L'importance des langages en informatique

Gérard Berry

Collège de France
Chaire Algorithmes, machines et langages
gerard.berry@college-de-france.fr

Cours 1, Inria Rennes, 04/11/2015

Suivi du séminaire de Thomas Jensen (Inria)

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Langage : système d'expression vocale ou écrite organisé sous forme de textes, dessins, formules, etc., permettant la formalisation des idées, leur préservation et leur communication

Les langages de programmation,
ils sont tous pareils,
il suffit d'en savoir un pour les savoir tous

Ceux qui disent ça comprennent-ils le domaine,
ou en montrent-ils au contraire leur ignorance ?

Au boulot pour la combler !

Les langages sont partout

Intention

Architecture

Microarchitecture

Spécification

Codage

Validation

Toutes ces activités utilisent leurs langages propres,
et doivent communiquer sans langage commun !

Quels langages pour quelles fonctions ?

- Discuter entre architectes (concepteurs)
  - échanger des idées et des pistes de réalisation
  - textes, formules, schémas, tableurs, prototypes, etc.
  - formalisation précise difficile et lente, donc rare, surtout dans les projets créatifs et innovants
- Discuter entre architectes et réalisateurs (programmeurs, designers de circuits)
  - spécifier : architectes → réalisateurs
    documents synthétiques plus ou moins précis
    listes de *requirements*
  - critiquer : réalisateurs → architectes
    retours d'expérience, détection d'incohérences, etc.
  - des relations difficiles, une grande source de bugs !

Quels langages pour quelles fonctions ?

- Affiner les spécifications
  - expliciter les choix de réalisation, supprimer les ambiguïtés dans les spécifications
  - langages ± formels (UML, B, Event-B, etc.)
  - un gros point faible du système → nid à bugs
- Programmer
  - rendre l'ensemble exécutable par une machine stupide
  - paradigmes et langages de programmation
  - outillage d'édition et de débogage
- Valider
  - tester et prouver
  - langages de définition et génération de tests
  - langages logiques (assertions, preuves, etc).

Les constituants d'un langage

syntaxe
textuelle ou graphique

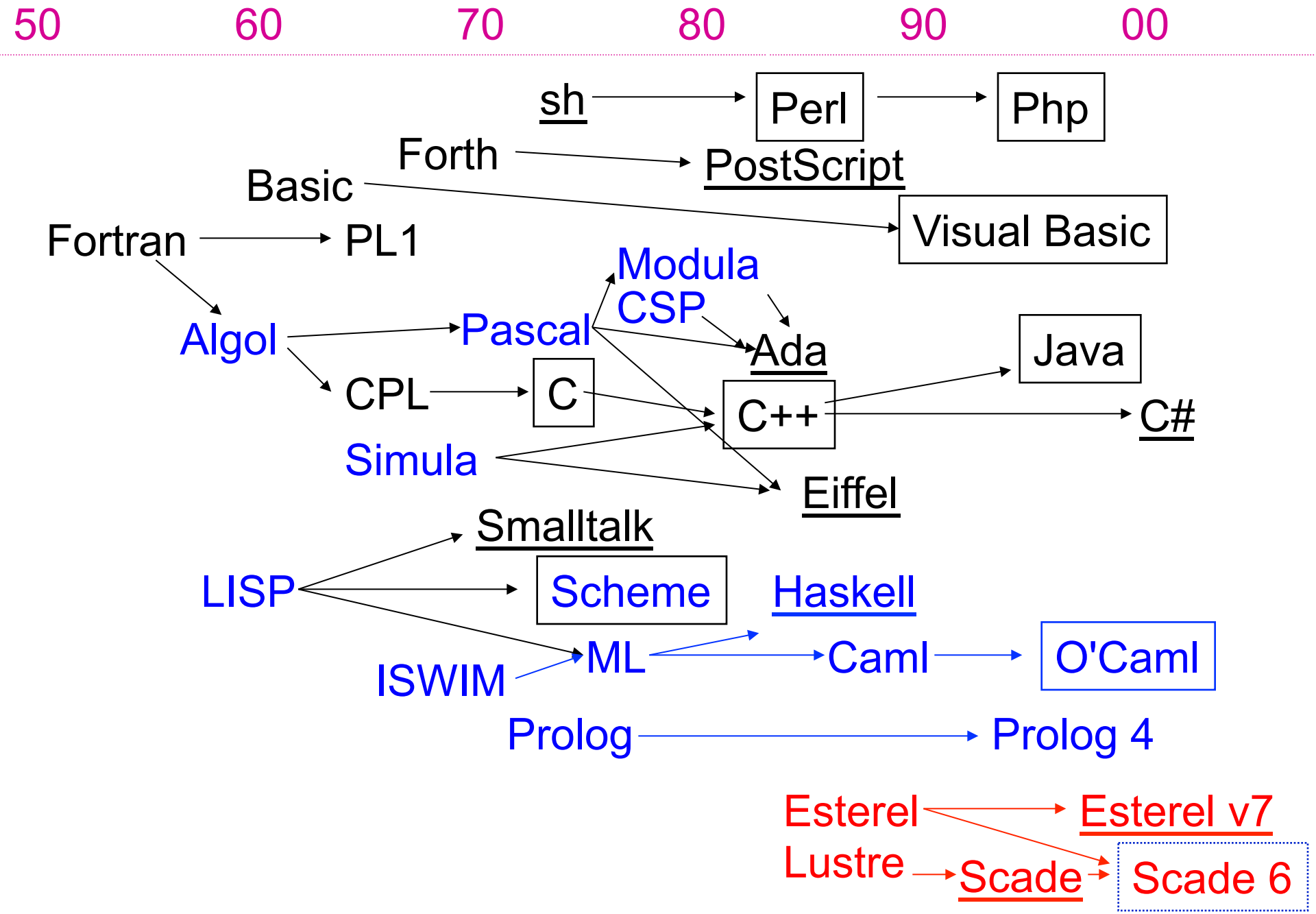
paradigme
style de programmation

sémantique
définit le sens des programmes

outillage : éditeurs, compilateurs, débogueurs

utilisateurs

Photo
Patick Sanfaçon
La Presse



leçon inaugurale n°1, 17/01/2008

D'où vient la pression?

efficacité

souplesse

sûreté

gestion

séquentiel?

parallèle?

science

Machines de
von Neumann

interprété?

compilé?

optimisation

typé?

logique,
λ-calcul (Church)

bases de
données

embarqué

Organigrammes

IHM

Automates

web

block-diagrams

domaines
spécifiques

OS

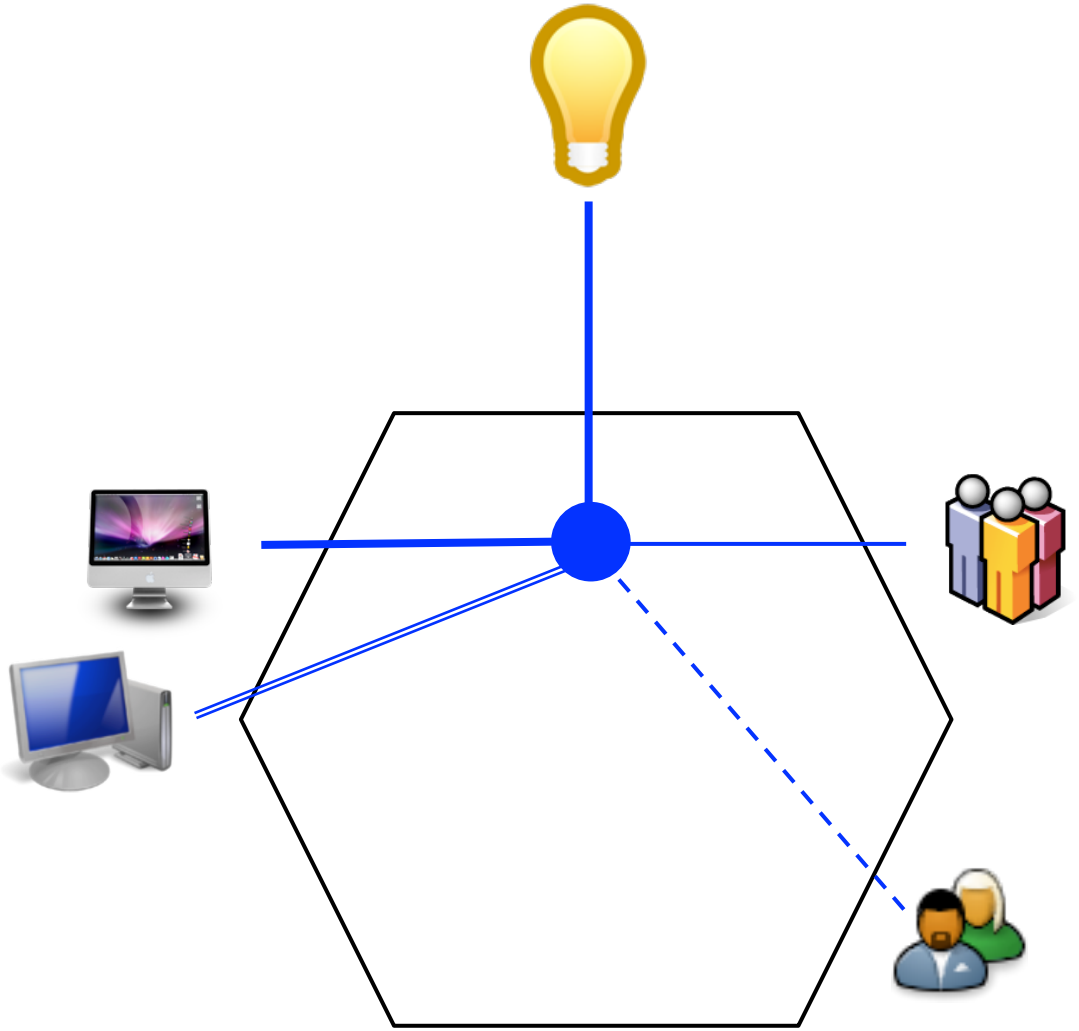
temps
réel

réseaux,
multiproc.

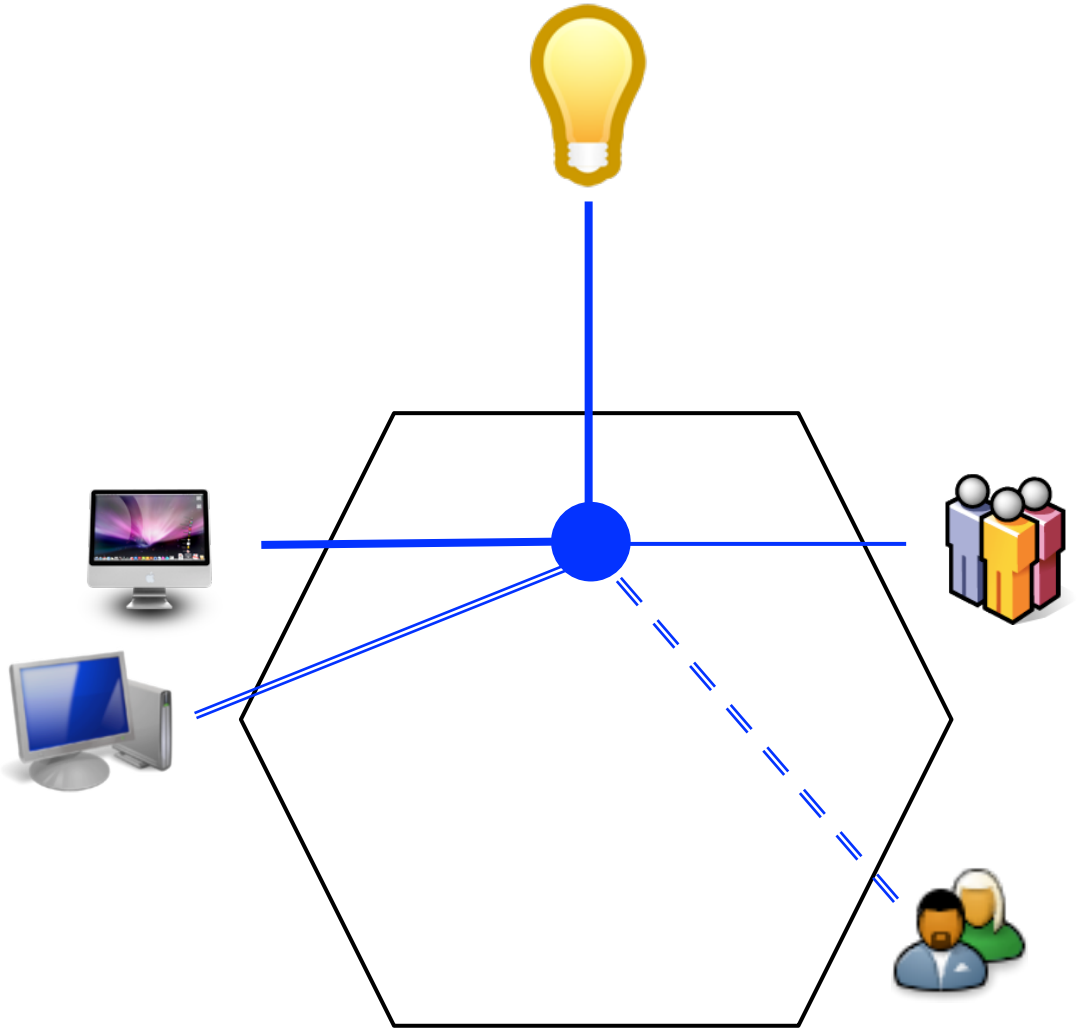
client-
serveur

téléchar-
gement

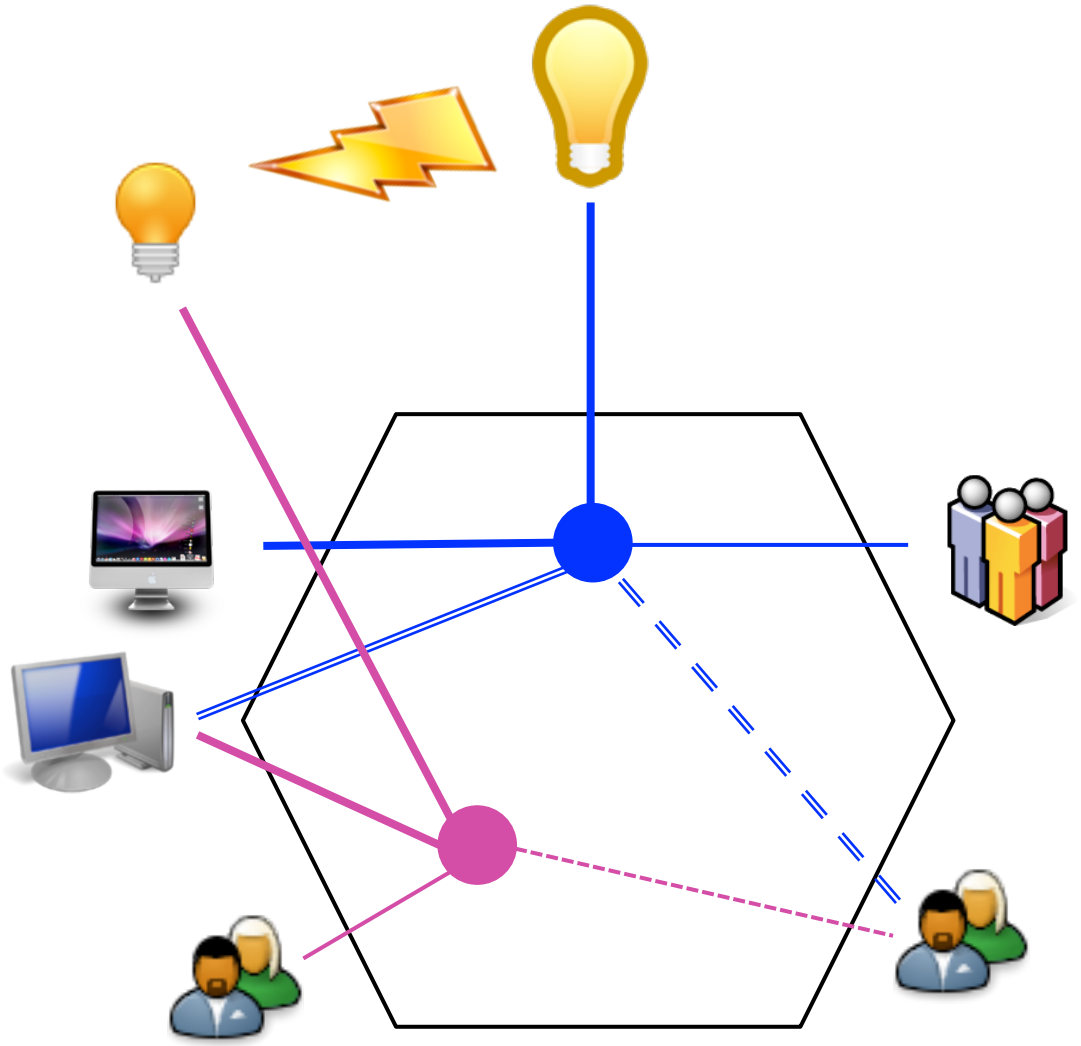
Pourquoi autant de langages ?



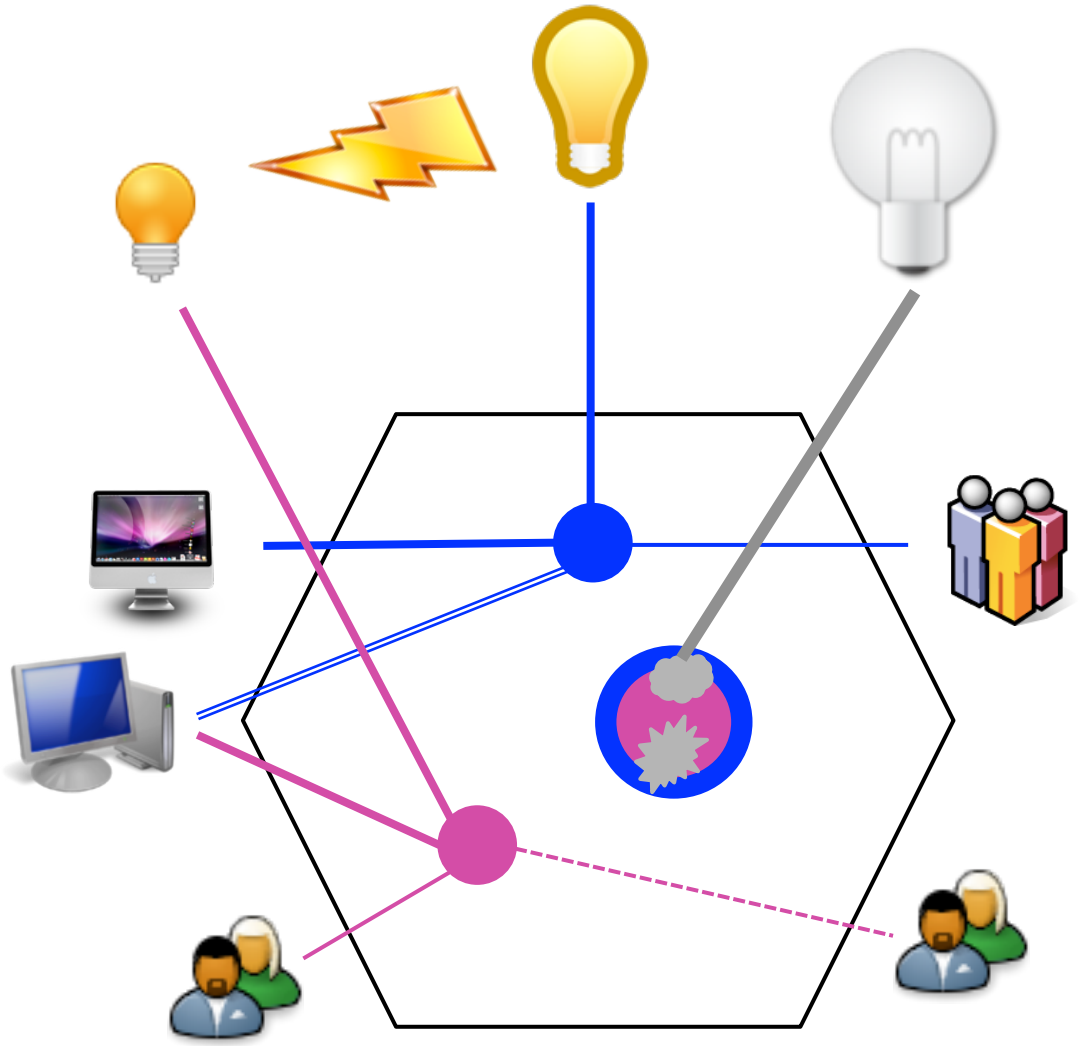
Pourquoi autant de langages ?



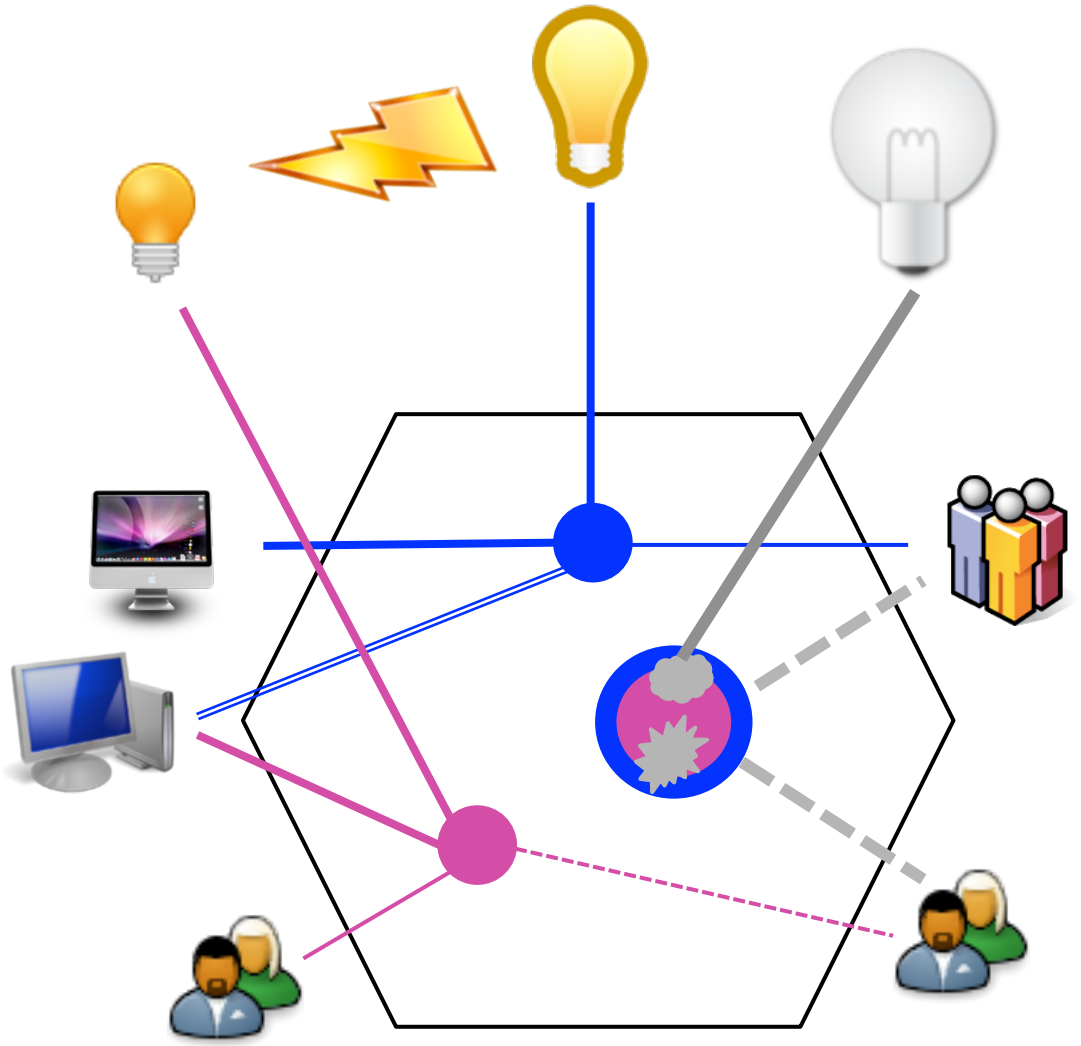
Et il y a plusieurs idées individuelles !



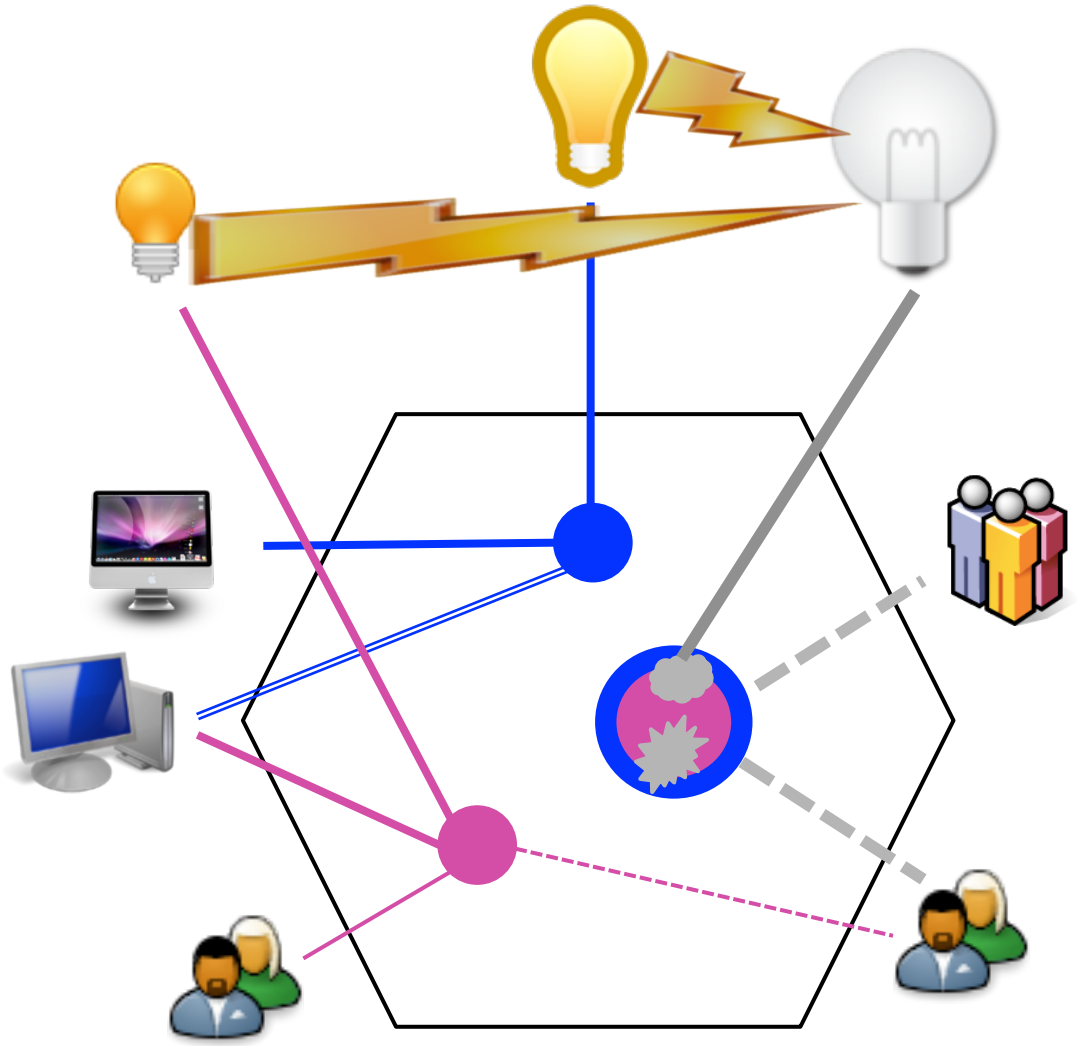
Des idées individuelles aux langages masala



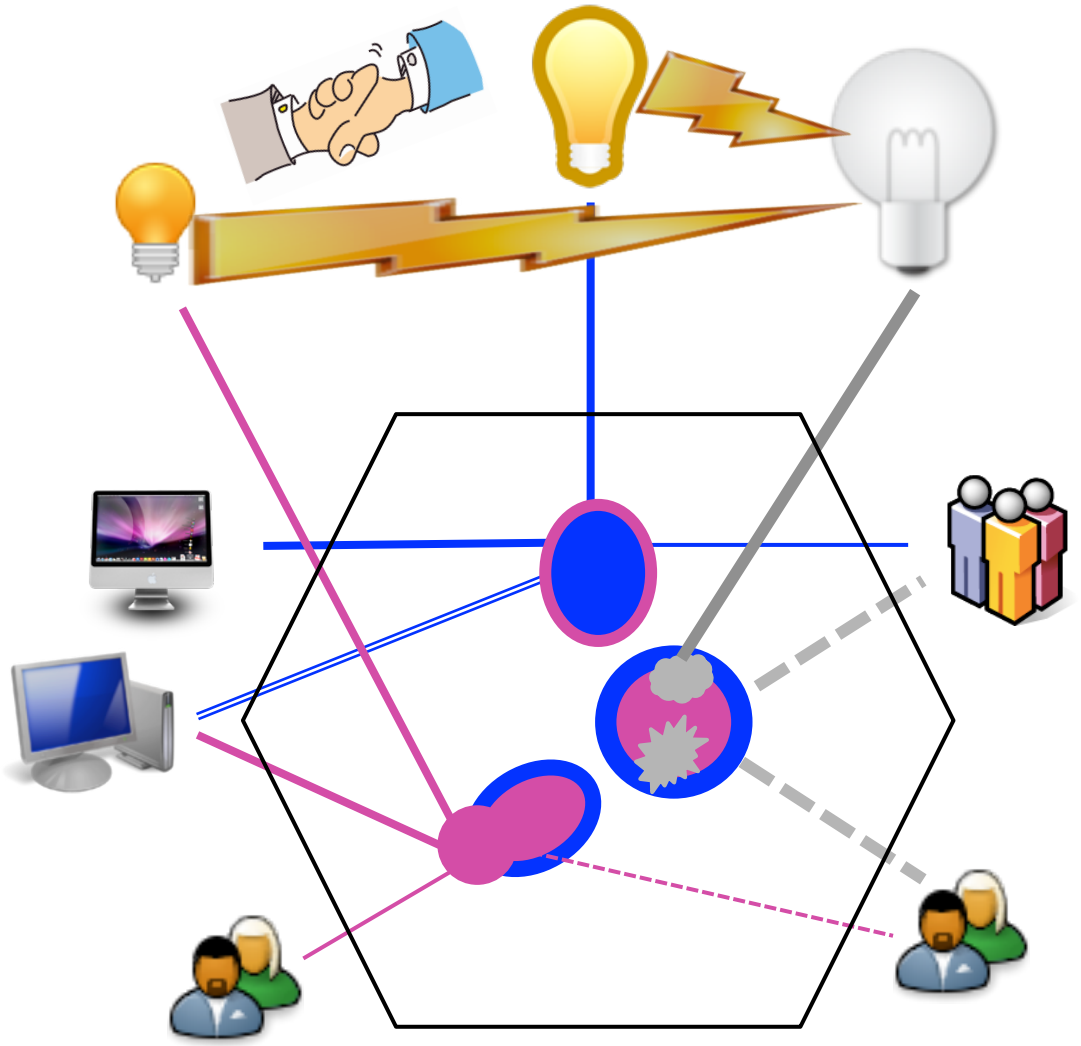
Des idées individuelles aux langages masala



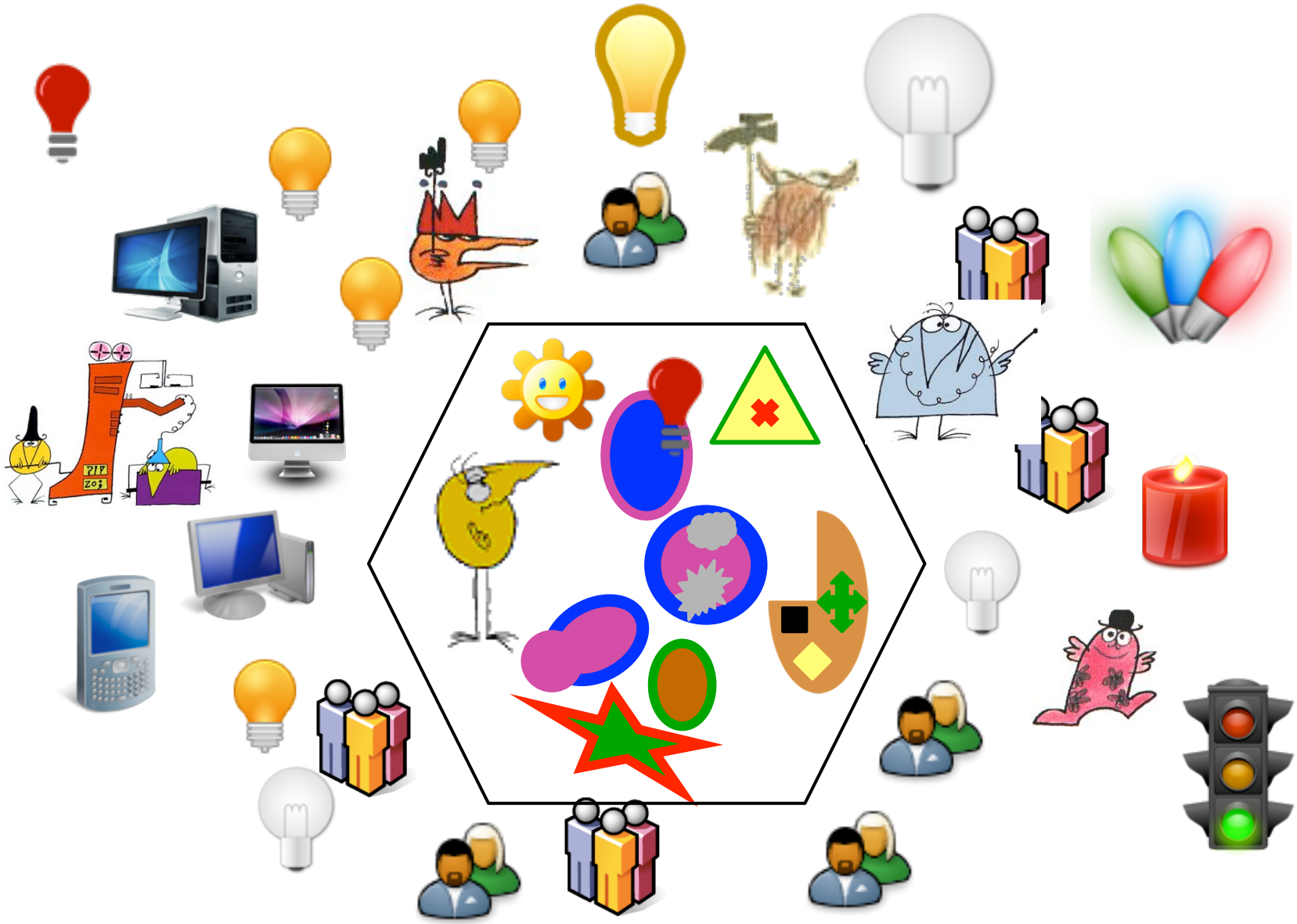
Des idées individuelles aux langages masala



Des idées individuelles aux langages masala



Et le paysage devient vite incompréhensible...



Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Les principes principaux

- Programmation impérative : rester près de la machine, en particulier pour l'efficacité, mais la rendre plus humaine.

  FORTRAN, COBOL, BASIC, PL/1, Pascal, C, ...

  → sémantique opérationnelle (changement d'état mémoire)
  sémantique dénotationnelle (VDM??)

- Programmation fonctionnelle: privilégier la justesse, en s'appuyant sur le socle mathématique solide des fonctions, de la logique et du lambda-calcul

  LISP, ALGOL, SCHEME, ML, Caml, Haskell, Coq, ...

  → sémantique dénotationnelle (fonctions, treillis, catégories)
  → sémantique opérationnelle (réécriture symbolique)

Impératif + fonctionnel : LISP (Mc Carthy, 1958)

```
(defun append (lst1 lst2) ; fonctionnel, récursif
    (cond
        ((null lst1) lst2
        (T (cons (car lst1) (append (cdr lst1) lst2)))))
=> append
```

```
(setq lst '(A B C)) ; affectation impérative
 => (A B C)
(append lst '(D E)) ; appel de fonction
 => (A B C D E)
```

Innovations majeures : interactivité, listes, calcul symbolique, récursivité systématique, gestion mémoire automatique, réflexivité (eval),...

liaison statique → SCHEME, typage → ML, Haskell, ...

Langages fonctionnels : ML, HASKEL

```
let rec append l l' =
  match l with
    [ ] −> l'
  | a :: l'' −> a :: (append l'' l')
```

Innovations majeures : ordre supérieur, typage fort, polymorphismes, types concrets, pattern matching, modules, sémantique rigoureuse, qualité du compilateur,...
Inconvénient : popularité restant (encore) limitée...

Les principes principaux

- Programmation objet : privilégier l'architecture mentale en groupant données et fonctions (méthodes) dans des classes

  Simula, Smalltalk, C++, Objective C, Java, ...

  → sémantique opérationnelle (trouver la bonne méthode)
  sémantique axiomatique (assertions)

- Programmation logique / par contraintes :
  spécifier ce qu'il faut calculer, mais pas comment le calculer

  Prolog, Datalog, CCP, python-constraint, ...

  → sémantique logique (résolution, SAT, etc.)
  solveurs d'équations et inéquations (SMT, etc.)

Les principes principaux

- Programmation parallèle asynchrone : faire coopérer et communiquer des processus sans relation temporelle

  CSP, ADA, OCCAM, CCP, Erlang

  → OS / tâches, threads, partage de mémoire
  → rendez-vous, calculs de processus
  → passages de messages (acteurs)

- Programmation parallèle vibratoire : simuler (plus ou moins) la propagation temporelle du courant dans un circuit

  Verilog, VHDL, System Verilog

  → interprète de référence (pas mieux, hélas)

Les principes principaux

- Programmation parallèle synchrone (réactive) : faire coopérer et communiquer des processus en synchronisme temporel

  Esterel, Lustre, Signal, Statecharts, SCADE, Scratch, ...

  → sémantiques mathématiques classiques ou constructives

- Programmation temps continu : simuler des phénomènes mécaniques ou physiques

  Stateflow, Modelica, ...

→ interprètes de référence (incorrects si continu / discret)

→ travail sémantique majeur en cours (Rennes, ENS Ulm)

Les principes principaux

- Langages de script: enchaînement dynamique de commandes

  sh, bash, Tcl, Perl, php, Javascript (?), Python(?), Ruby(?)

- langages spécialisés (DSL = Domain-Specific Languages)

  parler le langage des utilisateurs
  - tableurs: Excel, Visual Basic
  - expressions régulières : lex, grep, ...
  - grammaires : yacc
  - compilation : make
  - textes : nroff, PostScript, TeX, LaTeX
  - protocoles télécom : SDL
  - génération de tests pour circuits : E
  - formules temporelles : EMC, PSL
  - écriture musicale : Midi, Max, PureData, Antescofo, Giotto, ...
  - etc.

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Les guerres de religion

L'impératif est un nid à bug, en particulier à cause des pointeurs et de la gestion mémoire par l'utilisateur

Mais non, la programmation y est très naturelle, et il suffit d'y ajouter les assertions, les contrats, l'analyse statique, etc.

Le fonctionnel, c'est pour les intellos, et c'est inefficace

Mais non, ça demande juste un peu de culture scientifique, et l'inefficacité est du passé !

Le typage fort et la gestion automatique de la mémoire donnent la sécurité. Et on peut bien mieux raisonner sur les programmes, voire les prouver.

Les guerres de religion

L'objet, c'est la fausse bonne idée : ça a l'air très bien au début, mais il y a plein de problèmes (ex. héritage multiple) et c'est super-verbeux !

Mais non ! l'héritage simple suffit le plus souvent, la programmation est naturelle et proche de l'architecture, elle passe bien à l'échelle, et le résultat est très efficace

La programmation logique et les contraintes, c'est bien joli, mais on ne sait jamais si ça va marcher et combien de temps ça va prendre !

Causez toujours, vous ne savez absolument pas faire ce que nous faisons

Les guerres de religion

La programmation avec des threads, c'est le meilleur moyen de faire des bugs hyper-durs à trouver

Pas faux, mais il on peut aussi apprendre les bons algorithmes distribués et utiliser les bonnes bibliothèques

La programmation synchrone, c'est bien joli, mais c'est une petite niche

Tout est relatif. Ceci ne vaut que si vous regardez votre monde à vous. Le monde de la commande industrielle et celui des circuits sont très vastes !

Les guerres de religion

Les Domain-Specific Languages, c'est bien joli, mais ils sont tous moches et mal définis

Oui, mais ils sont "good enough" pour leur objectif, et utiliser vos outils est impossible pour nous, les programmes devenant compliqués et mal foutus

Et en plus, on peut utiliser les avancées de la sémantique pour rendre les DSLs propres et mathématiquement clairs

Plus intelligent : mélanges harmonieux !

- fonctionnel + impératif + objet : Ocaml, F#
- impératif + asynchrone : CSP, OCCAM, ADA,...
- fonctionnel + asynchone : Erlang
- impératif + synchrone : Reactive C, Scratch
- fonctionnel + synchrone : Reactive ML
- fonctionnel + impératif + synchrone : Hop / HipHop
- contraintes + asynchrone : BioCham
- impératif + fonctionnel + object + asynchrone : Scala, Python
- objet + téléchargement : Java, Python
- fonctionnel + objet + téléchargement : Javascript, Hop, Hopjs

Ces mélanges sont difficiles à réussir,
comme d'ailleurs les bons masalas !

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Les ingrédients communs (il y en a !)

- Les nombres entiers, bornés ou en précision arbitraire
- Les nombres flottants, à la norme IEEE 754
- Les chaînes de caractères, ASCII ou Unicode
- Les records (enregistrements)
- Les pointeurs ou les références
- Les fonctions, y compris récursives
- Les expressions en général
- De plus en plus, les exceptions
- La séquence, les tests, les boucles
- Etc.

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Une différence socialement essentielle :
la syntaxe textuelle

COBOL

```
add 1 to compteur.
add TOTAL-MATIN, TOTAL-SOIR giving TOTAL-JOURNEE.
subtract REMISE from PRIX.
multiply QUANTITE, PRIX-UNITAIRE giving PRIX.
```

FORTRAN

```
! Conversion de degrés en radians
   COEFF = (2.0 * 3.1416) / 360.
   DO DEG = 0, 90
     RAD = DEG * COEFF
     WRITE (*, 20) DEG, RAD
20 FORMAT (' * ',I4,' * ',F7.5,' *')
   END DO
```

80 : une constante de la nature?

LISP (Mc Carthy 1957) : tout en rondeurs

```
(defun append (lst1 lst2)
    (cond
        ((null lst1) lst2
        (T (cons (car lst1) (append (cdr lst1) lst2)))))
=> append
```

ML (Milner 1978), Caml : un look matheux

```
fun rec append l l' =
  match l with
    [] -> l'
  | a :: l'' -> a :: (append l'' l')
```

Algol, Pascal, Ada, Modula, Eiffel, Esterel, etc.

```
module M :
   input I, J;
   output O;
   every I do
      if J then emit O end if
   end every
end module
```

un peu lourd,
mais bien lisible
et bien parenthésé

C, C++, Verilog, Java, Scala, ...

```
int fact (int n) {
   int res = 1
   fo                    res*j;
   re
  }
```

BINGO !

conçu pour de
petites machines (32K)
compact, mais plein
de bizarreries

HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                                "http://www.w3.org/TR/html4/strict.dtd">
<html lang="fr">
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
    <title>Hello world !</title>
  </head>
   <body>
     <p>Hello World !</p>
   </body>
</html>
```

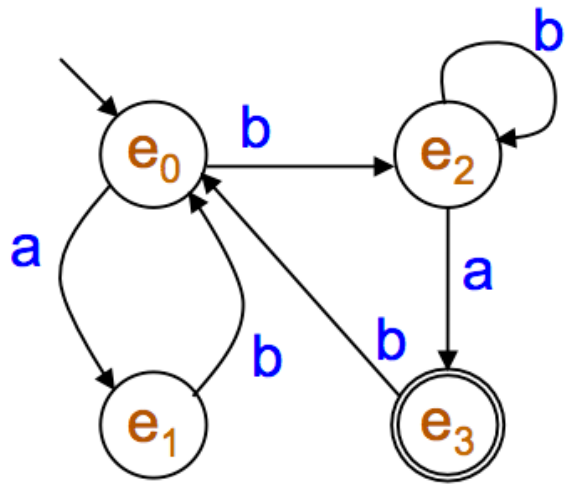
FORTE

```
begin TOOL HelloWorld;

  includes Framework;
  HAS PROPERTY IsLibrary = FALSE;

  forward  Hello;

   -- START CLASS DEFINITIONS

   class Hello inherits from
Framework.Object

   has public method Init;

   has property
      shared=(allow=off, override=on);
      transactional=(allow=off,
                        override=on);
      monitored=(allow=off, override=on);
      distributed=(allow=off, override=on);

  end class;
   -- END CLASS DEFINITIONS
```

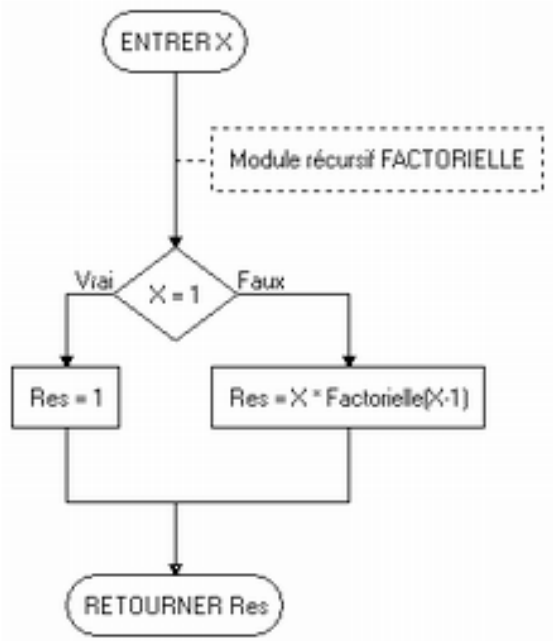
```
--- START METHOD DEFINITIONS

   ------------------------------------------------------
   method Hello.Init
   begin
   super.Init();

   task.Part.LogMgr.PutLine('hello, world');
   end method;
   -- END METHOD DEFINITIONS

   HAS PROPERTY
      CompatibilityLevel = 0;
      ProjectType = APPLICATION;
      Restricted = FALSE;
      MultiThreaded = TRUE;
      Internal = FALSE;
      LibraryName = 'hellowor';
      StartingMethod = (class = Hello, method = Init);

   end HelloWorld;
```

source Wikipedia

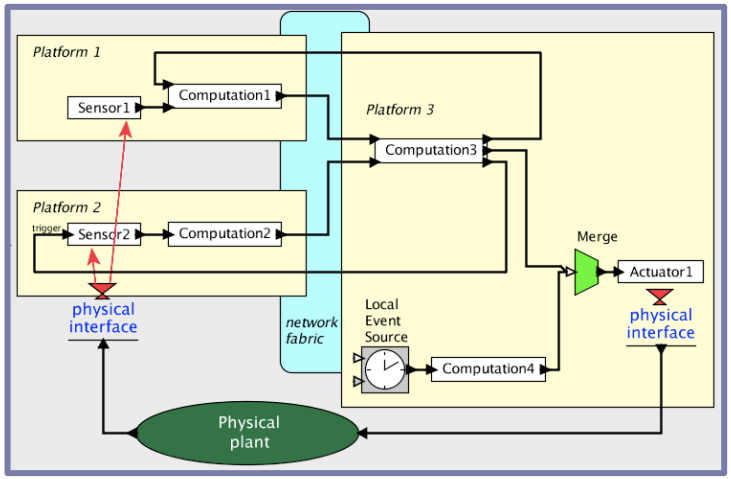
Syntaxes graphiques traditionnelles



automates



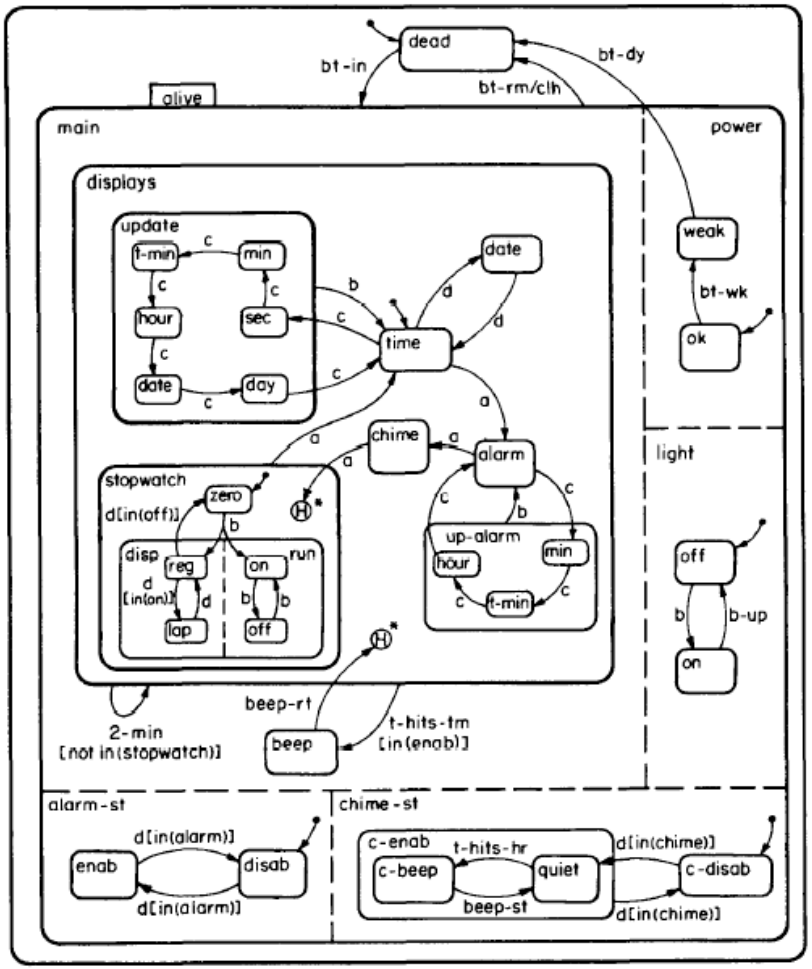
organigrammes

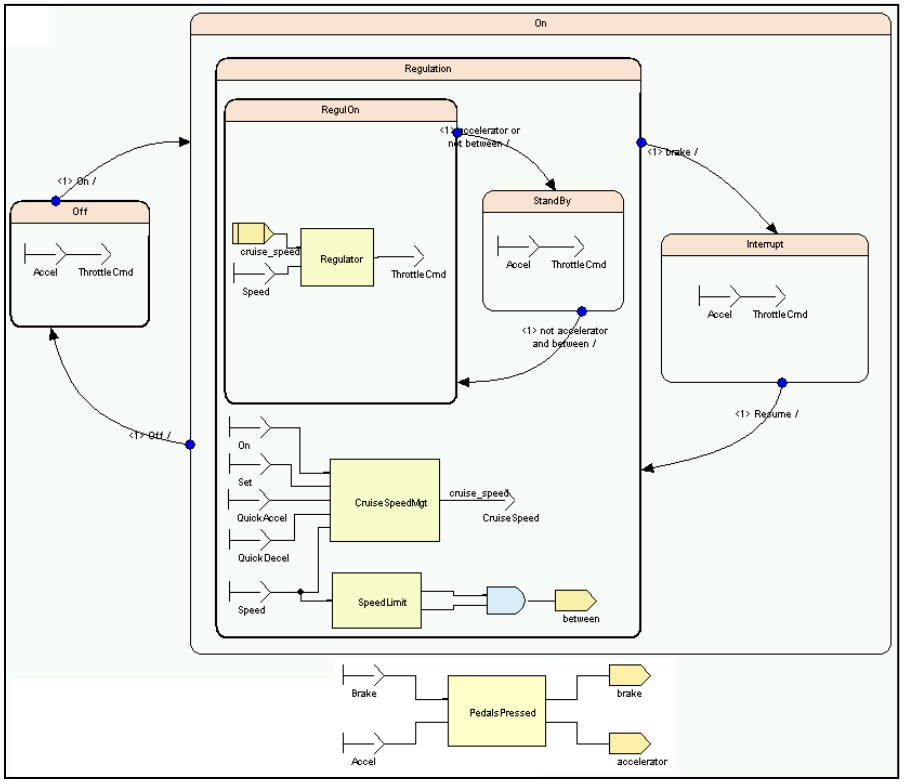


block-diagrams

Et beaucoup d'autres diagrammes,
cf UML / SysML

Syntaxes graphiques hiérarchiques





Statecharts, D. Harel

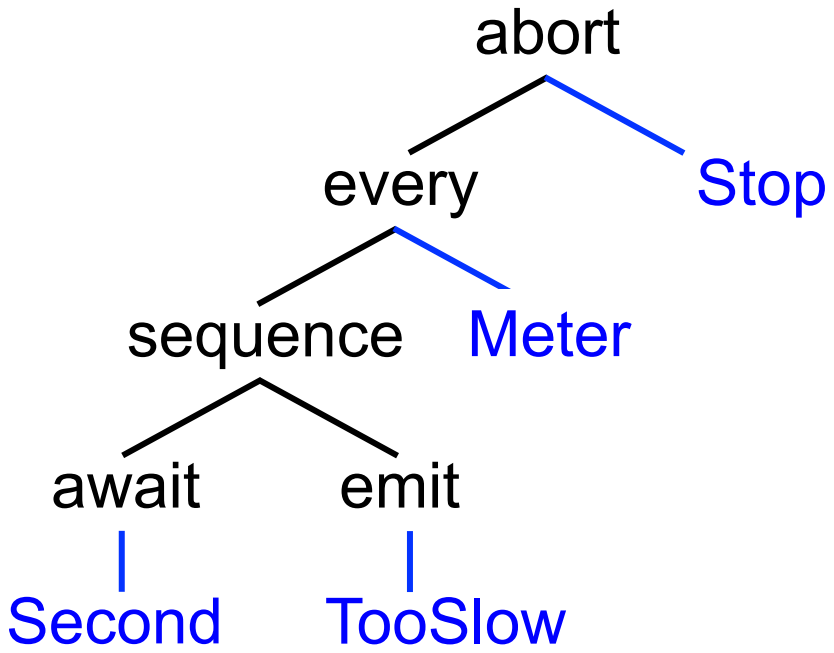
expressivité maximale

SyncCharts (C. André), SCADE 6

rigueur maximale

La syntaxe abstraite, arborescente

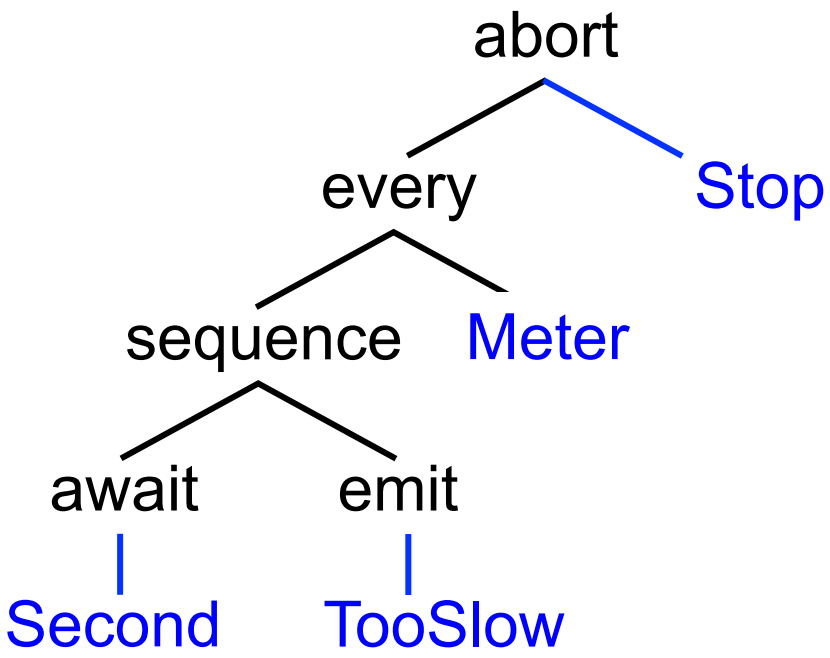
```
abort
   every Meter do
      await Second;
      emit TooSlow
   end every
when Stop
```



La syntaxe abstraite doit commander !

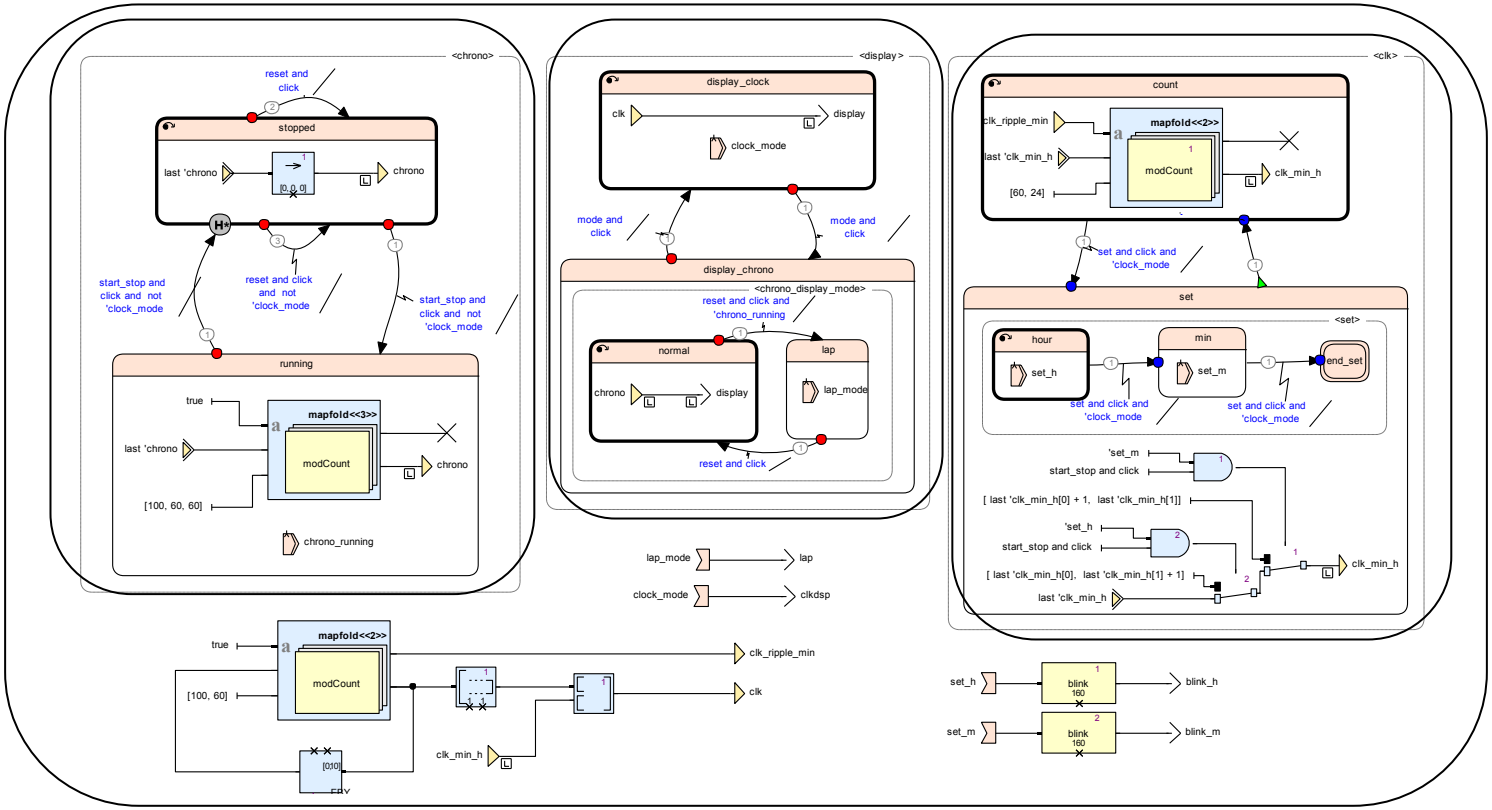
La syntaxe concrète ne doit être qu'une façon commode pour nous de représenter la syntaxe abstraite

XML, une syntaxe concrète pour la syntaxe abstraite




```
<abort sig="Stop">
   <every sig="Meter">
      <sequence>
         <await sig="Second"/>
         <emit sig="TooSlow"/>
      </sequence>
   </every>
</abort>
```

Ne pas laisser la syntaxe dicter la sémantique !



SCADE : vrai parallélisme, associatif-commutatif

Stateflow : faux parallélisme ☹

ordre d'exécution déterminé par le dessin !!!!!!!!!!!

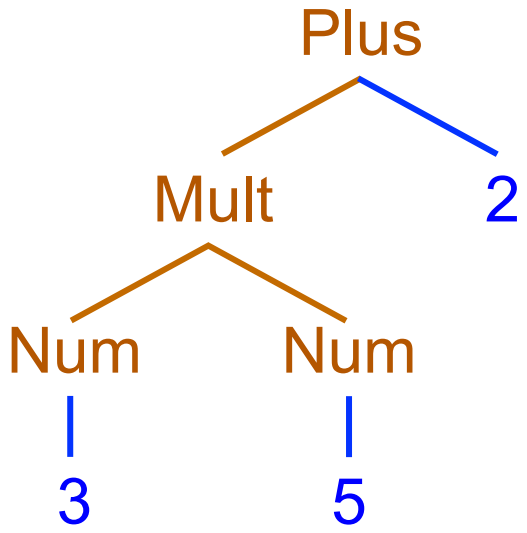
ex. : position (x,y) dans l'ordre de lecture

ordre de création des boîtes, numérotation manuelle...

Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

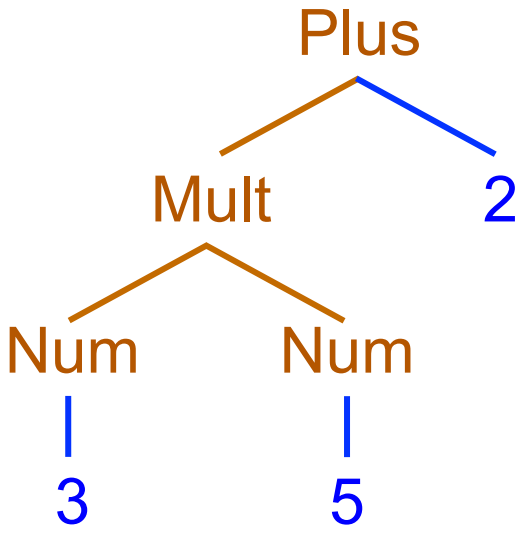
Style fonctionnel : OCaml




```
type Exp =
  Num of int
| Plus of Exp * Exp
| Mult of Exp * Exp ;

fun Eval (e) =>
  match e with
     Num n –> n
   | Plus e1 e2 –> (Eval e1) + (Eval e2)
   | Mult e1 e2 –> (Eval e1) * (Eval e2) ;

Eval (Plus (Mult (Num 3) (Num 5)) (Num 2)) ;;
 → 17 : int
```

Style Objet : C++

```
#include <iostream>

// Expressions
class Exp {
public:
   virtual int Eval () = 0;
};

// Number Expressions
class NumExp : public Exp {
   int n;
public:
   NumExp (int m) : n(m) { } // constructor
   int Eval () {
      return n;
   }
};
```



Style Objet : C++

```
// Binary Expressions
class BinExp : public Exp {
protected:
    Exp& exp1;
    Exp& exp2;
public:
    BinExp (Exp& e1, Exp& e2) : exp1(e1), exp2(e2) { }
};

// Plus expressions
class PlusExp : public BinExp {
public:
    PlusExp (Exp& e1, Exp& e2) : BinExp (e1, e2) { }
    int Eval () {
        return exp1.Eval() + exp2.Eval();
    }
};
```

Style Objet : C++

```
// Mult Expressions
class MultExp : public BinExp {
public:
   MultExp (Exp& e1, Exp& e2) : BinExp (e1, e2) { }
   int Eval () {
     return exp1.Eval() * exp2.Eval();
   }
};

// Let's go !
int main () {
   Exp& Deux = *new NumExp(2);
   Exp& Trois = *new NumExp(3);
   Exp& Cinq = *new NumExp(5);
   Exp& MyExp = *new PlusExp(*new MultExp(Trois, Cinq), Deux);
   std::cout << MyExp.Eval() << std::endl;
}
```

→ 17

Deux extensibilités orthogonales !

```
fun Eval e =>
   match e with
     Num n -> n
   | op1 e1 e2 -> ...
   | op2 e1 e2 -> ...
   | op3 e1 e2 -> ...
   | op4 e1 e2 -> ...
   | op5 e1 e2 -> ... ;
```

Compact, mais :

beaucoup de cas
⟹ peu lisible

Ajouter un opérateur
⟹ modifier l'intérieur du source

Exp.h

```
class Op : Op {
   ...virtual int Eval () = 0;
};
class Op1 : public Op {
   ... int Eval ();
};
class Op2 : public Op {
   ... int Eval ();
};
class Op3 : public Op {
   ... int Eval ();
};
```

Exp.cpp

```
int Op1::Eval () {...}
int Op2::Eval () {...}
int Op3::Eval () {...}
```

Plus verbeux, mais ajouter un opérateur demande simplement d'ajouter du source

Typé ou non typé ? C'est très différent !

C++

```
#include <iostream>

class Point {
public:
    int x;
    int y;
    Point (int xx, int yy) : x(xx), y(yy) { }
};

int main () {
    Point& p = *new Point(1, 5);
    std::cout << p.z << std::endl;
}
```

typerr.cpp:12:20: error: no member named 'z' in 'Point'

Python

```
class Point:
    def __init__(self,xx,yy):
        self.x = xx
        self.y = yy

p = Point(1,5)
p.z = 7      # accepté !
p.x+p.y+p.z
```

→ 13

p.z = 7 a créé dynamiquement un nouveau champ z pour p !
Attention aux fautes d'orthographe...

Typé ou non typé ? C'est très différent !

C++

```
#include <iostream>

class Point {
public:
    int x;
    int y;
    Point (int xx, int yy) : x(xx), y(yy) { }
};

int main () {
    Point& p = *new Point(1, 5);
    std::cout << p.z << std::endl;
}
```

```
typerr.cpp:12:20: error: no member
named 'z' in 'Point'
```

Python

```
class Point:
    def __init__(self,xx,yy):
        self.x = xx
        self.y = yy

p = Point(1,5)
p.z = 7      # accepté !
p.x+p.y+p.z
```

→ 13

```
q = Point(2,6)
q.z
```

```
AttributeError: Point instance
has no attribute 'z'
```

HOP : ajoute des champs dynamiquement aux classes

Typage et vérification formelle

- Typage dynamique : arrêter l'exécution en cas de détection d'opération illégale (choux+carotte)
  - Scheme, Python, etc.
- Typage statique : dès la compilation, classifier les expressions selon les valeurs dénotées, et assurer la compatibilité des arguments des opérations
- Vérification formelle : dès la compilation vérifier des propriétés quelconques (cf. séminaire de T. Jensen)

Que peut-on savoir sur le programme sans l'exécuter, et à quel prix?

Intuition: le typage doit être très rapide

Niveaux de typage

1. Interdire choux+carottes
2. Typer les fonctions et leurs applications : C, Java, etc.
3. Typer l'ordre supérieur (fonctions de fonctions)
4. Accepter les types polymorphes (paramétriques)
5. Sous-typage (langages objets)
6. Inférence de types : trouver les bons types même lorsqu'ils ne sont pas écrits par le programmeur
7. Typage des modules et APIs : *Programming in the Large*
8. Niveau maximal actuel : Coq, cf. cours du 18 mars 2015
   types dépendants, typage garantissant la terminaison, mais l'inférence générale n'est plus décidable

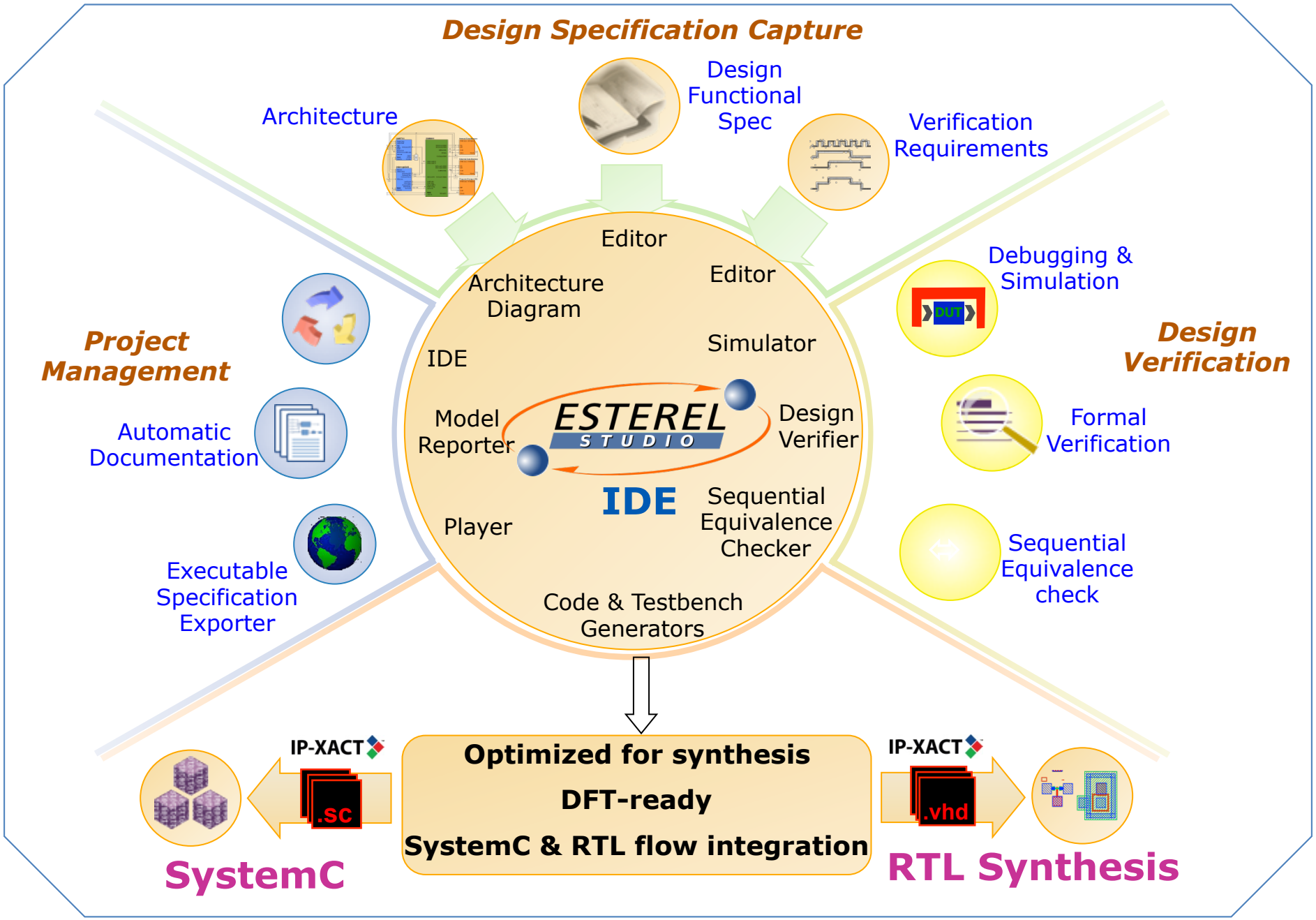
ML : ordre supérieur, polymorphisme, inférence

Appliquer une fonction aux éléments d'une liste :

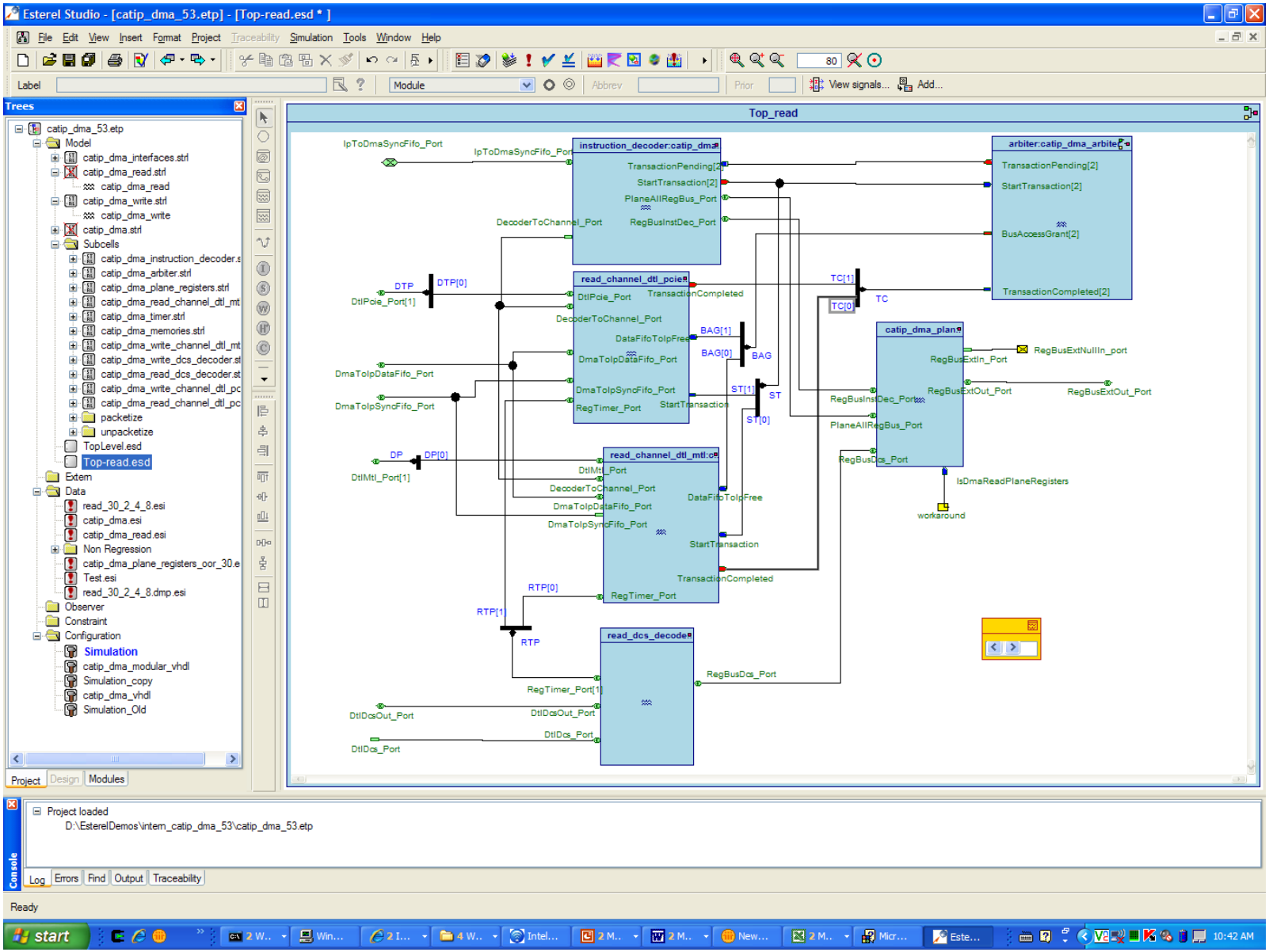
```
# let rec map f l = match l with
                      [] -> []
                    | hd :: tl -> (f hd) :: (map f tl) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map (fun x -> x+1) [1; 2; 3] ;;
- : int list = [2; 3; 4]

# let rec map f l = match l with
                      [] -> []
                    | hd :: tl -> (f tl) :: (map f hd) ;;
Error : This expression has type 'a but an expression
was expected of type 'a list
The type variable 'a occurs inside 'a list
```

Agenda

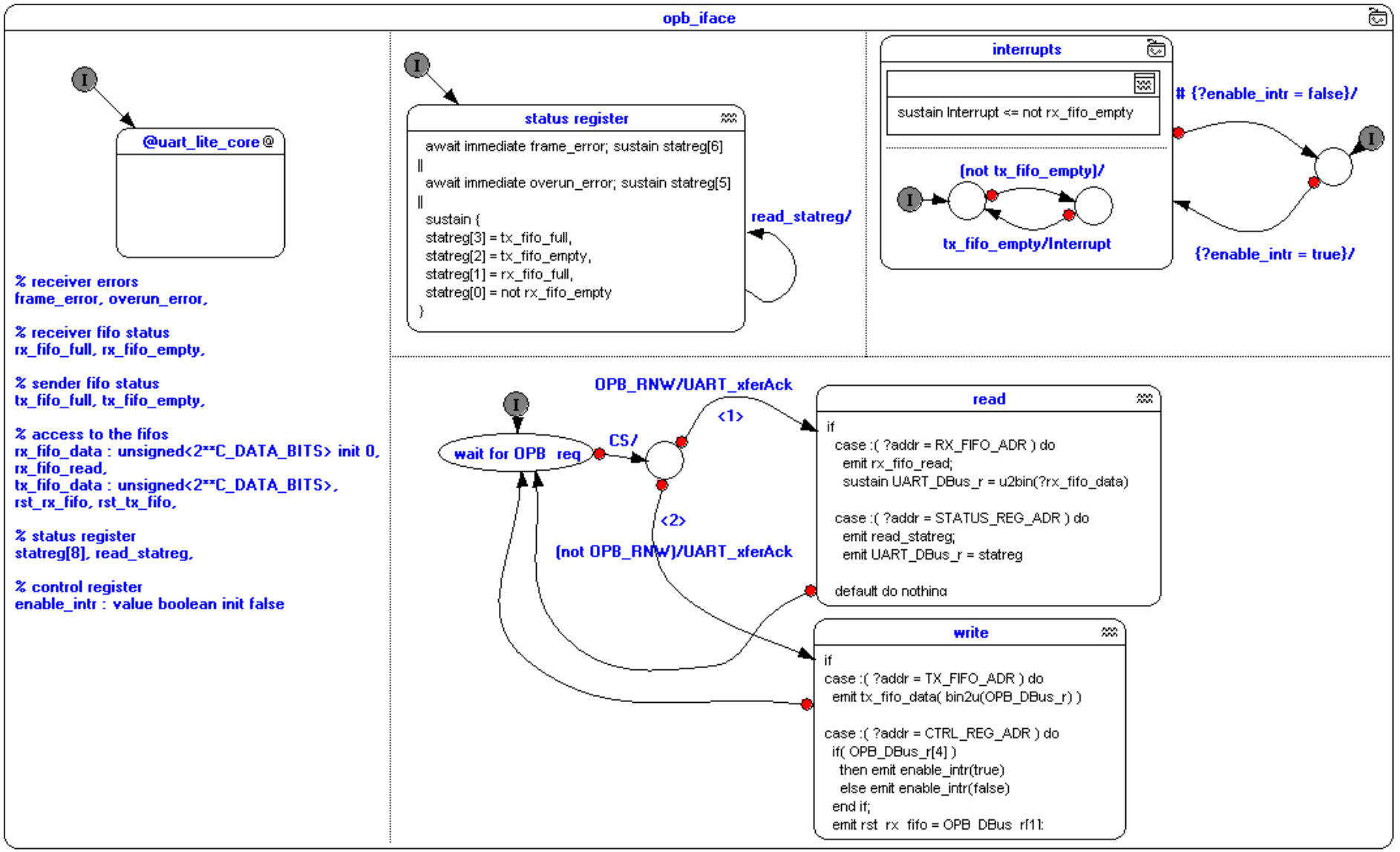
1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion


Design Specification Capture
Architecture
Design Functional Spec
Verification Requirements
Editor
Architecture Diagram
Editor
Simulator
IDE
Model Reporter
ESTEREL STUDIO
Design Verifier
IDE
Player
Sequential Equivalence Checker
Code & Testbench Generators
Project Management
Automatic Documentation
Executable Specification Exporter
Debugging & Simulation
DUT
Design Verification
Formal Verification
Sequential Equivalence check
IP-XACT
.sc
SystemC
Optimized for synthesis
DFT-ready
SystemC & RTL flow integration
IP-XACT
.vhd
RTL Synthesis

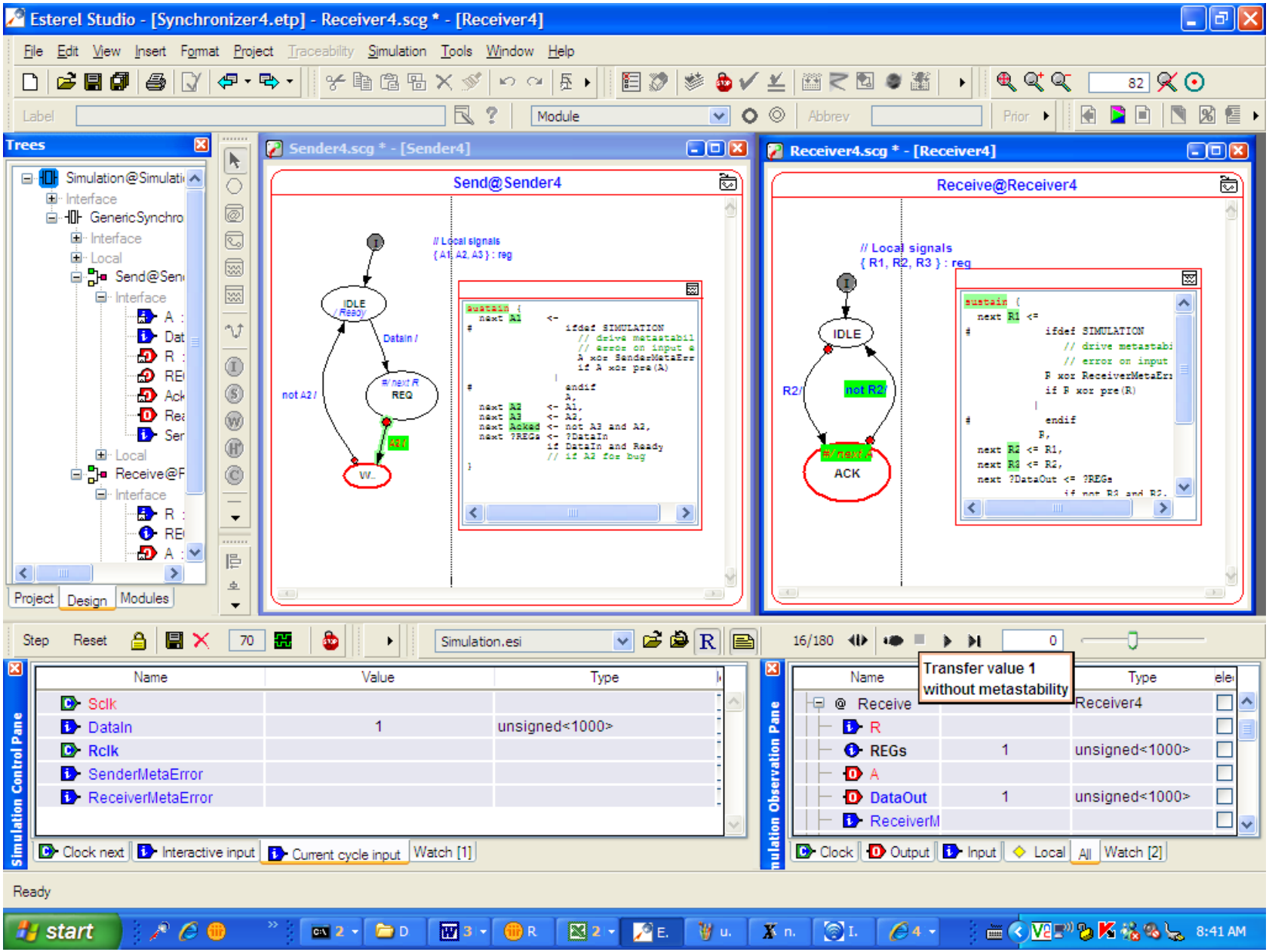
Diagrammes d'architecture



Automates hiérarchiques graphiques / textuels



Simulation avec animation du code source



Agenda

1. Les langages sont partout
2. Les principes principaux
3. Les guerres de religion
4. Les ingrédients communs
5. Syntaxes concrètes, syntaxe abstraite
6. Styles, types et génie logiciel
7. L'outillage
8. Conclusion

Qu'est ce qui fait le succès ou l'échec ?

- Un design harmonieux, une sémantique claire
- De bonnes documentations et implémentations
- De bons environnement de programmation
- Une liaison fine avec le test et la vérification formelle

- Une bonne adaptation à un type de problèmes
- Une communauté active d'utilisateurs
- Une installation complètement triviale
- De bons outils de formation
- Un bon succès médiatique

Mais surtout,
apporter une solution à un problème brûlant !

téléchargement → Java, Web → Javascript,
Scripting → tcl, perl, python, Ruby, etc..
enseignement → Scratch

Un solide problème de poule et d'œuf !

- Aux USA et en Asie, good enough et populaire valent mieux qu'intelligent et scientifique !
- Et l'industrie n'y embauche que des gens déjà formés

sources lesbrinsdherbes.org

La musique des langages

Les langages, c'est comme les instruments de musique.
On peut en aimer plusieurs, mais n'en maîtriser que peu
Et il n'est pas si facile d'en trouver des bons !

- C : le piano. Quand on tape sur les touches, ça fait toujours des sons, mais pas forcément de la musique
- Caml : le violon. Il faut apprendre à fabriquer le son, mais une fois qu'on sait, quel son !
- Esterel : la batterie. L'important est de bien synchroniser les mains et les pieds
- Python : le synthétiseur. On peut brancher tout sur tout, ce qui rend possible de faire des sons étonnants, mais de là à faire celui qu'on veut....