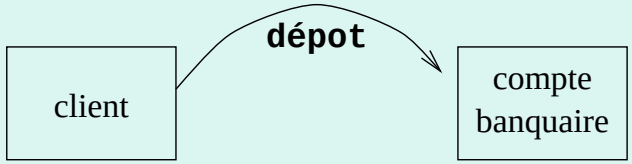
© 2011 Marc Feeley



Programmation orienté objet (1)

© 2011 Marc Feeley

- Calcul = **objets qui interagissent en s'envoyant des messages**



- Analogie: **recette de cuisine "modulaire"**
- Ce style est presque toujours combiné avec la programmation impérative
- **1968** : **SIMULA**, conçu pour la simulation, introduit le concept de "classe", coroutines
- **1980** : **Smalltalk**, typage dynamique, tout est un objet, messages entre objets

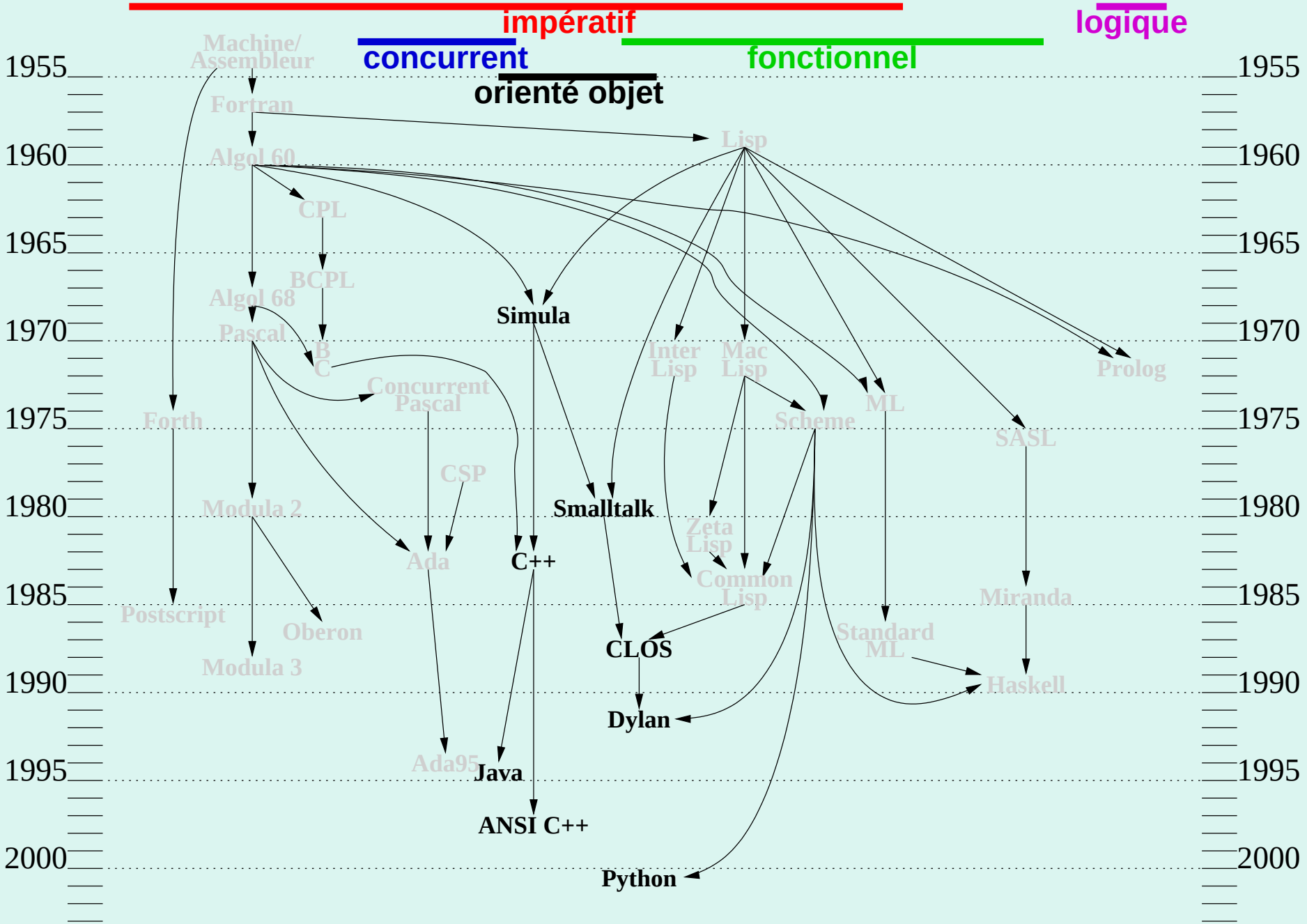
Programmation orienté objet (2)

© 2011 Marc Feeley

- **1983** : **C++**, extension stricte de C avec aspects OO de SIMULA
- **1995** : **Java**, variante épurée de C++, sécuritaire et portable, gestion mémoire automatique

Programmation orienté objet (3)

© 2011 Marc Feeley



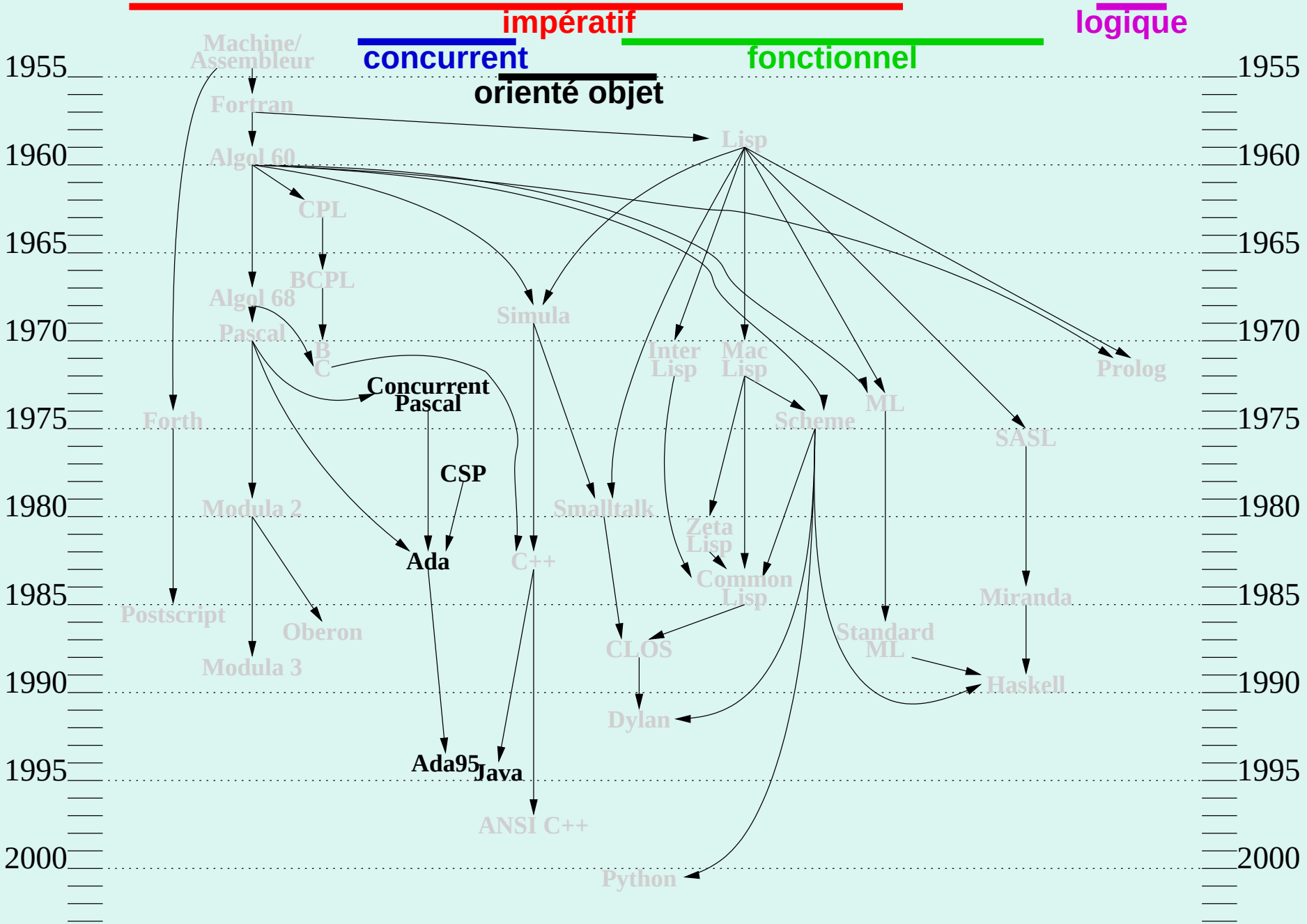
Programmation concurrente (1)

© 2011 Marc Feeley

- Calcul = **plusieurs tâches qui s'exécutent simultanément en coordonnant leurs actions**
- Analogie: **fourmilière**
- Cela permet le calcul distribué (p.e. WWW) et parallèle (p.e. prévision météo)

Programmation concurrente (2)

© 2011 Marc Feeley



Programmation logique (1)

© 2011 Marc Feeley

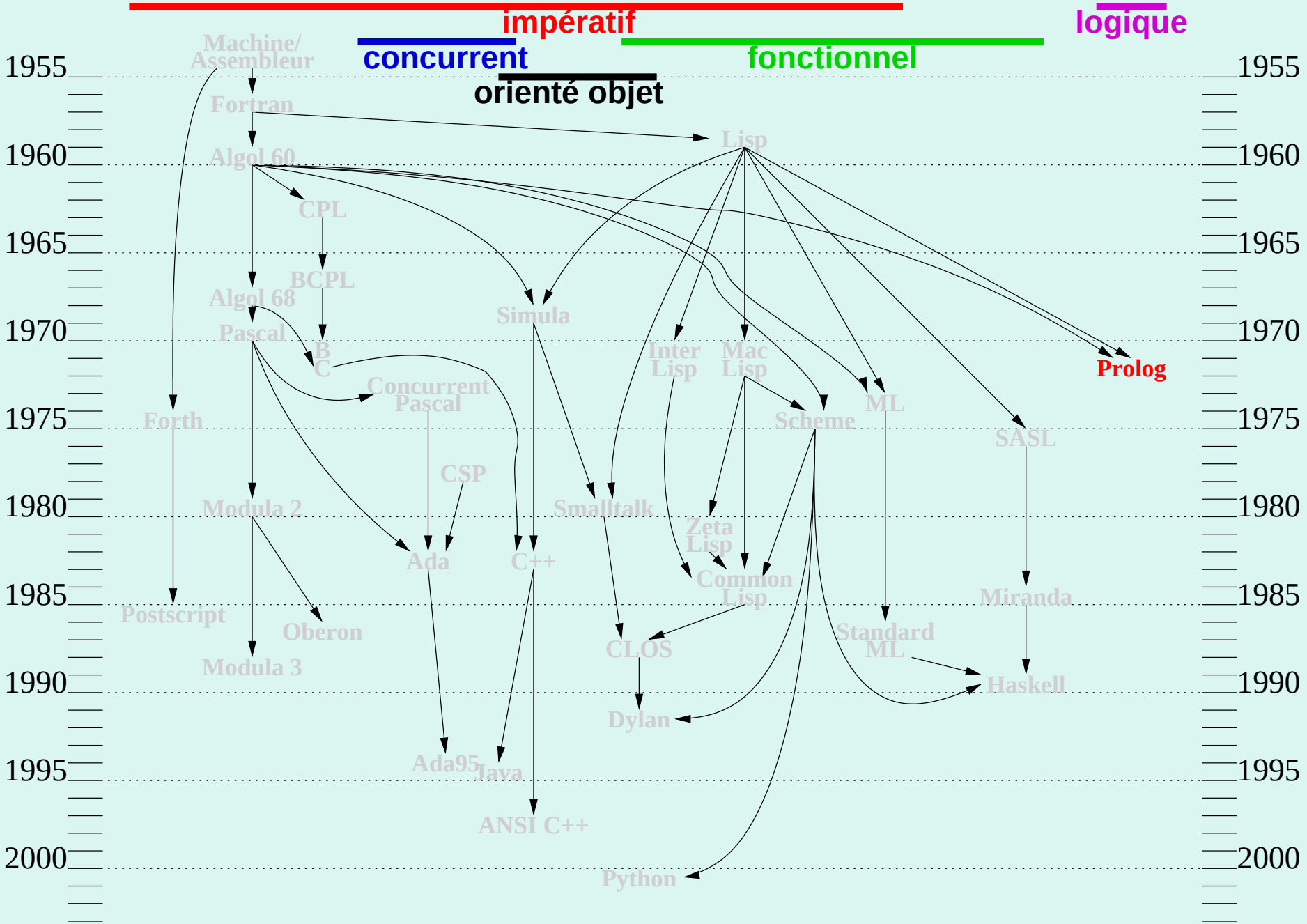
- Calcul = **preuve logique**

```
  César est humain
  Tous les humains sont mortels
--------------------------------
  César est mortel
```

- **1972** : **Prolog** (PROgrammation LOGique), créé pour le traitement de la langue naturelle, applications en IA

Programmation logique (2)

© 2011 Marc Feeley



Choix d'un langage de programmation (1)

© 2011 Marc Feeley

Facteurs qui influencent le choix d'un langage pour une certaine tâche:

- **Niveau d'abstraction**
  - doit être adapté à la tâche (p.e. C vs Prolog pour implanter un SE)
- **Simplicité conceptuelle**
  - nombre et régularité des concepts
  - plus le langage est simple, plus c'est facile de le maîtriser (p.e. C vs C++, Common Lisp vs Scheme)

Choix d'un langage de programmation (2)

© 2011 Marc Feeley

- **Puissance expressive**
  - supporte plusieurs styles de programmation (p.e. C vs Python)
- **Degré de spécialisation**
  - un langage spécialisé est conçu pour faciliter des traitements spécifiques (p.e. SPSS=stat, Tcl/Tk=GUI, Postscript=dessin, Icon=texte, Perl=script, COBOL=administration)
  - un langage général (p.e. C, Ada, ML) peut faire n'importe quoi mais avec plus d'efforts de programmation

Choix d'un langage de programmation (3)

© 2011 Marc Feeley

- **Fiabilité**
  - le langage aide-t-il à prévenir les erreurs? (p.e. programmation structurée)
  - détection des erreurs? à la compilation? à l'exécution? (p.e. C vs Lisp)
- **Support de gros projets**
  - le langage offre-t-il des mécanismes pour aider la réalisation de gros projets ($>$ 10,000 lignes)?
  - modules? compilation séparée? (p.e. Pascal vs Ada)

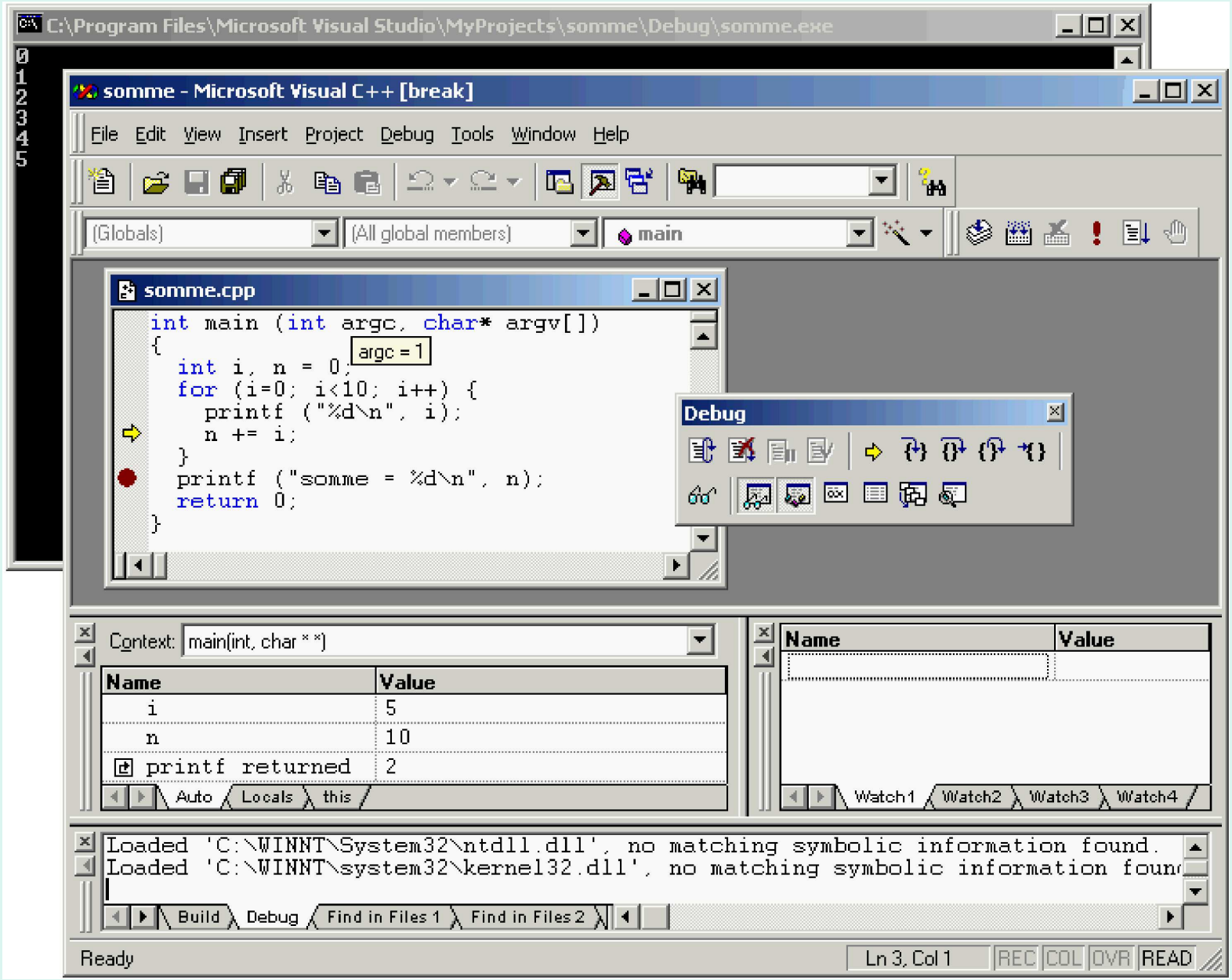
Choix d'un langage de programmation (4)

© 2011 Marc Feeley

- **Environnement de développement**
  - existe-t-il des outils de développement adaptés au langage?
  - éditeur avec indentation automatique et coloration syntaxique
  - compilateur avec messages précis
  - débogueur symbolique avec exécution pas-à-pas
  - exemple: "Integrated Development Environment" (IDE) de Microsoft Visual C/C++

Choix d'un langage de programmation (5)

© 2011 Marc Feeley



Choix d'un langage de programmation (6)

© 2011 Marc Feeley

- **Performance**
  - compilateur rapide? exécutable rapide? choix d'optimisations (p.e. C++ vs Java)
- **Coûts de développement et maintenance**
  - les programmeurs disponibles maîtrisent-ils le langage? sont-ils productifs? le langage permet-t-il de réutiliser et étendre des librairies et programmes existants? (p.e. C vs Lisp) le langage incite-t'il une bonne programmation? (p.e. APL et FORTH sont "write-only")

Choix d'un langage de programmation (7)

© 2011 Marc Feeley

- **Portabilité**
  - le langage est-il répandu? existe-t-il un standard bien établi? (p.e. C vs C#)

Choix d'un langage de programmation (8)

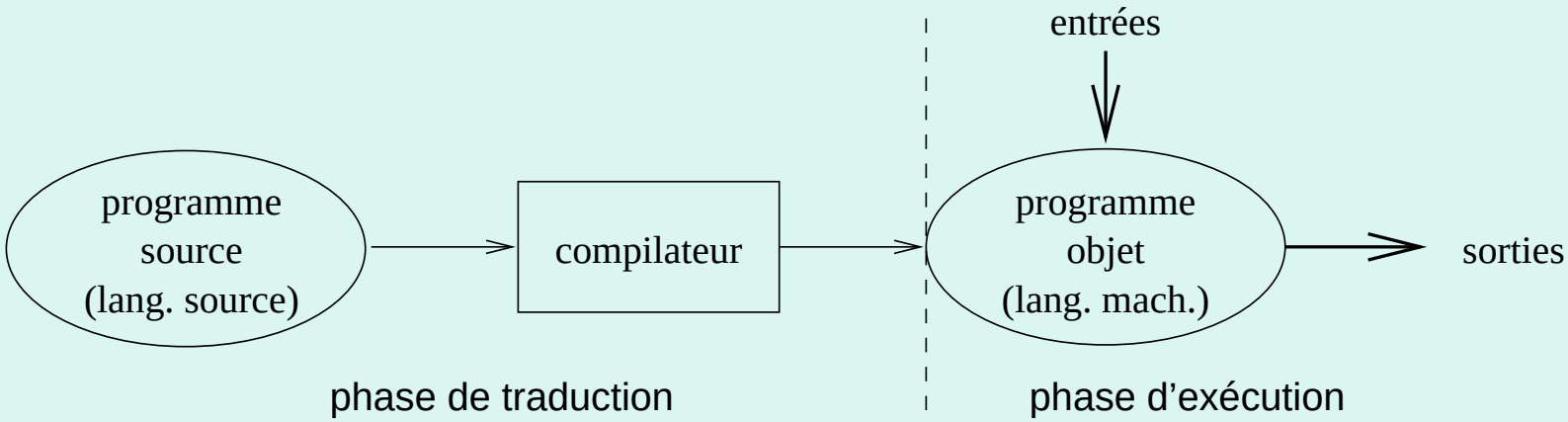
© 2011 Marc Feeley

- Note 1: on fait souvent face à des **compromis** car ces facteurs sont souvent contradictoires (p.e. Java est de haut niveau, de haute puissance expressive, sécuritaire et portable, mais pas très performant)
- Note 2: parfois il est possible d'écrire un programme en utilisant **plus qu'un langage**, ce qui permet d'utiliser le langage le plus approprié pour chaque partie (p.e. avec la venue des architectures orienté-objet COM et CORBA il est maintenant facile d'écrire des "composantes" réutilisables et d'y faire appel à partir de n'importe quel langage)

Implantation des langages de programmation (1)

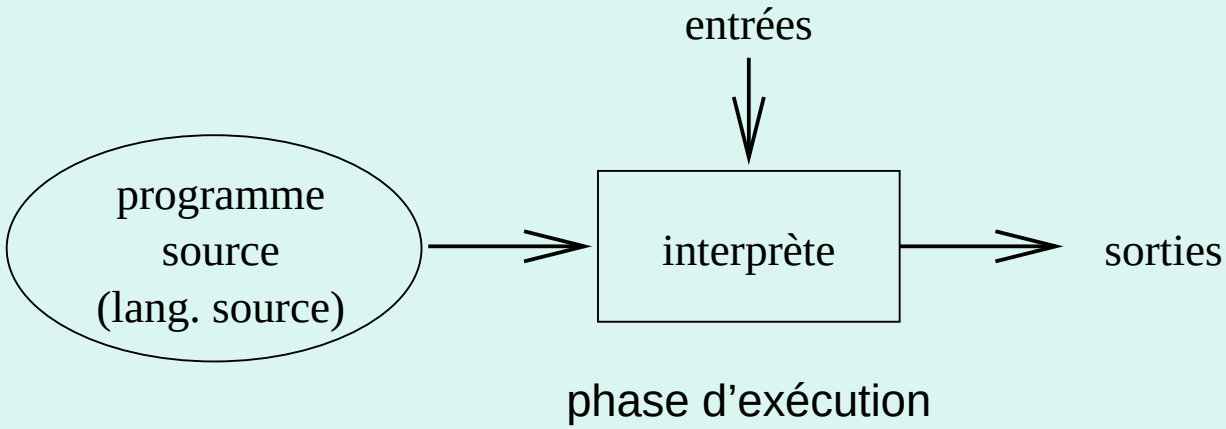
© 2011 Marc Feeley

- **Compilation**: 2 phases
  1. **traduction** du programme source en un programme objet équivalent en langage machine
  2. **exécution** du programme objet



Implantation des langages de programmation (2)

- **Interprétation**: 1 seule phase
  1. **exécution** du programme source, instruction par instruction (lecture, décodage, exécution)



Implantation des langages de programmation (3)

© 2011 Marc Feeley

C++ compilé comparé au “shell script” interprété :

Fichier: "somme.cc"

```
#include <iostream>
using namespace std;
main ()
{ long s = 0;
  for (int i=1; i<=10000000; i++)
    s = s + i;
  cout << s << endl;
}
```

Fichier: "somme.sh"

```
#! /bin/sh
(( s = 0 ))
for (( i=1 ; i<=10000000 ; i++ )) do
  (( s = s + i ))
done
echo $s
```

```
$ time g++ somme.cc -o somme
        0.27 real         0.21 user         0.04 sys
$ time ./somme
50000005000000
        0.03 real         0.02 user         0.00 sys
$ time ./somme.sh
50000005000000
      153.37 real       141.36 user        11.53 sys
$ ls -l somme somme.sh
-rwxr-xr-x  1 feeley  feeley  9688 Jan 11 17:30 somme
-rwxr-xr-x  1 feeley  feeley    91 Jan 11 17:33 somme.sh
```

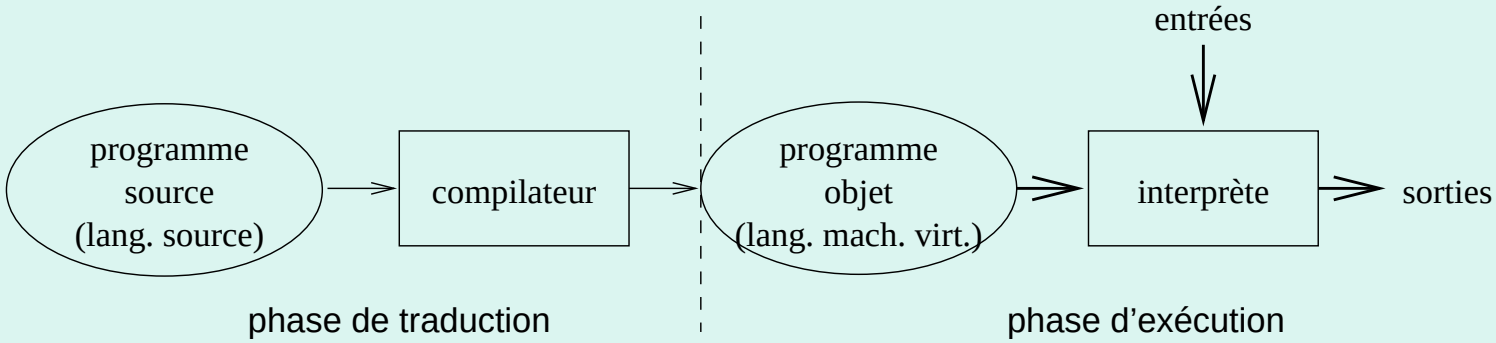
Implantation des langages de programmation (4)

© 2011 Marc Feeley

- Avantage de le **compilation**: **exécution plus rapide** si le programme est exécuté plusieurs fois ou contient des boucles
- Avantages de l'**interprétation**:
  - Plus **compact** (taille des programmes)
  - Plus **flexible** (facile d'étendre le langage et de l'intégrer à d'autres logiciels)
  - Plus **portable** (l'interprète peut être implanté dans un langage de haut-niveau sans dépendances machine)
  - Interaction usager plus **prompte** (temps entre l'entrée d'une commande et son exécution)

Implantation des langages de programmation (5)

- Un interprète implante en quelquesorte une “**machine virtuelle**” (son langage machine = langage source de l’interprète)
- **Approche hybride**: compilation vers machine virtuelle est très portable et moyennement rapide (p.e. JVM)



Syntaxe et sémantique (1)

© 2011 Marc Feeley

- Un langage (naturel ou de programmation) est défini par sa **syntaxe** et sa **sémantique**
- **Syntaxe:** apparence textuelle
- **Sémantique:** sens attaché au texte
- En Allemand:
  - **Ich liebe Dich.** sens = je t'aime
  - **Liebe Ich Dich.** pas de sens

Syntaxe et sémantique (2)

© 2011 Marc Feeley

- Exemple 1: en C
  1. `X = 4+1;` est un énoncé valide
  2. `X+4 = 1;` n'est pas un énoncé valide

- Exemple 2: ces énoncés C et Pascal ont le même sens mais pas la même syntaxe
  1. `if (X == 0) Y = 5; /*C*/`
  2. `if X = 0 then Y := 5 (*Pascal*)`

- Exemple 3: l'expression `6+4/3` n'a pas le même sens en C (résultat = 7), Pascal (résultat = 7.3333), Six (résultat = 22/3) et Smalltalk (résultat = 3.3333)

Spécification et implantation (1)

© 2011 Marc Feeley

- Il est important de distinguer la **spécification** d'un langage de programmation (p.e. manuel de programmation, description informelle, norme) et une **implantation** du langage (p.e. un compilateur ou interprète particulier)
- Un compilateur est **conforme** à une spécification s'il satisfait toutes les exigences de la spécification (la spécification est le **contrat** que le compilateur respecte (ou non))
- Exemple: spécification **norme ANSI C X3.159-1989**, compilateurs **gcc 2.91.66** et **Microsoft Visual C/C++ 6.0**

Spécification et implantation (2)

© 2011 Marc Feeley

- Normalement une spécification **permet des variantes** pour des raisons d'efficacité et de portabilité du comp. (il y a une plus grande liberté dans les choix d'implantation du comp.)
- Un compilateur qui offre des **extensions** au langage peut rester conforme à la spécification
- Exemple 1:
  - en ANSI C l'ordre d'évaluation des sous-expressions est indéfini
  - `t[f(1)]=g(2)+h(3);` appels à `f`, `g` et `h` dans n'importe quel ordre
  - `i=0; t[i]=i++;` affecte `t[0]` ou `t[1]`

Spécification et implantation (3)

© 2011 Marc Feeley

- Exemple 2:
  - **en ANSI C:** `char` $\subseteq$ `short` $\subseteq$ `int` $\subseteq$ `long`
  - `char` = `short` = `int` = `long` est **possible**
  - **en général** les compilateurs C implantent: `char` $= \{-2^7..2^7-1\}$, `short` $= \{-2^{15}..2^{15}-1\}$, `int` $= \{-2^{31}..2^{31}-1\}$, `long` = `int`
  - **exceptions:** `char` $= \{0..2^8-1\}$ (certains vieux compilateurs C), `short` = `int` = `long` (cc sur Cray T90), `long` $= \{-2^{63}..2^{63}-1\}$ (gcc sur Compaq Alpha)
  - **extensions:** `long long` $= \{-2^{63}..2^{63}-1\}$ (gcc sur toutes les plateformes)

Syntaxes des expressions (1)

© 2011 Marc Feeley

- Il existe 3 notations pour les expressions, qui se distinguent par la position de l'opérateur (p.e. +) par rapport aux opérandes (p.e. E1 et E2)
- **Infixe**: E1 + E2 (op. entre les opérandes)
- **Préfixe**: + E1 E2 (op. avant opérandes)
  - Lisp et appel de fonction
  - `+ * x 2 / 1 x` (préf.) = `x*2+1/x` (inf.)
  - `modulo x y` (préf.) = `x modulo y` (inf.)
- **Postfixe**: E1 E2 + (op. après opérandes)
  - Postscript, FORTH et langage machine
  - `x 2 * 1 x / +` (postf.) = `x*2+1/x` (inf.)
  - `x y lineto` (postf.) = `lineto x y` (préf.)

Syntaxes des expressions (2)

© 2011 Marc Feeley

- Pour les opérateurs **unaires** (i.e. une seule opérande) il faut utiliser une notation **préfixe** ou **postfixe**

- Exemple, “`not`” = négation Booléene
  - `not < x 0` (préf.) = `not x<0` (inf.)
  - `x 0 < not` (postf.) = `not x<0` (inf.)

Syntaxes des expressions (3)

© 2011 Marc Feeley

- La notation infixe est familière mais le sens n'est pas clair dans certains contextes
  - `not x<0` = `(not x)<0` ou `not (x<0)` ?
  - `a+b*c` = `(a+b)*c` ou `a+(b*c)` ?
  - `a-b-c` = `(a-b)-c` ou `a-(b-c)` ?
- Solution: employer des règles supplémentaires
- **Niveau de précéance** d'un opérateur indique dans quel ordre regrouper les sous expressions
  - Typique:

```
  ^      plus haut niveau (en premier)
* /
+ -      plus bas niveau
```

Syntaxes des expressions (4)

© 2011 Marc Feeley

- **Ordre d'associativité** d'un opérateur indique dans quel ordre regrouper les sous expressions lorsque le niveau est le même
  - Typique: `^` droite à gauche
`* / + -` gauche à droite
  - `1+2^3^4*5-6` = `1+2^(3^4)*5-6` = `1+(2^(3^4))*5-6` = `1+((2^(3^4))*5)-6` = `(1+((2^(3^4))*5))-6`

- **Parenthèses** pour forcer un ordre précis
  - Inutile en **préfixe** et **postfixe** si le nombre d'opérandes des opérateurs est connu
  - `x*(y-2)` = `* x - y 2` (préf.)
= `x y 2 - *` (postf.)

Syntaxes des expressions (5)

© 2011 Marc Feeley

C	Scheme	Postscript
`2*(7-4)`	`(* 2 (- 7 4))` **parenthèses requises**	`2 7 4 sub mul` **pas de parenthèses**

- **En C:** les parenthèses sont requises seulement lorsque la priorité des opérateurs le demande
  `2-7*4 = 2-(7*4) = ((2)-((7)*(4)))`
- **En Scheme:** les parenthèses sont requises car en général les opérateurs sont à **arité variable**
  `(- 1 (* 2 3) 4) = (- (- 1 (* 2 3)) 4)`

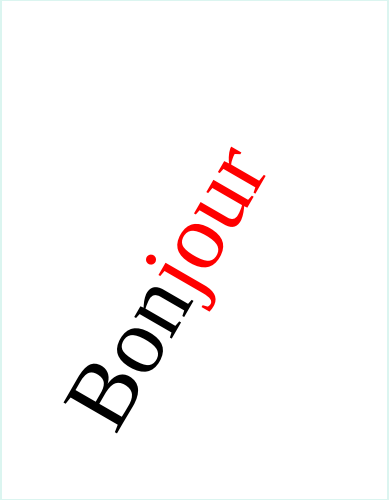
Syntaxes des expressions (6)

© 2011 Marc Feeley

C	Scheme	Postscript
- 5	(- 5) ou -5	5 neg ou -5
7.5 / 2	(/ 7.5 2)	7.5 2 div
7 / 2	(quotient 7 2)	7 2 idiv
7 % 2	(modulo 7 2)	7 2 mod
sqrt(2)	(sqrt 2)	2 sqrt
pow(10,5)	(expt 10 5)	

- Programme Postscript complet:

```
%!PS
/Times-Roman findfont
150 scalefont setfont
180 100 moveto 60 rotate
(Bon) show
1 0 0 setrgbcolor (jour) show
showpage
```



Syntaxes des expressions (7)

© 2011 Marc Feeley

C a 15 niveaux!

```
15: x(y)  x[y]                  appel de fonction et indexation
14: x++   ++x   x--   --x       post/pré- incré/décré- mentation de x
    ~x                          négation logique bit-à-bit
    !x                          négation Booléene, équivalent à x==0
    &x                          retourne pointeur vers la variable x
    *x                          indirection de pointeur
    -x    +x                    inverse additif et identité
13: x*y   x/y   x%y             multiplication, division, modulo
12: x+y   x-y                   addition, soustraction
11: x<<y  x>>y                  décalage à gauche/droite
10: x<y   x<=y  x>y   x>=y      1 si la relation est vraie, 0 sinon
 9: x==y  x!=y                  1 si la relation est vraie, 0 sinon
 8: x&y                         et logique bit-à-bit
 7: x^y                         ou exclusif logique bit-à-bit
 6: x|y                         ou logique bit-à-bit
 5: x&&y                        et Booléen, équivalent à x?(y!=0):0
 4: x||y                        ou Booléen, équivalent à x?1:(y!=0)
 3: x?y:z                       si x!=0 retourne y sinon retourne z
 2: x=y                         affecte y à x et retourne y
    x+=y  x-=y  x*=y  x/=y      affectations composées: x=x+y, etc
    x%=y  x&=y  x^=y  x|=y
    x>>=y  x<<=y
 1: x,y                         évalue x puis retourne la valeur de y
```

Syntaxes des expressions (8)

© 2011 Marc Feeley

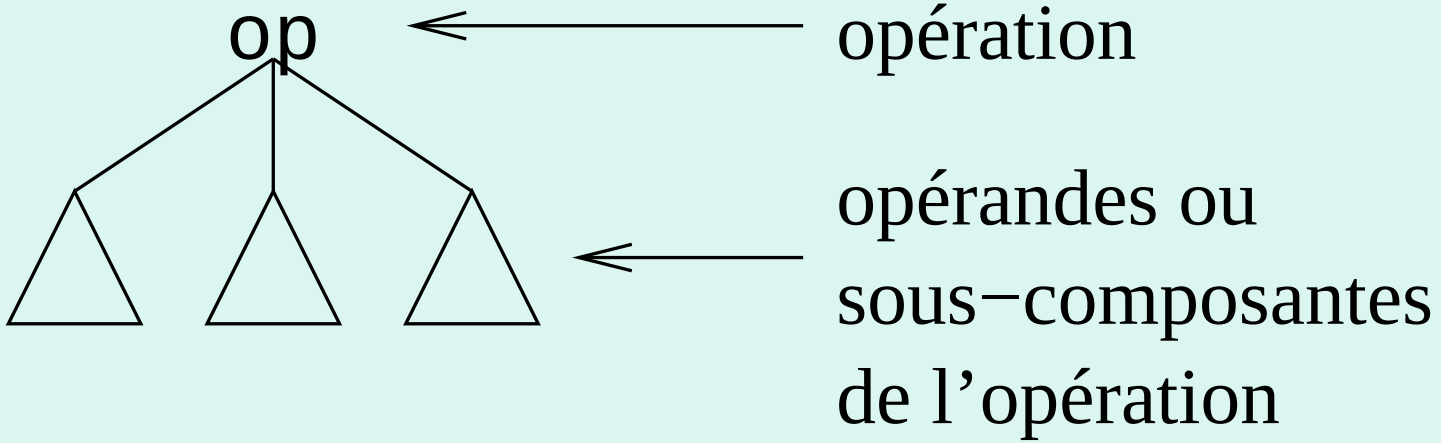
- Les opérateurs de C s'associent de gauche à droite sauf aux niveaux 14, 3 et 2
  - `1-2-3-4 = ((1-2)-3)-4`
  - `*x++ = *(x++)`
  - `a?b:c?d:e = a?b:(c?d:e)`
  - `a=b=c = a=(b=c)`
- Il est difficile de se rappeler de tous les niveaux et certains sont contre intuitifs, par exemple
  - `(x&1 == 0) = (x & (1==0))`
  - morale: utiliser des parenthèses sauf pour les opérateurs les plus communs

ASA: Arbres de syntaxe abstraite (1)

© 2011 Marc Feeley

- Déf: représentation d'un (fragment de) programme sous forme d'arbre qui **souligne sa structure** et **élimine les détails syntaxiques**

- Forme générale:

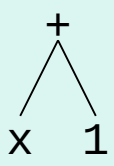


ASA: Arbres de syntaxe abstraite (2)

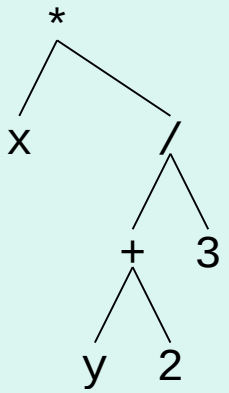
© 2011 Marc Feeley

- Exemples:

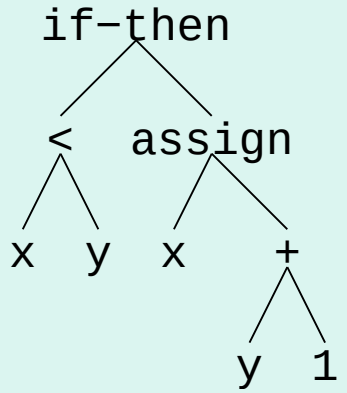
`x+1`



`x*((y+2)/3)`




```
if x<y then x:=y+1
```



- L'ASA ne contient pas les parenthèses, point-virgules, etc car il exprime implicitement les regroupements qu'elles produisent
- Si deux fragments dans 2 langages différents ont le même ASA c'est qu'ils sont "équivalents"
  - Par exemple en C: `if (x<y) x=y+1;`

Définition formelle de la syntaxe (1)

© 2011 Marc Feeley

- Méthode classique = définir une **grammaire** pour le langage
- La grammaire spécifie exactement les programmes (ou “phrases”) qui sont **acceptables syntaxiquement**
- Analogie avec langue naturelle:

```
ce chat est noir        OK
est noir chat ce        pas OK

x := 0 ;                OK
0 ; := x                pas OK
```

- Note: un programme est composé d’une séquence de **symboles** (identificateurs, mot réservés, ponctuation, etc)

Définition formelle de la syntaxe (2)

© 2011 Marc Feeley

- Déf: **Vocabulaire** (Σ) = ensemble de symboles qu'on peut retrouver dans une phrase

  Ex: Σ={0,1} Σ={+,-,a,...,z} Σ={if,and,:=,<,...}

- Déf: **Phrase** = séquence possiblement vide de symboles tirés de Σ

  Ex: Σ={0,1} phrase1=00101 phrase2= (vide)

- Déf: **Langage** = ensemble de phrases, possiblement infini

  Ex1: Σ={0,1} L1={00,01,10,11}
  Ex2: Σ={0,1} L2={1,10,100,1000,...}

Définition formelle de la syntaxe (3)

© 2011 Marc Feeley

- Déf: **Grammaire hors-contexte** en format BNF (Backus-Naur Form) = ensemble de **catégories** et **productions**
  - **Catégorie** = nom qui désigne un type de fragment de phrase, par convention entouré de $<$ ... $>$

    Ex: $<$*expression*$>$ $<$*entier*$>$ $<$*vide*$>$
  - **Production** = règle de la forme

    $<$*cat*$>$ `::=` x_1 x_2 ... x_n

    où $<$*cat*$>$ est une catégorie et x_i est une catégorie ou symbole pour tout i

    Ex: $<$*X*$>$ `::= 0` $<$*Y*$>$ `1`

Définition formelle de la syntaxe (4)

© 2011 Marc Feeley

- Par convention la première production indique la **catégorie de départ**

- Exemple: grammaire G1 =

  ⟨*bin*⟩ ::= 0
  ⟨*bin*⟩ ::= 1
  ⟨*bin*⟩ ::= ⟨*bin*⟩ ⟨*bin*⟩

Définition formelle de la syntaxe (5)

© 2011 Marc Feeley

- Déf: **Dérivation directe** (notation X=>Y)
  Soit $x_1 \ldots x_n$ une chaîne et
  1. x_i est une catégorie ou symbole pour tout i
  2. x_j = <C>
  3. il existe une production <C> ::= $y_1 \ldots y_m$

  alors

  $x_1 \ldots x_{j-1}$ <C> $x_{j+1} \ldots x_n$ => $x_1 \ldots x_{j-1}$ $y_1 \ldots y_m$ $x_{j+1} \ldots x_n$

- Déf: **Dérivation** (d'une phrase) = une séquence de dérivations directes à partir de la catégorie de départ dont le résultat est la phrase en question
  Ex: <bin> => <bin> <bin> => <bin> 0 => 1 0

Définition formelle de la syntaxe (6)

© 2011 Marc Feeley

- Déf: **L(G)** = le langage défini par la grammaire G = ensemble de toutes les phrases dérivables par G
- L(G) = { p | <*départ*> => ... => p }
- Par exemple: L(G1) = { `0`, `1`, `00`, `01`, `10`, `11`, ... } car

  <*bin*> => `0`

  <*bin*> => `1`

  <*bin*> => <*bin*> <*bin*> => <*bin*> `0` => `0 0`

  <*bin*> => <*bin*> <*bin*> => <*bin*> `1` => `0 1`

  <*bin*> => <*bin*> <*bin*> => <*bin*> `0` => `1 0`

  etc

Définition formelle de la syntaxe (7)

© 2011 Marc Feeley

- Exemple: grammaire G2 =

<*flottant*> ::= <*entier*> . <*entier*>
<*entier*> ::= <*chiffre*>
<*entier*> ::= <*chiffre*> <*entier*>
<*chiffre*> ::= 0
<*chiffre*> ::= 1
<*chiffre*> ::= 2
<*chiffre*> ::= 3
<*chiffre*> ::= 4
<*chiffre*> ::= 5
<*chiffre*> ::= 6
<*chiffre*> ::= 7
<*chiffre*> ::= 8
<*chiffre*> ::= 9

Définition formelle de la syntaxe (8)

© 2011 Marc Feeley

- Rappel: grammaire G2 =

  <*flottant*> ::= <*entier*> . <*entier*>
  <*entier*> ::= <*chiffre*>
  <*entier*> ::= <*chiffre*> <*entier*>
  <*chiffre*> ::= 0
  ...

- 1.5 dans L(G2)? oui:

  <*flottant*> => <*entier*> . <*entier*> => <*chiffre*> . <*entier*>
  => 1 . <*entier*> => 1 . <*chiffre*> => 1 . 5

  ou bien

  <*flottant*> => <*entier*> . <*entier*> => <*entier*> . <*chiffre*>
  => <*chiffre*> . <*chiffre*> => <*chiffre*> . 5 => 1 . 5

- .5 dans L(G2)? non

Arbre de dérivation (1)

© 2011 Marc Feeley

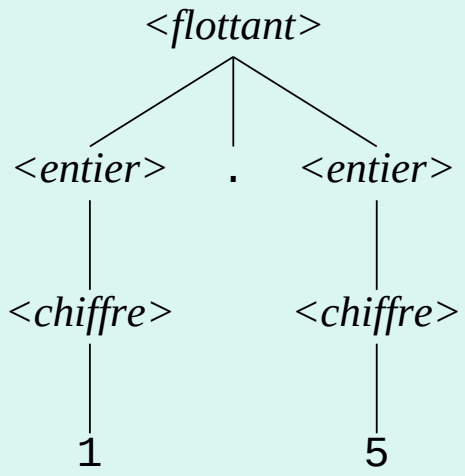
- Utile pour représenter la **structure syntaxique** d'une phrase
- Étant donné une certaine dérivation d'une phrase P, l'arbre de dérivation correspondant aura la catégorie de départ à sa racine, des symboles aux feuilles, et chaque noeud interne est une catégorie qui a comme enfants la partie droite de la production qui l'a remplacée dans la dérivation
- Les feuilles de l'arbre de dérivation = la phrase
- L'arbre de dérivation **ne représente pas l'ordre** dans lequel les dérivations sont effectuées

Arbre de dérivation (2)

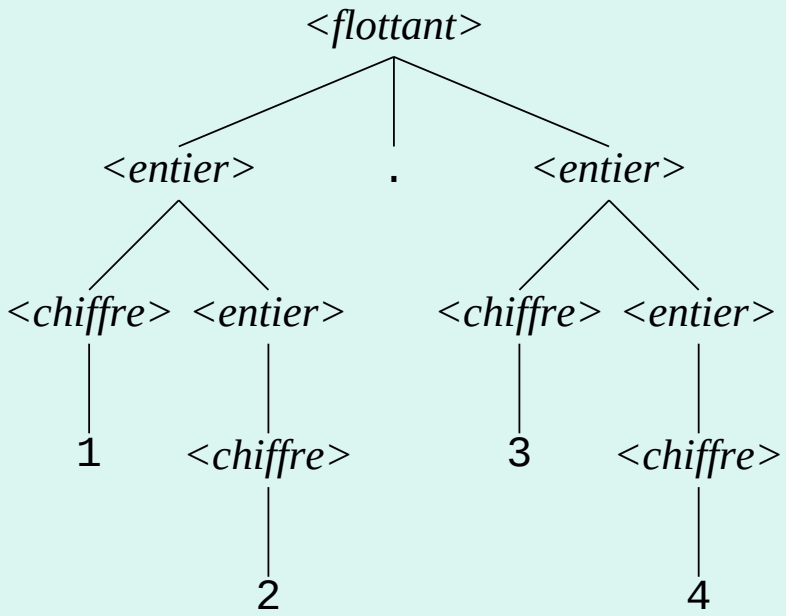
© 2011 Marc Feeley

- Exemples avec grammaire G2

`1.5`



`12.34`



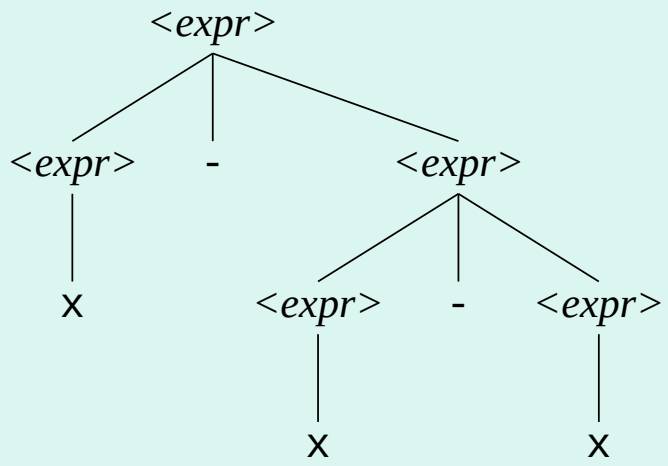
<flottant> => <entier> . <entier> => <chiffre> . <entier>
=> 1 . <entier> => 1 . <chiffre> => 1 . 5

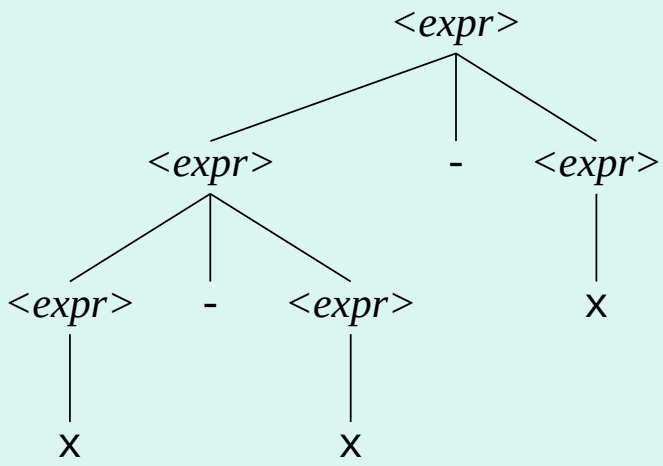
<flottant> => <entier> . <entier> => <entier> . <chiffre>
=> <chiffre> . <chiffre> => <chiffre> . 5 => 1 . 5

Grammaires ambiguës (1)

© 2011 Marc Feeley

- Déf: Une grammaire G est **ambiguë** ssi il existe une phrase P dans L(G) tel que P a **plus qu'un arbre de dérivation**
- Exemple de grammaire ambiguë: G3 =
  <*expr*> ::= x
  <*expr*> ::= <*expr*> - <*expr*>
- 2 arbres de dérivation pour: x - x - x





Grammaires ambiguës (2)

© 2011 Marc Feeley

- Les grammaires ambiguës sont à éviter car normalement le compilateur utilise l'arbre de dérivation pour **attacher un sens au programme**

- Si la grammaire est ambiguë, le compilateur doit utiliser d'autres règles pour retirer les ambiguïtés

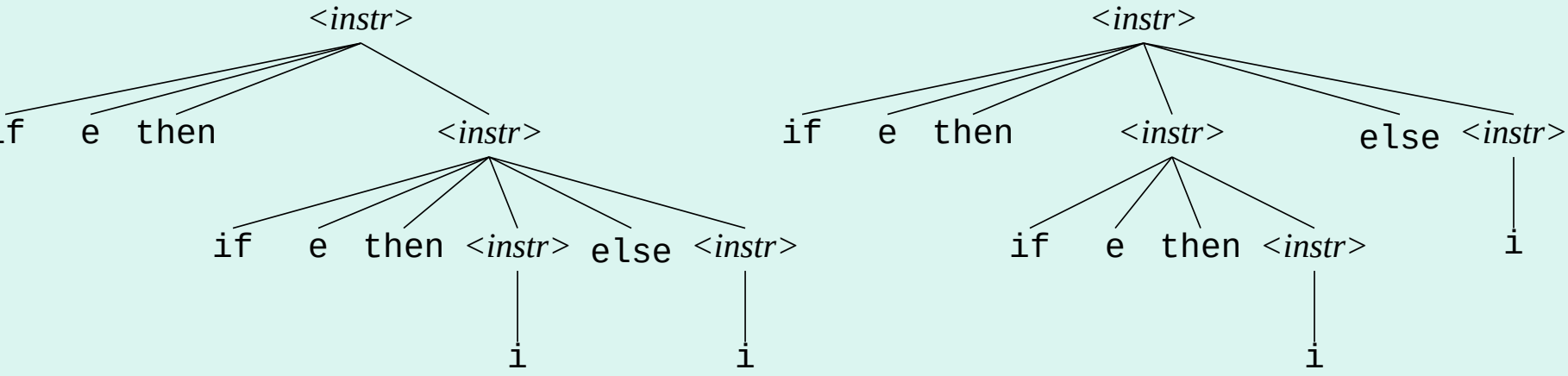
Grammaires ambiguës (3)

© 2011 Marc Feeley

- Cas classique, le **“else” pendant**: gram. G4 =

  <*instr*> ::= i
  <*instr*> ::= if e then <*instr*>
  <*instr*> ::= if e then <*instr*> else <*instr*>

  if e then if e then i else i



Grammaires ambiguës (4)

© 2011 Marc Feeley

- Solution typique (Algol60/Pascal/C): règle “`else` va avec le `if` le plus proche”
- Algol68/Modula-2/Ada/Fortran77 évitent ce problème en demandant un `fi`/`END`/`end if`/`endif` à la fin d’un `if`
- Grammaire G5 =

<*instr*> `::= i`

<*instr*> `::= if e then` <*instr*> `fi`
<*instr*> `::= if e then` <*instr*> `else` <*instr*> `fi`

```
if e then if e then i else i fi fi
if e then if e then i fi else i fi
```

Autres formalismes syntaxiques (1)

© 2011 Marc Feeley

- Déf: **BNF étendu (EBNF)** = BNF + expressions régulières
- Notation permise dans les productions:
  - `[ X ]` $->$ X est optionnel
  - `{ X }` $->$ répétition de X (≥ 0 fois)
  - `X | Y` $->$ choix entre X et Y
  - `( X Y Z )` $->$ groupement
- Exemple: gram. G6 =

<*flottant*> ::= <*entier*> . { <*chiffre*> } [<*exp*>]

<*entier*> ::= <*chiffre*> { <*chiffre*> }

<*chiffre*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*exp*> ::= E [+ | -] <*entier*>

Autres formalismes syntaxiques (2)

© 2011 Marc Feeley

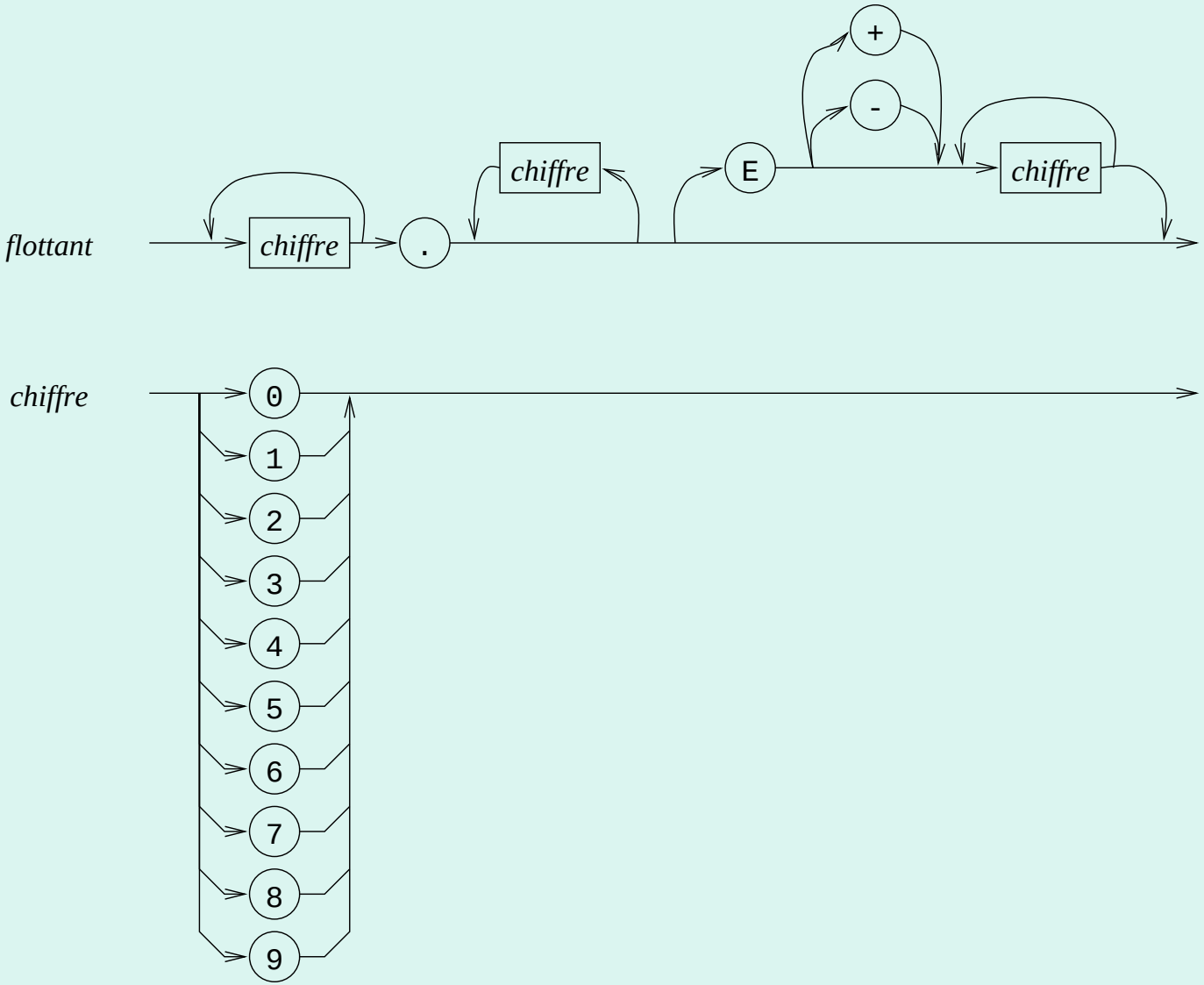
- Déf: **Diagramme syntaxique** = représentation graphique d'une grammaire avec 1 graphe orienté par catégorie (les noeuds des graphes sont soit des catégories (dans des rectangles) ou symboles du vocabulaire (dans des ovales))

- Langage = ensemble de tous les chemins possible dans le graphe qui traversent la catégorie de départ

Autres formalismes syntaxiques (3)

© 2011 Marc Feeley

- Exemple: équivalent de la grammaire G6



Programmation impérative / procédurale

© 2011 Marc Feeley

- Style de programmation offert par la plupart des langages de programmation (machine, assembleur, FORTRAN, Algol, SIMULA, C/C++, Pascal, Ada, Java, Lisp, ML, ...)

- Basé sur le modèle d'une **machine**; le programme indique la **séquence d'opérations** que la machine doit suivre pour faire le calcul

- Le programmeur dicte précisément chaque étape que la machine doit effectuer pour réaliser le calcul

Points de contrôle

© 2011 Marc Feeley

- Déf: **point de contrôle** = un emplacement du programme entre deux opérations

```
void main ()  /*programme C*/

{ char c;
①
  c = getchar ();
②
  c = toupper (c);
③
  putchar (c);
④
}
```

- Note: dépendant du point de vue certains énoncés du programme peuvent donner lieu à plus d'un point de contrôle si ils font plus qu'une opération (p.e. `c=toupper(c);`)

Exécution d'un programme (1)

© 2011 Marc Feeley

- Déf: l'**état** de l'exécution d'un programme (si on suppose une seule entrée et sortie et pas d'appel de procédure) =
  1. **point de contrôle** où se trouve l'exécution
  2. **contenu** de toutes les variables et structures de données
  3. partie de l'**entrée** qui reste à lire et **sortie** produite jusqu'à date

```
void main ()  /*programme C*/
{ char c;
①
  c = getchar ();
②
  c = toupper (c);
③
  putchar (c);
④
}
```

EXÉCUTION

pt.	variables	entrée	sortie
1	c=?	ab	
2	c='a'	b	
3	c='A'	b	
4	c='A'	b	A

Exécution d'un programme (2)

© 2011 Marc Feeley

- L'état change après l'exécution de chaque opération (donc revenir à un état précédent $=>$ boucle infinie)
- L'**ordre des opérations** est important

```
/* ce programme n'est pas équivalent */

c = getchar ();
putchar (c);
c = toupper (c);
```

- Pourquoi? Parce que l'effet de chaque opération dépend de l'état du programme **au moment de son exécution**

Énoncés de contrôle

© 2011 Marc Feeley

- Normalement, après l'exécution d'un énoncé, le point d'exécution se trouve juste avant le prochain énoncé (i.e. exécution **séquentielle**)
- Pour un ordre d'exécution autre que séquentiel, il faut utiliser des **énoncés de contrôle**
- Exemple:

```
①
if (x < 0)
 ②
  x = 0;
else
 ③
  x = x*2;
④
```

L'ordre d'exécution sera 1-2-4 ou 1-3-4 dépendant de la valeur de la variable x

Énoncé de contrôle “goto”

© 2011 Marc Feeley

- Énoncé de contrôle **le plus primitif**; correspond directement avec les instructions de branchement en langage machine; offert par FORTRAN, SIMULA, C, Pascal, ...
- Change le point de contrôle de façon arbitraire
- La destination est indiquée par une **étiquette**

```
     x = 1;
     goto fin;
deb: x = x*2;
fin: if (x<10) goto deb;
     printf ("%d\n", x);
```

```
x = 1;
while (x<10)
  x = x*2;
printf ("%d\n", x);
```

- Problème: difficile de comprendre un programme à base de `goto` car la séquence d’opérations obtenue n’est pas évidente

Programmation structurée (1)

© 2011 Marc Feeley

- Afin d'éviter le spaghetti les langages modernes prônent la **programmation structurée**
  - idée: la structure syntaxique d'un programme doit **aider à comprendre** ce que fait le programme (en particulier au niveau du séquencement des opérations)
- Déf: le **flux de contrôle** c'est la progression du point d'exécution pendant l'exécution (la séquence de points de contrôle atteints)

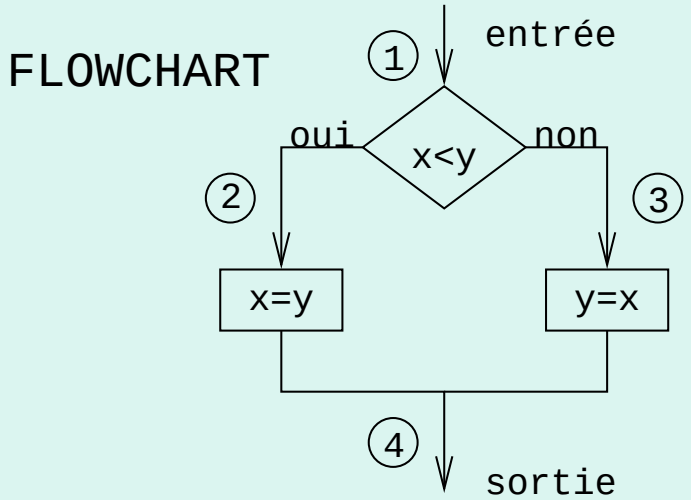
Programmation structurée (2)

© 2011 Marc Feeley

- Déf: un programme est **structuré** si le flux de contrôle est directement lié à sa structure syntaxique

- En général, chaque énoncé (simple ou composé) a **un seul** point de contrôle d'entrée et **un seul** point de contrôle de sortie

```
①
  if (x < y)
   ②
    x = y;
  else
   ③
    y = x;
④
```



Énoncé “bloc” (1)

© 2011 Marc Feeley

- Exécution séquentielle d’un groupe d’énoncés
- Exemple en Pascal:

```
begin x:=x+1; y:=y+1 end
```

- Exemple en C:

```
{ int t; t=x; x=y; y=t; }
```

- Certains langages (p.e. C et SIMULA) permettent d’introduire des **variables locales** au bloc; en général au début seulement, C++ et Java sont des exceptions notables:

```
{ x++; int y=x/2; z=y*y; }
```

Énoncé “bloc” (2)

© 2011 Marc Feeley

- Le point-virgule est un **séparateur** d’énoncés dans certains langages (p.e. Pascal et SIMULA) et un **terminateur** d’énoncés dans d’autres (p.e. C, C++ et Java)

- En Pascal ce bloc a 3 énoncés:

```
begin x:=x+1; y:=y+1; end   (*3ème = vide*)
```

- En Pascal ceci est illégal:

```
if x < y then
  x := y;
else          (* erreur: else inattendu *)
  y := x;
```

- “;” comme **terminateur est plus intuitif**

Énoncé conditionnel (1)

© 2011 Marc Feeley

- La plupart des langages offrent deux formes:
  - `if` <*expr*> `then` <*énoncé*>
  - `if` <*expr*> `then` <*énoncé*> `else` <*énoncé*>
- Cette syntaxe est ambiguë
- La syntaxe Modula-2 (EBNF):

```
IF <expr> THEN <séq_énoncés>
{ ELSIF <expr> THEN <séq_énoncés> }
[ ELSE <séq_énoncés> ]
END
```

Énoncé conditionnel (2)

© 2011 Marc Feeley

- Les cascades de `IF`s ne sont pas élégants si on n'utilise pas `ELSIF`:

```
(* sans ELSIF *)

IF c1 THEN
  e1
ELSE
  IF c2 THEN
    e2
  ELSE
    IF c3 THEN
      e3
    ELSE
      e4
    END
  END
END
```

```
(* avec ELSIF *)

IF c1 THEN
  e1
ELSIF c2 THEN
  e2
ELSIF c3 THEN
  e3
ELSE
  e4
END
```

Énoncé conditionnel (3)

© 2011 Marc Feeley

- Certains langages (p.e. SIMULA, C, C++ et Java) offrent également des **expressions conditionnelles**:
  - SIMULA: `if` *<expr1>* `then` *<expr2>* `else` *<expr3>*
  - C/C++/Java: *<expr1>* `?` *<expr2>* `:` *<expr3>*
- Exemple, calcul de la valeur absolue de `x`:

```
x := if x<0 then -x else x;  ! SIMULA ;

x = (x<0) ? -x : x;          /* C */
```

Énoncé conditionnel (4)

© 2011 Marc Feeley

- En C, toute expression suivie d'un point-virgule est un **énoncé** (à noter que C traite <*var*>=<*expr*> comme une expression)

- Cela permet d'utiliser les opérateurs "&&", "||" et "," pour faire des **traitements conditionnels**:

```
rappel:

    X && Y   -->  0 si X==0 sinon Y!=0
    X || Y   -->  1 si X!=0 sinon Y!=0
    X , Y    -->  Y  (après avoir évalué X)

x<0 && (x=-x);      /* x = abs (x); */

x==5 || (x=y=0, z=1);

    /* if (x!=5) { x=0; y=0; z=1; } */
```

Énoncés `while` et `repeat` (1)

© 2011 Marc Feeley

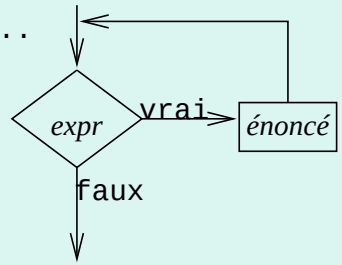
- Nombre d'itérations **pas connu à l'avance**
- Pascal: `while` <*expr*> `do` <*énoncé*>
  - <*expr*> évaluée **avant** chaque itération et après dernière itération
  - ≥ 0 itérations
- Pascal: `repeat` <*séq_énoncés*> `until` <*expr*>
  - <*expr*> évaluée **après** chaque itération
  - ≥ 1 itérations
  - équivalent:

    <*séq_énoncés*>`;`
    `while not` <*expr*> `do begin` <*séq_énoncés*> `end;`
  - variante C: `do` <*énoncé*> `while ( `<*expr*>` );`

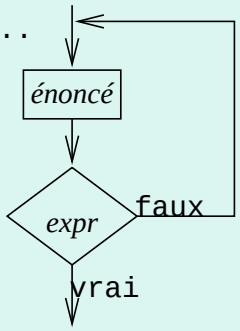
Énoncés `while` et `repeat` (2)

© 2011 Marc Feeley

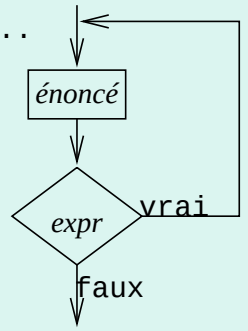
`while ... do ...`



`repeat ... until ...`



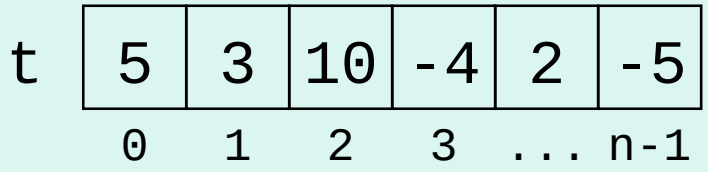
`do ... while ...`



Énoncé `break` de C (1)

© 2011 Marc Feeley

- `break;` => quitter la **boucle courante**
- Exemple: imprimer la première valeur négative dans un tableau `t` d'entiers de longueur `n`




```
1.  i = 0;
    while (i < n && t[i] >= 0) i++;
    if (i < n) printf ("%d", t[i]);

2.  for (i=0; i<n; i++)
      if (t[i] < 0)
        { printf ("%d", t[i]); break; }

3.  t[n] = -1;  /* placer une sentinelle */
    i = 0;
    while (t[i] >= 0) i++;
    if (i < n) printf ("%d", t[i]);
```

Énoncé `break` de C (2)

© 2011 Marc Feeley

- Le `break` de C ne s'applique qu'à la **boucle englobante la plus proche**

```
while (...)              /* boucle 1 */
 {
  while (...)            /* boucle 2 */
   {
    ... break; ...       /* quitte boucle 2 */
   }
 }
```

```
while (...)              /* boucle 1 */
 {
  while (...)            /* boucle 2 */
   {
    ... goto fin; ... /* quitte boucle 1 */
   }
 }
fin:
```

Énoncé `exit` de Ada

© 2011 Marc Feeley

- Ada permet de nommer les boucles et offre une variante de `break`:

  `exit [` <*nom_de_boucle*> `] [when` <*expr*> `];`

```
boucle1:
  while x<10 loop                      -- boucle 1
    ...
    boucle2:
      while x<5 loop                   -- boucle 2
        ...
        if x<0 then exit end if; -- quitte boucle 2
        exit when x<0;           -- quitte boucle 2
        exit boucle1;            -- quitte boucle 1
        exit boucle1 when x<0;   -- quitte boucle 1
        ...
      end loop;
    ...
  end loop;
```

Énoncé `continue` de C

- `continue;` => passer à **prochaine itération**
- Exemple: copier l'entrée à la sortie en enlevant les lignes vides
  - sous UNIX, fin de ligne = caractère `'\n'`
  - idée: ne pas copier `'\n'` précédé d'un `'\n'` ou au début

```
int c, fdl;

fdl = TRUE;
while ((c = getchar ()) != EOF)
  { if (c == '\n')
      { if (fdl) continue;
        fdl = TRUE;
      }
    else
      fdl = FALSE;
    putchar (c);
  }
```

Énoncés `break` et `continue` de Java

© 2011 Marc Feeley

- Java n'a pas le `goto` mais permet de nommer les énoncés et de faire

  `break` [<*nom_d'un_énoncé_englobant*>]
  `continue` [<*nom_d'une_boucle_englobante*>]

```
un:
  for (int x=0; x<10; x++)
    {
      ...
      deux:
        while (i<x)
          {
            ...
            trois:
              {
                ...
                break;           // quitte la boucle while
                break un;        // quitte la boucle for
                break trois;     // quitte le bloc
                continue un;     // passe au prochain x
                continue trois; // illegal
                ...
              }
            ...
          }
      ...
    }
```

Énoncé `return` de C, Java et Ada

© 2011 Marc Feeley

- `return;` => quitter immédiatement la **procédure courante**
- `return` <*expr*>`;` => quitter immédiatement la **fonction courante**
- Exemple: obtenir la position dans un tableau ou se trouve un élément particulier

```
int position (int element)
{ int i = 0;
  while (i < n)
    { if (t[i] == element) return i;
      i++;
    }
  return -1; /* indiquer échec */
}
```

Retour de fonction en Pascal, SIMULA et Modula-2

© 2011 Marc Feeley

- Dans le corps, le nom de la fonction dénote **une variable contenant le résultat de la fonction**

```
function cube (x:integer) : integer;
begin
  cube := x*x*x
end

function position (element:integer) : integer;
var i : integer;
begin
  i := 0;
  while i < n do
    begin
      if t[i] = element then
        begin
          position := i;
          goto fin
        end;
      i := i+1
    end;
  position := -1;
  fin:
end
```

Énoncé `loop` de Ada et Modula-3 (1)

© 2011 Marc Feeley

- `loop` <*séq_énoncés*> `end loop;`
- C/Java: `for (;;)` <*corps_de_boucle*>
- Donne une **boucle infinie**, sauf si sortie prématurée (par exemple avec `exit` ou `break`)
- Exemples:

```
-- Ada

loop
  exit when x > 10;
  x := x*2;
end loop;
```

```
(* en Pascal sans goto *)

while x <= 10 do
  x := x*2
```

Énoncé `loop` de Ada et Modula-3 (2)

© 2011 Marc Feeley


```
-- Ada

loop
  x := x*2;
  exit when x > 10;
end loop;


loop
  x := read (f);
  x := x*x;
  exit when x > 10;
  write (x);
end loop;
```

```
(* en Pascal sans goto *)

repeat
  x := x*2
until x > 10


x := read (f);
x := x*x;
while x <= 10 do
  begin
    write (x);
    x := read (f);
    x := x*x
  end
(* duplication de code
   pas élégant *)
```

Énoncés de boucle défini (1)

© 2011 Marc Feeley

- Nombre d'itérations **connu à l'avance**
- Pascal:

  `for` <*var*> `:=` <*expr1*> (`to`|`downto`) <*expr2*> `do`
  <*énoncé*>

  $\Rightarrow$ les bornes doivent être de type entier ou énuméré

- Ex.: `for i := x+1 to x*x do writeln (i)`
- Une implantation possible de `for`:

```
(* cas ``to'' *)
LIM := <expr2>;
<var> := <expr1>;
while <var> <= LIM do
  begin
    <enoncé>;
    <var> := <var>+1
  end
```

```
(* cas ``downto'' *)
LIM := <expr2>;
<var> := <expr1>;
while <var> >= LIM do
  begin
    <enoncé>;
    <var> := <var>-1
  end
```

Énoncés de boucle défini (2)

© 2011 Marc Feeley

- Pour permettre plus de liberté à l'implanteur du compilateur, en Pascal:
  - il est **interdit de modifier** <*var*> tant que la boucle n'est pas finie
  - la valeur de <*var*> est **indéfinie** à la sortie de la boucle
- Cela admet cette implantation qui est plus performante sur plusieurs machines:

```
(* cas ``to'' *)
LIM := <expr2>;      (* LIM et T stockées dans *)
T := <expr1>;        (* des registres rapides *)
while T <= LIM do
  begin
    <var> := T;
    <enoncé>;        (* modification de <var> ici *)
    T := T+1         (* ne changerais pas le *)
  end                (* nombre d'itérations *)
```

Énoncés de boucle défini (3)

© 2011 Marc Feeley

- Note: à la fin du `for`:
  - Implantation 1: $<var>$ = $<expr2>$+1 (si $\geq$1 itérations) et $<expr1>$ (si 0 itération)
  - Implantation 2: $<var>$ = $<expr2>$ (si $\geq$1 itérations) et valeur avant le `for` (si 0 itération)

Énoncés de boucle défini (4)

© 2011 Marc Feeley

- SIMULA:

  `for` <*var*> `:=` <*item*> { `,` <*item*> } `do` <*énoncé*>
  <*item*> ::= <*exp*> | <*exp1*> `step` <*exp2*> `until` <*exp3*>

- Exemple:
  `for i := 2, 3, 5 step 5 until 30 do f (i)`

- C/Java:
  `for` ([<*exp1*>]; [<*exp2*>]; [<*exp3*>]) <*énoncé*>

- Le `for` de C/Java est presque équivalent à:
  `{ <exp1>; while (<exp2>) { <enoncé>; <exp3>; } }`
  - `<exp2>` est remplacée par `TRUE` si absent
  - C++/Java: `<exp1>` peut déclarer une var.
  - un `continue` saute à <*exp3*>

Énoncés de boucle défini (5)

© 2011 Marc Feeley

- Exemples:

```
for (i=1; i<5; i++) printf ("%d", i);
  /* imprime: 1 2 3 4 */

for (i=1; i<9; i=i*2) printf ("%d", i);
  /* imprime: 1 2 4 8 */

for (p=list; p!=NULL; p=p->next) ...;

for (int i=1; i<5; i++) printf ("%d", i);
  /* imprime: 1 2 3 4 */

for (;i<10;) ...;

for (;;)
  { if ((c = getchar ()) == EOF) break;
    putchar (c);
  }
```

Énoncés de boucle défini (6)

© 2011 Marc Feeley

- Variations sémantiques:
  - **exécution du corps au moins une fois?** (FORTRAN=oui, en général=non)
  - **évaluation unique des bornes?** (Pascal=oui, C=non, SIMULA=oui sauf pour <*exp3*> après le `until`)
  - **progression de la variable de contrôle?** (Pascal=pas de +1 ou -1, C=pas variable, SIMULA=pas quelconque mais constant pour la durée de la boucle)
  - **type de la variable de contrôle?** (Pascal=entier, C=quelconque, SIMULA=entier ou flottant)

Énoncés de boucle défini (7)

© 2011 Marc Feeley

- Variations sémantiques:
  - **déclaration locale de la variable de contrôle?** (Pascal=non, C=non; SIMULA=non, Ada=oui, Modula-3=oui, C++&Java=possible)
  - **variable de contrôle modifiable pendant la boucle?** (Pascal=non, C=oui, SIMULA=oui)
  - **variable de contrôle définie après la boucle?** (Pascal=non, C=oui, SIMULA=oui)

Énoncé de sélection par cas (1)

© 2011 Marc Feeley

- Sélection d'un énoncé parmis plusieurs
- Pascal:

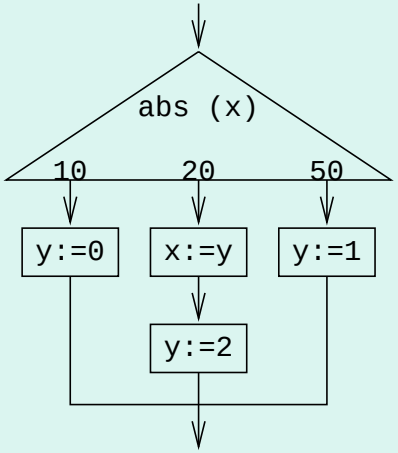
```
case <expr> of
   <constante> { , <constante> } : <énoncé1> ;
   <constante> { , <constante> } : <énoncé2> ;
   ...
end
```

Énoncé de sélection par cas (2)

© 2011 Marc Feeley

- Exemple

```
case abs (x) of
  10: y := 0;
  50: y := 1;
  20: begin x := y; y :=2; end;
end
```



- En général
  - constantes entières ou énumérées seulement
  - constantes dans n'importe quel ordre
  - ≥ 1 constantes par énoncé
  - constantes distinctes

Énoncé de sélection par cas (3)

© 2011 Marc Feeley

- Variations sémantiques:
  - **action lorsque cas pas précisé?** (Pascal=erreur, C=rien, Modula-2=erreur)
  - **action de défaut permise?** (Pascal=pas standard, C=oui, Modula-2=oui)
  - **intervalle de cas permis (p.e. 10..20)?** (Pascal=non, C=non, Modula-2=oui)

Énoncé de sélection par cas (4)

© 2011 Marc Feeley

- C offre: `switch` (<*expr*>) <*énoncé*>

  => <*énoncé*> peut contenir comme sous-énoncés:

  `case` <*constante*> `:` <*énoncé*>

  `default:` <*énoncé*>
  `break;`

- Chaque branche se poursuit à la prochaine branche (il faut utiliser `break;` pour sortir du `switch`)

- Exemple:

```
switch (x)
  {
    case 10: y = 1; break;
    case 20:
    case 30: y = 2; break;
    default: y = 3;
  }
```

Énoncé de sélection par cas (5)

© 2011 Marc Feeley

- La syntaxe de C est une source de bugs:

```
switch (x)
  {
    case 1: y = 1; break;
    defaut: y = 2; /* jamais exécuté */
  }
```

- Il ne faut pas oublier de `break;`!

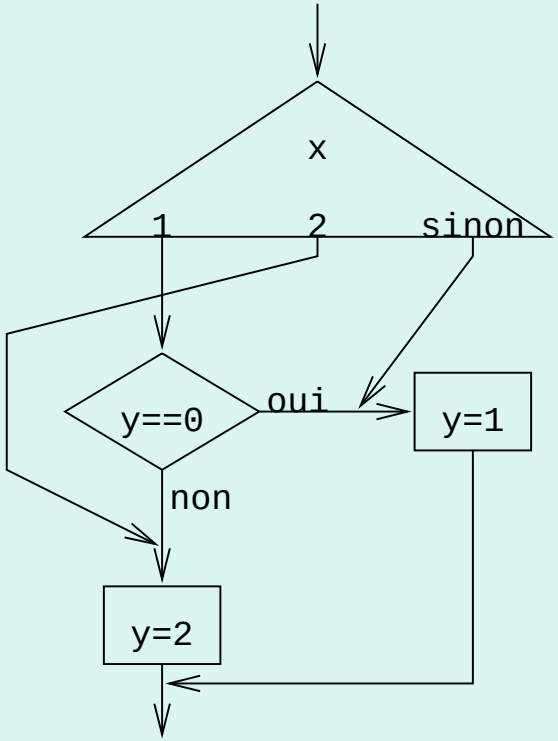
```
switch (x)
  {
    case 1: y = 10;
    case 2: y = 20;
    case 3: y = 30;
  }
```

Énoncé de sélection par cas (6)

© 2011 Marc Feeley

- L'énoncé `switch` n'est pas structuré

```
switch (x)
  case 1:
    if (y==0)
      default: y = 1;
    else
      case 2: y = 2;
```



Énoncé de sélection par cas (7)

© 2011 Marc Feeley

- Cela est parfois utile (**mécanisme de Duff**):

```
void copier (char *src, char *dst, int n)
{
  while (n > 0) { *dst++ = *src++; n--; }
}
```

```
void copier (char *src, char *dst, int n)
{
  switch (n & 3)
    {
      case 0: while (n > 0) { *dst++ = *src++;
      case 3:                 *dst++ = *src++;
      case 2:                 *dst++ = *src++;
      case 1:                 *dst++ = *src++;
                              n -= 4;
                            }
    }
}
```

Énoncé de sélection par cas (8)

© 2011 Marc Feeley

- Cela n'est pas permis en Java, car la syntaxe du `switch` en Java est:

```
<SwitchStatement> ::=
   switch ( <Expression> ) <SwitchBlock>

<SwitchBlock> ::=
   { <SwitchBlockStatementGroups_opt> <SwitchLabels_opt> }

<SwitchBlockStatementGroups> ::=
   <SwitchBlockStatementGroup>
 | <SwitchBlockStatementGroups> <SwitchBlockStatementGroup>

<SwitchBlockStatementGroup> ::=
   <SwitchLabels> <BlockStatements>

<SwitchLabels> ::=
   <SwitchLabel>
 | <SwitchLabels> <SwitchLabel>

<SwitchLabel> ::=
   case <ConstantExpression> :
 | default :
```

Exceptions (1)

© 2011 Marc Feeley

- Une **exception** c'est un évènement exceptionnel qui demande au programme de changer son flux de contrôle normal
- Il y a plusieurs causes d'exceptions:
  - **erreurs de logique:** division par zero, accès hors des bornes d'un tableau, arguments hors domaine (e.g. `sqrt(-1.5)`), débordement, allocation d'un tableau de taille négative, accès par un pointeur nul
  - **manque de ressources:** mémoire épuisée, fichier inexistant, manque de permission
  - **erreurs matérielles:** panne de réseau, erreur de lecture du disque, papier coincé

Exceptions (2)

- Il est important de **prévoir les exceptions** qui peuvent se produire afin de faire un traitement approprié (**robustesse**)
  - message d'erreur + terminaison "propre"
  - demander à l'usager comment continuer
  - nouvel essai de l'opération fautive
  - continuer l'exécution dans un état cohérent

Exceptions (3)

© 2011 Marc Feeley

- L'approche la plus élémentaire pour gérer les exceptions c'est d'utiliser des **codes d'erreurs** (approche adoptée par C)
  - `malloc()` retourne `NULL` si mémoire épuisée
  - `fopen()` retourne `NULL` si fichier inexistant
  - `scanf()` retourne `EOF` s'il y a une erreur de lecture
  - après une erreur, `errno` contient un code entier qui précise la nature de l'erreur (`EPERM`, `ENOENT`, `ENOSPC`, etc)

Exceptions (4)

- Avec les codes d'erreurs il faut **entremêler** le traitement des exceptions avec le code exécuté normalement, ce qui rend le code lourd, difficile à comprendre et maintenir:

```
int traiter_fichier ()
{ char *t = malloc (1000);
  if (t != NULL)
    {
      FILE *f = fopen ("fich", "r");
      if (f != NULL)
        {
          ...lire f et utiliser t...
          close (f);
        }
      else
        {
          free (t);
          return 0;
        }
      free (t);
    }
  else
    return 0;
  return 1;
}
```

Exceptions en Java

© 2011 Marc Feeley

- Exemple sans traitement d'exception: **renverser le contenu d'un fichier**

```
import java.io.*;

public class exceptions          // note: ce programme ne compile pas
  {
    static int lire_fichier (String nom, char[] t)
      {
        FileReader f = new FileReader (nom);  // fichier inexistant?
        int n = 0;
        int c;
        while ((c = f.read ()) != -1)         // erreur de lecture?
          t[n++] = (char)c;                   // index hors bornes?
        f.close ();
        return n;
      }

    static void renverser (String nom)
      {
        char[] t = new char[1000];            // mémoire épuisée?
        int n = lire_fichier (nom, t);
        while (n>0) System.out.print (t[--n]);// erreur d'écriture?
      }

    public static void main (String[] args)
      {
        renverser (args[0]);                  // index hors bornes?
      }
  }
```

try et catch de Java/C++

© 2011 Marc Feeley


```
public class exceptions   // avec try et catch
  {
    static int lire_fichier (String nom, char[] t)
        throws FileNotFoundException, IOException
      { FileReader f = new FileReader (nom); ... }

    static void renverser (String nom)
        throws FileNotFoundException, IOException
      { ... }

    public static void main (String[] args)
      { try
          {
            renverser (args[0]);
          }
        catch (FileNotFoundException e)
          {
            System.out.println ("fichier inexistant");
          }
        catch (IOException e)
          {
            System.out.println ("erreur d'E/S");
          }
        catch (IndexOutOfBoundsException e)
          {
            System.out.println ("accès hors bornes");
          }
        catch (OutOfMemoryError e)
          {
            System.out.println ("mémoire épuisée");
          }
      }
  }
```

`finally` de Java

© 2011 Marc Feeley


```
public class exceptions   // avec try et catch et finally
  {
    static int lire_fichier (String nom, char[] t)
        throws FileNotFoundException, IOException
      {
        FileReader f = new FileReader (nom);
        int n = 0;
        try
          {
            int c;
            while ((c = f.read ()) != -1)
              t[n++] = (char)c;
          }
        finally
          {
            f.close ();
          }
        return n;
      }

    static void renverser (String nom)
        throws FileNotFoundException, IOException
      { ... }

    public static void main (String[] args)
      { ... }
  }
```

Types de données et leur représentation

© 2011 Marc Feeley

- Il faut distinguer **objet** et **représentation**
  - **Objet** = donnée du point de vue de l'application
  - **Représentation** = organisation des valeurs composant l'objet (i.e. encodage de l'objet)

- Exemple: objet = date du 3 février
  - Représentation 1: entier 34
  - Représentation 2: entiers 3 et 2
  - Représentation 3: chaîne "03/02"

Types de données (1)

© 2011 Marc Feeley

- Déf: **Type** = ensemble d'objets (possiblement infini)

- Exemples:

1. Type Booléen = { VRAI, FAUX }
2. Type entier = { 0, 1, -1, 2, -2, ... }

Types de données (2)

© 2011 Marc Feeley

- Classifiés en 2 groupes
- **Types simples** (normalement prédéfinis), p.e. entier, caractère, Booléen, point flottant
- **Types composés** (normalement définis explicitement), formés à l'aide d'autres types (éléments de même type=**homogène**, sinon **hétérogène**)

Types de données (3)

© 2011 Marc Feeley

- **Tableau** = groupe ordonné d'objets accessibles par leur position
  - Index connu à l'exécution
  - En général homogène, mais hétérogène en Lisp et les langages OO
  - 2 bornes (Pascal, Ada) / longueur + borne inférieure fixée à 0 (C/Java, Lisp) ou 1 (FORTRAN)
  - **Bornes fixées?** à la compilation (**tableau statique**: C, FORTRAN) ou à l'exécution (**tableau dynamique**: Java, Pascal, Lisp)

Types de données (4)

© 2011 Marc Feeley

- **Enregistrement** = groupe ordonné d’objets accessibles par leur nom
  - Nom connu à la compilation
  - En général hétérogène

Types de données (5)

© 2011 Marc Feeley

- **Ensemble** = groupe non-ordonné d'objets sans duplication, accessibles par élément
  - En général homogène avec un type discret de petite taille, mais hétérogène en Lisp
  - Exemple: ensemble de Booléens = { {}, {VRAI}, {FAUX}, {VRAI,FAUX} }
  - opérations: union, intersection, différence, "élément de", inclus, égal

Représentation des entiers (1)

© 2011 Marc Feeley

- Unités de mesure = bit, octet, mot (taille naturelle pour l'UCT, 32/64 bits = 4/8 octets)
- Entier = sous-ensemble de **Z** (sauf Lisp où: entier = **Z** = { 0, 1, -1, 2, -2, ... })
- Entier = entiers signés représentables par un mot: complément à 2, i.e. $-2^{31}..2^{31}-1$, ou complément à 1, i.e. $-2^{31}+1..-0,0..2^{31}-1$ (Cray et CDC)

Représentation des entiers (2)

© 2011 Marc Feeley

- C permet de choisir la taille et signé ou pas:
  `[signed|unsigned]( int|short|long|char )`
- Note 1: `char` occupe une “unité” d’espace **souvent 8 bits**
  Exemple:
  `unsigned char x;  0<=x<=255`
- Note 2: `char` ⊆ `short` ⊆ `int` ⊆ `long`
- Note 3: `sizeof(char)=1`, et souvent `sizeof(short)=2`, `sizeof(int)=4`, ...
- Note 4: `#include <limits.h>` pour avoir les limites de chaque type: `INT_MIN`/`INT_MAX`

Représentation des entiers (3)

© 2011 Marc Feeley

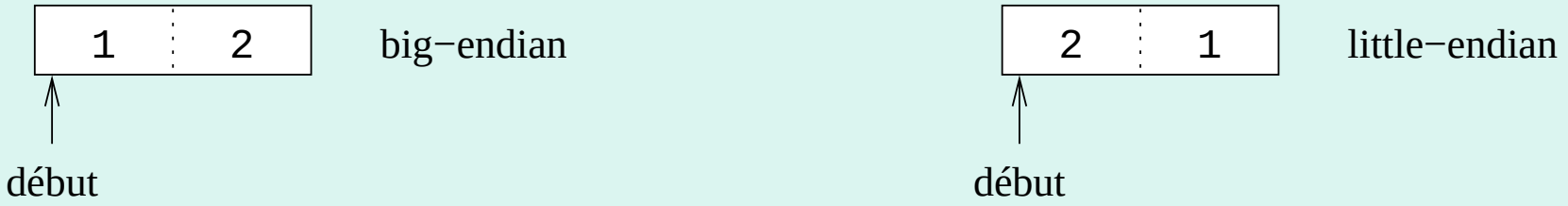

```
/* Fichier: "test.c" */

#include <stdio.h>
#include <limits.h>

int main(int argc, char *argv[])
{
  printf("c=%lu s=%lu i=%lu l=%lu ll=%lu p=%lu\n",
         sizeof(char), sizeof(short), sizeof(int),
         sizeof(long), sizeof(long long), sizeof(void *));

  printf("CHAR_MIN=%d CHAR_MAX=%d\n", CHAR_MIN, CHAR_MAX);

  printf("INT_MIN=%d INT_MAX=%d\n", INT_MIN, INT_MAX);

  return 0;
}
```

```
% gcc test.c -o test.exe
% ./test.exe
c=1 s=2 i=4 l=8 ll=8 p=8
CHAR_MIN=-128 CHAR_MAX=127
INT_MIN=-2147483648 INT_MAX=2147483647
```

Représentation des entiers (4)

© 2011 Marc Feeley

- Une autre variation dans la représentation des entiers est l'ordre des octets en mémoire
- Les deux ordres principaux sont "**big-endian**" (MIPS, SPARC, M68000) et "**little-endian**" (Intel, DEC Alpha)
- Exemple, l'entier 16 bits 258 = 1*256 + 2



- Cela cause des problèmes lors du transfert de données binaires entre ordinateurs de type différent (par exemple sur l'internet)

Représentation des entiers (5)

© 2011 Marc Feeley

- Intervalle du type entier
- Pascal: <const1>..<const2>
  = { <const1>,<const1>+1,...,<const2> }
- Nombre bits = $\lceil log_2(C2 - C1 + 1) \rceil$
- Exemple: `1..2` $->$ 1 bit, `5..7` $->$ 2 bits
- Souvent les compilateurs arrondissent vers l'octet le plus proche pour un accès rapide
- Utilisé en Pascal pour définir les bornes des tableaux:

```
var T1: array [10..20] of integer;
    T2: array [char] of integer;
```

Représentation des types énumérés (1)

© 2011 Marc Feeley

- Chaque objet élément du type a un nom symbolique
- Exemple en Pascal:

```
var j: (lundi,mardi,mercredi,jeudi,
        vendredi,samedi,dimanche);

begin
  j := mercredi;
  if j = lundi then ...;
  j := succ(j); (* j=jeudi *)
  j := pred(j)  (* j=mercredi *)
end
```

- Représenté comme type sous-intervalle

Représentation des types énumérés (2)

© 2011 Marc Feeley

- Exemple en C:

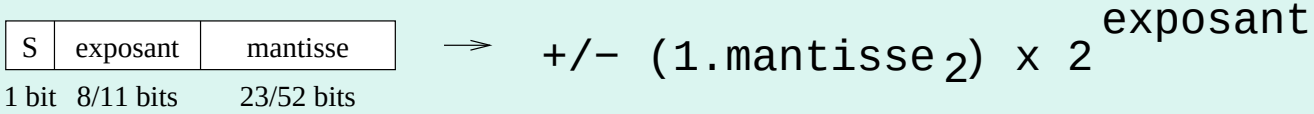
```
enum {lundi,mardi,mercredi,jeudi,
      vendredi,samedi,dimanche} j;

j = lundi;
j = j+1;
```

- Le type Booléen de Pascal est un type énuméré:
  `boolean = (false,true)`

Représentation des types réels

© 2011 Marc Feeley

- Réel = sous-ensemble de **R** (en fait sous-ensemble de **Q**, les rationnels)
- Réel = nombres rationnels représentables avec un encodage point-flottant (le plus répandu = IEEE-754)



- Précision “simple” (32 bits) $<= 3.4 \times 10^{38}$ (6 chiffres significatifs)
- Précision “double” (64 bits) $<= 1.8 \times 10^{308}$ (15 chiffres significatifs)
- En C: `float` $\subseteq$ `double`

Représentation des tableaux (1)

© 2011 Marc Feeley

- Pascal:

  `var T: array [`<*type_index*>`{ , `<*type_index*>`}] of `<*type_élem*>`;`

  => <*type_index*> doit être sous-intervalle ou énuméré

- C: <*type_élem*> `T[`<*constante*>`]{ [`<*constante*>`] } ;`

- Java: <*type_élem*> `[] { [] } T ;`

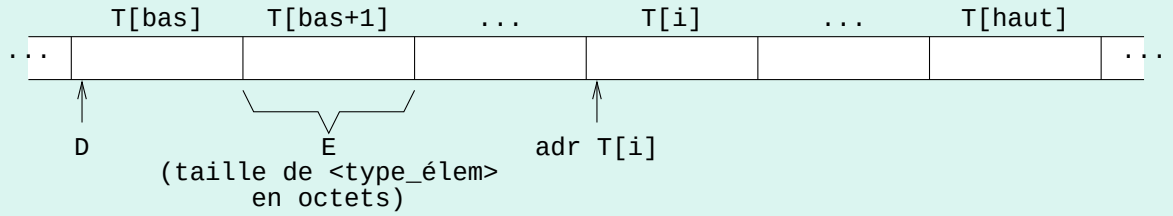
- Exemples Pascal, C, Java:

```
var T: array [boolean] of real;                 T[false] := 1.5;
var T: array [(rouge,vert,bleu)] of boolean;    T[vert] := true;
var T: array [10..12] of integer;               T[11] := 123;
var T: array [0..4,0..9] of real;               T[2,3] / T[2][3]
var T: array [0..4] of array [0..9] of real;    T[2,3] / T[2][3]

int T[2];           T[0] = 123;
float T[5][10];     T[2][3] = 1.5;

int[] T = new int[2];
float[][] T = new float[5][10];
```

Représentation des tableaux (2)

© 2011 Marc Feeley

- Représentation de
  `var T: array [bas..haut] of integer;`



- Taille de `T` = (`haut - bas + 1`)*E
- Adr de `T[i]` = D + (`i-bas`)*E = `i`*E + K
  où K = D − `bas`*E
- Quand indexation rapide?
  1. Utilisation d'un tableau statique, car K est calculable à la compilation
  2. Utilisation d'une taille d'élément E = 2^n

Représentation des tableaux (3)

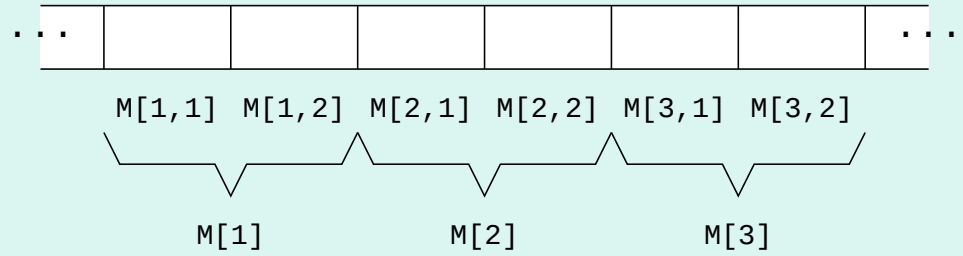
© 2011 Marc Feeley

- Tableau multidimentionnel:

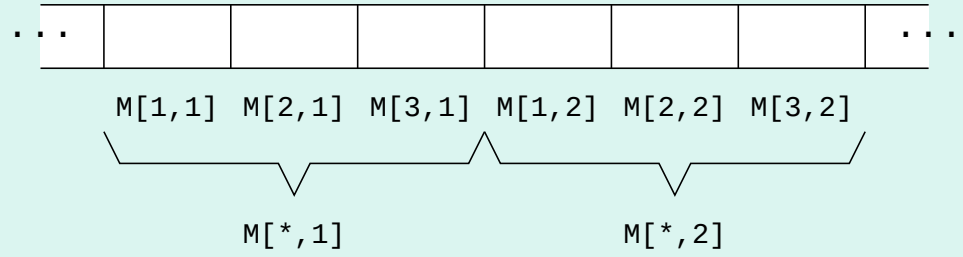
```
var M : array [1..3] of array [1..2] of integer;
```

définition d'une rangée de la matrice

$$\begin{bmatrix} M[1,1] & M[1,2] \\ M[2,1] & M[2,2] \\ M[3,1] & M[3,2] \end{bmatrix}$$



''row-major''



''column-major''
(FORTRAN)

Représentation des tableaux (4)

© 2011 Marc Feeley

- Taille de `M` = `(h1-b1+1)*(h2-b2+1)*E`

- Adr de `M[i,j]` (en représentation “row-major”)

  = `D + (i-b1)*(h2-b2+1)*E + (j-b2)*E`

  = `i*(h2-b2+1)*E + j*E + (D-b1*(h2-b2+1)*E-b2*E)`
  = `i*K1 + j*E + K2`

- Donc `array[1..3,1..2]` est plus rapide à indexer que `array[1..2,1..3]`

Représentation des tableaux (5)

© 2011 Marc Feeley

- Java et SIMULA offrent les tableaux dynamiques

```
void p (int n)    // Java
{
  int[] t = new int[n];
  ...
}

procedure p(n); integer n;  -- SIMULA
  begin
    integer array t(1:n);
    ...
  end;

p(10);  p(20);
```

- Le calcul d'adresse doit être **dynamique** (donc il faut stocker les bornes en plus des éléments)

Représentation des tableaux (6)

© 2011 Marc Feeley

- Représentation 1: permet de vérifier les indices à l'exécution

bas1	haut1	bas2	haut2	...	elem1	elem2	...

- Représentation 2: plus rapide mais pas de vérification et suppose borne du bas = 0 (Di = `haut_i`+1)

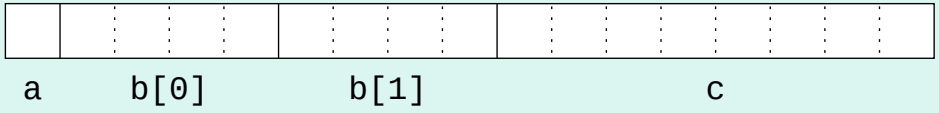
D1*E	D1*D2*E	...	elem1	elem2	...

Représentation des enregistrements (1)

© 2011 Marc Feeley

- Disposition contiguë des champs en mémoire

```
struct
  {
    char a;     /* adresse relative = 0 */
    int b[2];   /* adresse relative = 1 */
    double c;   /* adresse relative = 9 */
  }
```



- Les contraintes de la machine peuvent affecter la position (p.e. `int` (`double`) à des adresses qui sont des multiples de 4 (8))

Représentation des enregistrements (2)

© 2011 Marc Feeley


```
struct s1 { char a; } t1[100];
struct s2 { int a; } t2[100];
struct s3 { int a; char b; } t3[100];

struct s4 { char a; char b; char c; };
struct s4 t4[100];

typedef struct { char a; int b[2]; double c; } s5;
s5 t5[100];

int main(int argc, char *argv[])
{ printf("s1=%-4lu s2=%-4lu s3=%-4lu s4=%-4lu s5=%-4lu\n",
         sizeof(struct s1), sizeof(struct s2),
         sizeof(struct s3), sizeof(struct s4), sizeof(s5));

  printf("t1=%-4lu t2=%-4lu t3=%-4lu t4=%-4lu t5=%-4lu\n",
         sizeof(t1), sizeof(t2), sizeof(t3), sizeof(t4), si

  return 0;
}
```

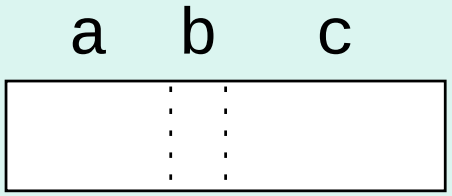
```
% gcc test.c -o test.exe ; ./test.exe
s1=1    s2=4    s3=8    s4=3    s5=24
t1=100  t2=400  t3=800  t4=300  t5=2400
```

Représentation des enregistrements (3)

© 2011 Marc Feeley

- C permet un contrôle fin (au niveau des bits) sur la disposition des champs

```
struct
  { unsigned int a : 3; /* 0..7 */
    unsigned int b : 1; /* 0..1 */
    unsigned int c : 4; /* 0..15 */
  }
```



3 bits 1 bit 4 bits

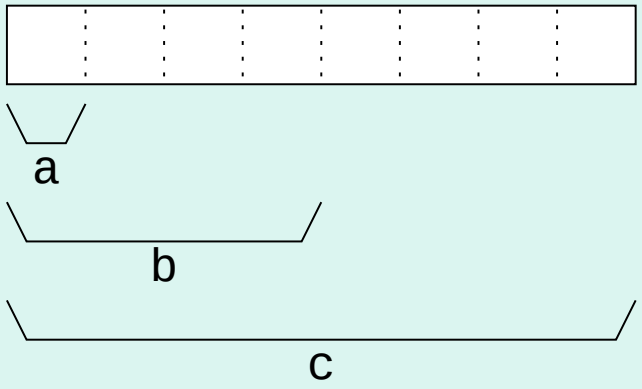
1 octet

Représentation des enregistrements (4)

© 2011 Marc Feeley

- Déf: **type union** = union de plusieurs types
- En C la syntaxe est similaire aux structures

```
union
  { char a;    /* adresse relative = 0 */
    int b;     /* adresse relative = 0 */
    double c;  /* adresse relative = 0 */
  }
```



- Taille = taille du plus grand champs

Représentation des enregistrements (5)

© 2011 Marc Feeley

- Seulement un des champs est actif à un instant donné et c'est impossible de savoir lequel uniquement en examinant l'objet

```
struct
  {
    int sorte; /*0..2*/
    union
     {
       char a;
       int b;
       double c;
     } val;
  } T[10];

for (i=0; i<10; i++)
  switch (T[i].sorte)
    {
      case 0: printf("%c",T[i].val.a); break;
      case 1: printf("%d",T[i].val.b); break;
      case 2: printf("%f",T[i].val.c); break;
    }
```

Représentation des ensembles

© 2011 Marc Feeley

- Pascal: `set of` <*type*>
  => <*type*> doit être sous-intervalle ou énuméré
- Représenté par un mot machine avec 1 bit par élément possible (1=élement de l'ensemble, 0=pas élement de l'ensemble)
- Exemple

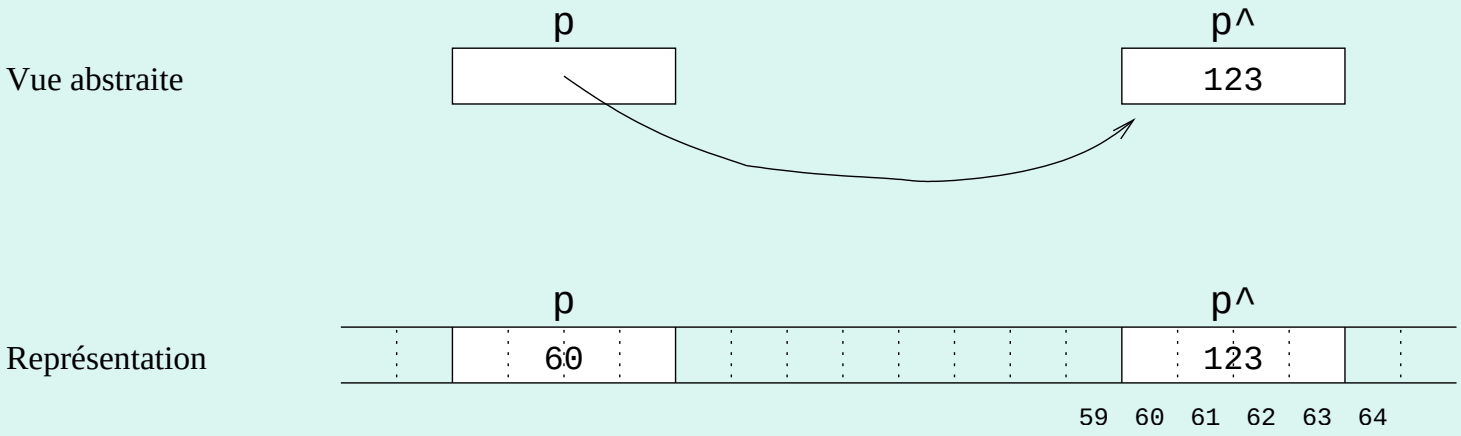
```
var s1, s2, s3 : set of 1..5;

s1 := [1,4,5];  (* 0000000000011001 = 25 *)
s2 := [2,4];    (* 0000000000001010 = 10 *)
s3 := s1 + s2;  (* 0000000000011011 = 27 *)
s3 := s1 * s2;  (* 0000000000001000 = 8  *)
s3 := s1 - s2;  (* 0000000000010001 = 17 *)
if 2 in s1 then ...;
```

Représentation des pointeurs (1)

© 2011 Marc Feeley

- Déf: **pointeur** = objet qui permet un accès **indirect** à un autre objet
- Pascal: ^ *<type>*

```
var p : ^integer;
```



- C: *<type>* *

Représentation des pointeurs (2)

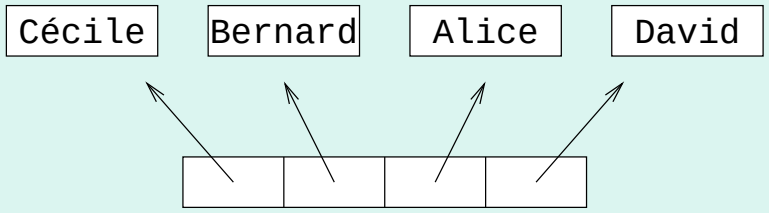
© 2011 Marc Feeley

- La taille du pointeur est indépendante de la taille de l'objet pointé (typiquement 1 mot machine)
- En général, un pointeur est représenté par l'**adresse en mémoire** de l'objet pointé
- Cas spécial: pointeur nul (`nil` en Pascal, `NULL` en C/C++) $=>$ 0 ou autre adresse invalide
- Un pointeur nul indique qu'il n'y a **pas d'objet pointé**

Représentation des pointeurs (3)

© 2011 Marc Feeley

- Applications
  - Structures de données dont la taille varie à l'exécution (chaînes, listes, arbres, graphes)
  - Manipulation rapide d'objets (la copie d'un pointeur est plus rapide que la copie de l'objet pointé)
    Exemple: trier des enregistrements



Représentation des pointeurs (4)

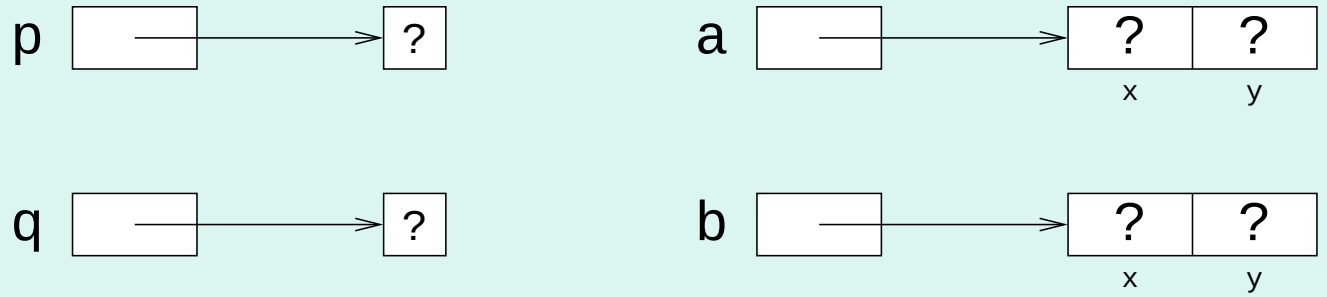
© 2011 Marc Feeley

- Opérations principales sur les pointeurs
  - **Allocation** d'un objet dans le tas (trouve espace inutilisé + marque comme utilisé)
  - **Récupération** de l'espace alloué à un objet (marque inutilisé)
  - **Indirection** (accès à l'objet pointé)

Représentation des pointeurs (5)

© 2011 Marc Feeley

Pascal	C	SIMULA
DÉCLARATION DE TYPE ET VARIABLE		
`type R = record x, y : integer; end;`	`struct R { int x, y; };`	`class R; begin integer x, y; end;`
`var p, q : ^char; a, b : ^R;`	`char *p, *q; struct R *a, *b;`	`ref(R) a, b;`
ALLOCATION		
`new(p);`	`p = malloc(1);`	
`new(q);`	`q = malloc(1);`	
`new(a);`	`a = malloc(sizeof(struct R));`	`a :- new R;`
`new(b);`	`b = malloc(sizeof(struct R));`	`b :- new R;`

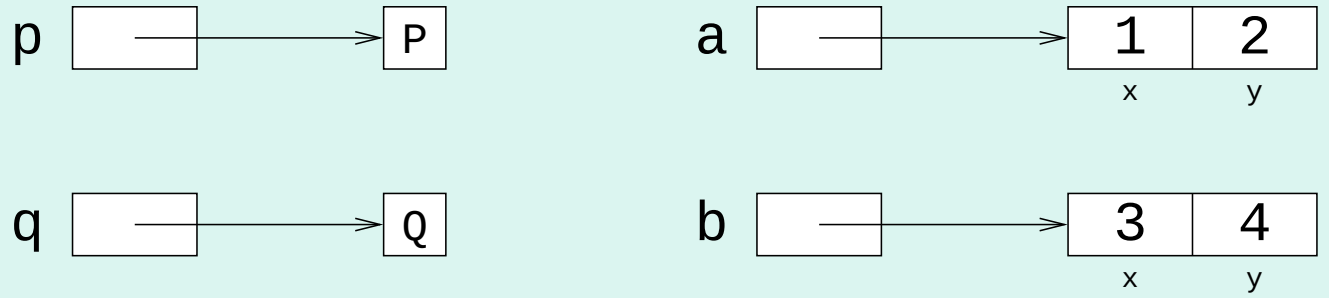


Représentation des pointeurs (6)

© 2011 Marc Feeley

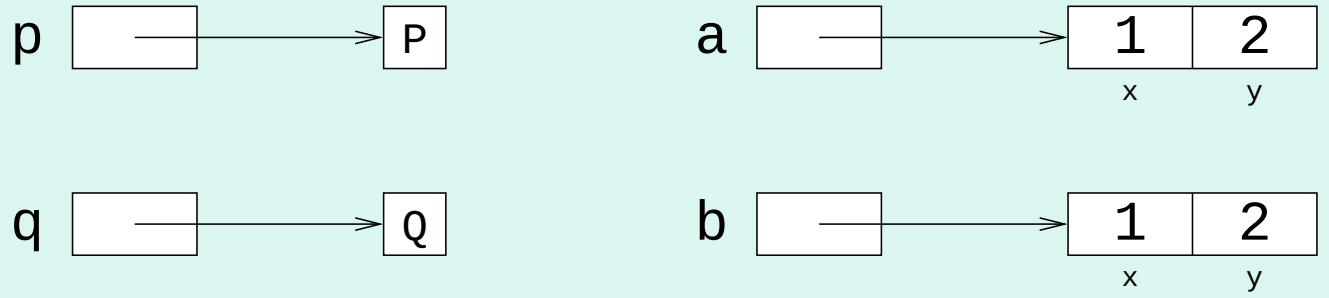
INDIRECTION

```
pˆ := ’P’;          *p = ’P’;
qˆ := ’Q’;          *q = ’Q’;
aˆ.x := 1;          (*a).x = 1; a->x = 1;      a.x := 1;
...
```



AFFECTATION DE L’OBJET POINTÉ

```
bˆ := aˆ;            *b = *a;                  b := a;
```



Représentation des pointeurs (7)

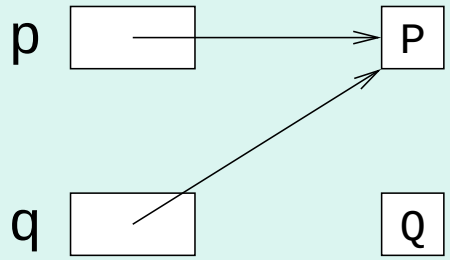
© 2011 Marc Feeley

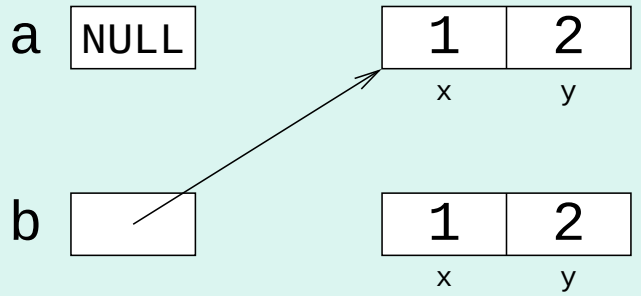
AFFECTATION DE POINTEUR

```
q := p;
b := a;
a := nil;
```

```
q = p;
b = a;
a = NULL;
```

```
b :- a;
a :- none;
```





ÉGALITÉ

```
if a=b then ...
```

```
if (a==b) then ...
```

```
if a==b then ...
```

RÉCUPERATION

```
dispose(b);
```

```
free(b);
```

```
**AUTOMATIQUE**
```

```
==> ne modifie pas le pointeur b
```

Représentation des pointeurs (8)

© 2011 Marc Feeley

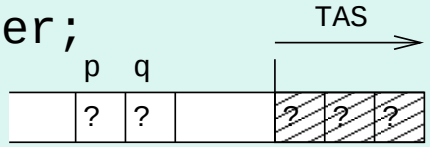
- Il y a des dangers importants lorsque la récupération de la mémoire est sous la responsabilité du programmeur (**récupération manuelle**)

- Le programmeur risque de récupérer l'espace **trop tôt** ce qui engendre des **pointeurs fous** (pointeur vers un espace réservé pour un autre usage)

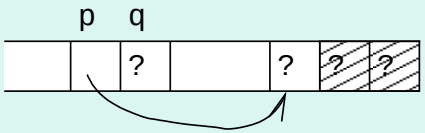
Représentation des pointeurs (9)

© 2011 Marc Feeley

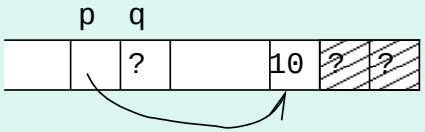

```
var p,q:^integer;
```



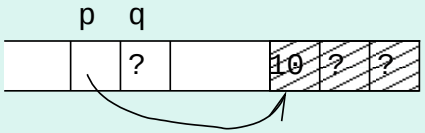

```
new(p);
```



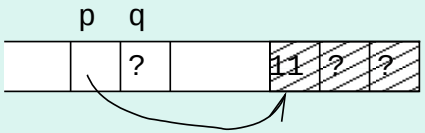

```
p^ := 10;
```



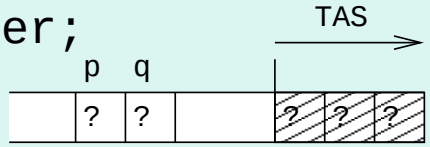

```
dispose(p);
```



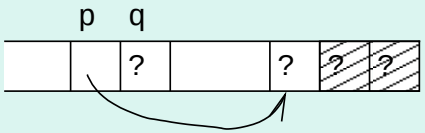

```
p^ := p^+1;
```



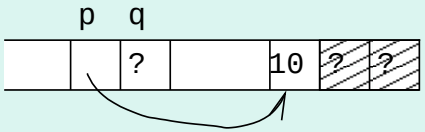

```
var p,q:^integer;
```



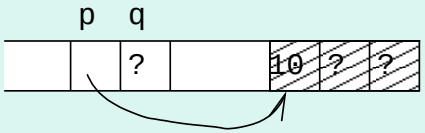

```
new(p);
```



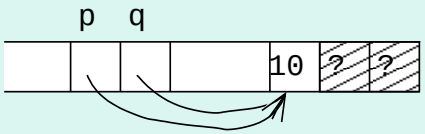

```
p^ := 10;
```



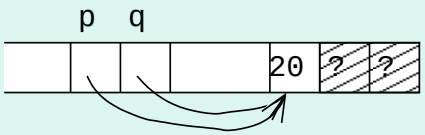

```
dispose(p);
```



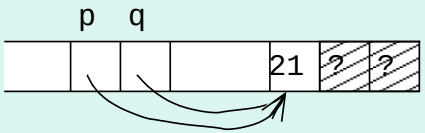

```
new(q);
```




```
q^ := 20;
```




```
p^ := p^+1;
```



Représentation des pointeurs (10)

© 2011 Marc Feeley

- Le programmeur risque de récupérer l'espace **trop tard** (ou même jamais) ce qui engendre des **fuites de mémoire**

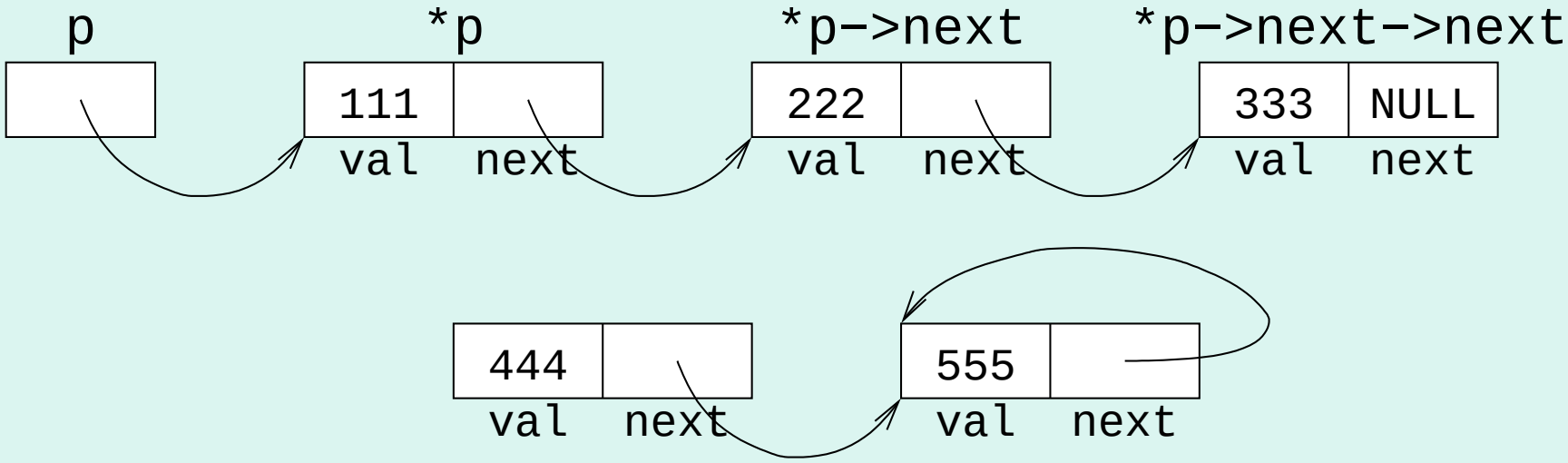
```
char *p = malloc(1);  /* alloc. objet 1 */
char *q = malloc(1);  /* alloc. objet 2 */

*p = 'P';

p = q;  /* fuite de mémoire: objet 1 n'est
           plus accessible et ne peut plus
           être récupéré */
```

Représentation des pointeurs (11)

© 2011 Marc Feeley

- Déf: **objet mort** = objet qui ne sera jamais plus utilisé par le programme
- Déf: **objet atteignable** = objet qui est pointé par une variable du programme ou par un des éléments/champs d'un objet atteignable

```
struct cell { int val; struct cell *next; };
struct cell *p;
```



Représentation des pointeurs (12)

© 2011 Marc Feeley

- Un objet qui n'est plus atteignable est mort (mais l'inverse n'est pas vrai)
- Une fois qu'un objet n'est plus atteignable il est mort pour toujours (il est impossible d'obtenir un pointeur vers cet objet)
- Déf: **fuite de mémoire** = création d'objets morts
- **Bug insidieu** car le programme peut marcher longtemps sans problèmes mais tranquillement la performance se dégrade et cause éventuellement un arrêt total (p.e. Microsoft Word)

Représentation des pointeurs (13)

© 2011 Marc Feeley

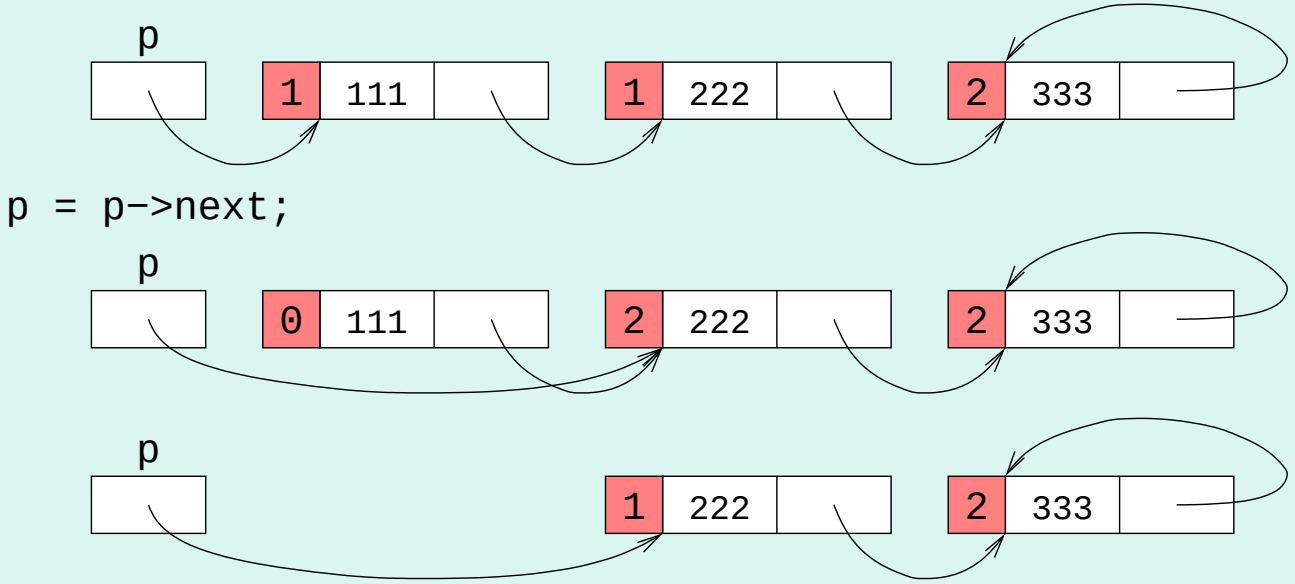
- La récupération manuelle est une **source de bugs majeurs** dans les programmes complèxes car il est difficile de savoir quel module est responsable de récupérer l'espace (de 30%-40% du temps de débogage)

- Lisp, SIMULA et Java possèdent des "**ramasses-mièttes**" ou "**glâneurs de cellules**" (**garbage collectors** ou tout simplement **GC**) qui détectent la création d'objets morts et récupère automatiquement l'espace associé

Représentation des pointeurs (14)

© 2011 Marc Feeley

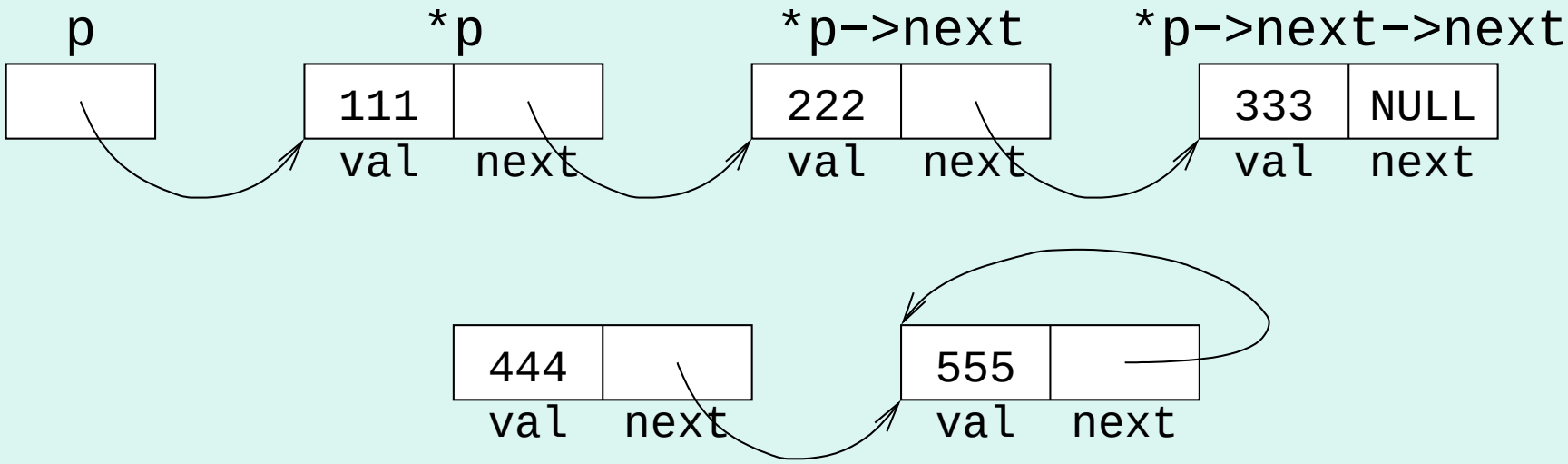
- Approche 1: Chaque objet a un **compteur de référence** = nombre de pointeurs vers l'objet (initialisé à 1, incrémenté à chaque copie de pointeur, et décrémenté à chaque fois qu'un pointeur disparait, p.e. écrasé par un autre pointeur) et récupérer l'espace lorsque compteur = 0



Représentation des pointeurs (15)

© 2011 Marc Feeley

- Approche 2: à partir des variables du programme **déterminer tous les objets atteignables** en passant par des pointeurs et marquer ces objets. À la fin, les objets non-marqués sont récupérés.



Représentation des pointeurs (16)

© 2011 Marc Feeley

- C possède plusieurs routines de gestion mémoire (disponibles dans "`stdlib.h`")

```
void *malloc (int taille_bloc);
void *calloc (int nb_elem, int taille_elem);
void *realloc (void *ptr, int nouvelle_taille);
void free (void *ptr);
```

- `malloc` alloue un bloc
- `calloc` alloue un tableau
- `realloc` change la taille d'un objet alloué dans le tas (ce qui demande parfois d'allouer un nouveau bloc et de copier l'objet)
- `NULL` est retourné si la mémoire est épuisée

Représentation des pointeurs (17)

© 2011 Marc Feeley

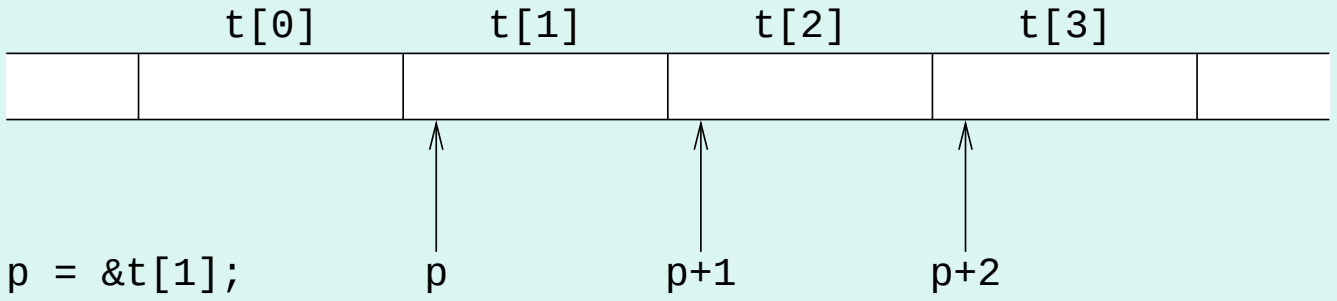
- Le pointeur retourné est de type `void*` (**pointeur universel**)
- En C++, il faut le convertir vers le bon type de pointeur avec un “cast”
- C++ possède 2 opérateurs de gestion mémoire de plus haut niveau que C:

```
int *p;
char *s;

p = new int;      // p = (int*)malloc(sizeof(int));
s = new char[5]; // s = (char*)calloc(5,sizeof(char));
delete p;         // free(p);
delete [] s;      // free(s);
```

Représentation des pointeurs (18)

© 2011 Marc Feeley

- Le langage C permet une forme d'**arithmétique sur les pointeurs** (opérateurs + et -)
- &<*var*> => pointeur vers <*var*>
- p+n (où n est de type entier et p est de type T*) = pointeur vers nième objet de type T après objet pointé par p



- p-n = pointeur vers nième objet avant *p

Représentation des pointeurs (19)

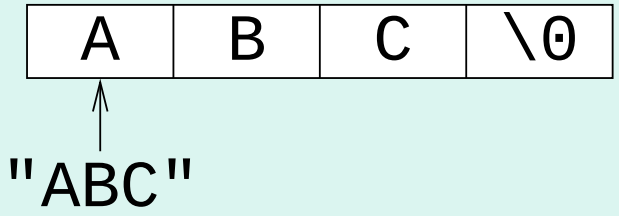
© 2011 Marc Feeley

- `p2-p1` (où `p1` et `p2` sont de type `T*`) = nombre d'objets de type T de `*p1` à `*p2`
- `t` (où `t` est un tableau) = pointeur vers premier élément du tableau
- C définit: `x[y]` = `*(x+y)` = `*(y+x)`
  Donc: `1[t]` = `t[1]` et `1[t][t]` = `t[t[1]]`
- Les opérateurs ++ et `--` sont utilisables sur les pointeurs pour avancer ou reculer d'un élément

Représentation des pointeurs (20)

© 2011 Marc Feeley

- Le traitement de chaînes de caractères en C est basé sur les pointeurs




```
char t[4];
char *p, *q;

p = "ABC";

putchar (p[2]);   putchar ("XYZ"[2]);

q = t;                    /* copier dans t */
while (*p != '\0')
  *q++ = *p++;
*q = '\0';

if (t == "ABC") ...; /* toujours faux! */

/* utiliser strcmp(t,"ABC") de <string.h> */
```

Typage et sûreté (1)

© 2011 Marc Feeley

- La notion de type est utile pour définir la notion d'**erreur**, par exemple:
  1. `int *p; ... abs(p) ...` est incorrect car (en C) l'argument de `abs` doit être entier
  2. `int t[10]; ... t[1.5] ...` est incorrect car l'index doit être entier
  3. `int t[10]; ... t[i] ...` est incorrect si `i` est négatif (ou plus grand que 9)
  4. `int n; ... 100/n ...` est incorrect si `n` est égal à `0`
- Déf: Une **erreur de type** survient lorsqu'une fonction (ou opérateur) est appelée avec un argument qui n'est pas du type attendu

Typage et sûreté (2)

© 2011 Marc Feeley

- Les **vérifications de type** permettent de garantir qu'aucune opération incorrecte ne sera exécutée
- Il est avantageux d'effectuer les vérifications de type **à la compilation** (**typage statique**) car cela évite de ralentir l'exécution du programme
- Cela rend le programme plus **sûr**, c'est-à-dire qu'il y a peu ou aucune possibilité d'erreur de type à l'exécution

Typage et sûreté (3)

© 2011 Marc Feeley

- Les vérifications de type peuvent également se faire à l'exécution (**typage dynamique**)
  - C'est requis dans certains cas (exemples 3 et 4) car ces propriétés sont **indécidables**
  - C'est dangereux si la partie du programme contenant l'erreur est exécutée rarement car il sera difficile d'éliminer l'erreur pendant la phase de développement (il est bon d'avoir des **tests de couverture** qui exécutent au moins une fois chaque partie du programme)

Typage et sûreté (4)

© 2011 Marc Feeley

- Le typage statique se base sur les **règles de typage** du langage
- Chaque opération produit un **résultat** de type R si on lui fournit des opérandes de type T_1, T_2, ...
- Déf: un opérateur est **surdéfini** (“overloaded”) si l’opération effectuée dépend du type des opérandes

Typage et sûreté (5)

© 2011 Marc Feeley

- Exemple d'opérateurs surdéfinis de C:

T_1 + T_2	int	long	double	void*	E*	struct S
int	int	long	double	*erreur*	E*	*erreur*
long	long	long	double	*erreur*	E*	*erreur*
double	double	double	double	*erreur*	*erreur*	*erreur*
void*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*
E*	E*	E*	*erreur*	*erreur*	*erreur*	*erreur*
struct S	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*

T_1 - T_2	int	long	double	void*	E*	struct S
int	int	long	double	*erreur*	*erreur*	*erreur*
long	long	long	double	*erreur*	*erreur*	*erreur*
double	double	double	double	*erreur*	*erreur*	*erreur*
void*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*
E*	E*	E*	*erreur*	*erreur*	ptrdiff_t	*erreur*
struct S	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*

Typage et sûreté (6)

© 2011 Marc Feeley

- Typiquement le langage demande de **spécifier explicitement** le type de chaque variable, paramètre et résultat de fonction (Pascal, C, Ada, Modula-3)
- Cela simplifie l'algorithme de vérification statique de type qui peut partir des feuilles d'une expression pour trouver le type global de l'expression
- Certains langages permettent **d'inférer automatiquement** le type des variables, paramètres et résultats de fonction (Haskell, ML), ce qui demande de résoudre un système de contraintes

Typage et sûreté (7)

© 2011 Marc Feeley

- Exemple en Standard ML:

```
- fun f (x : bool, y : int) : int =
    if x then y else 9;
val f = fn : bool * int -> int


- fun g (x, y) = if x then y else 9;
val g = fn : bool * int -> int


- fun h (x, y) = if x then y else x;
val h = fn : bool * bool -> bool


- fun i (x) = if x then x else 9;
stdIn:4.13-4.33 Error: case object and rules
                       don't agree [literal]
 rule domain: bool
 object: int
 in expression: (case x of true => x | false => 9)
```