Principes des langages de programmation

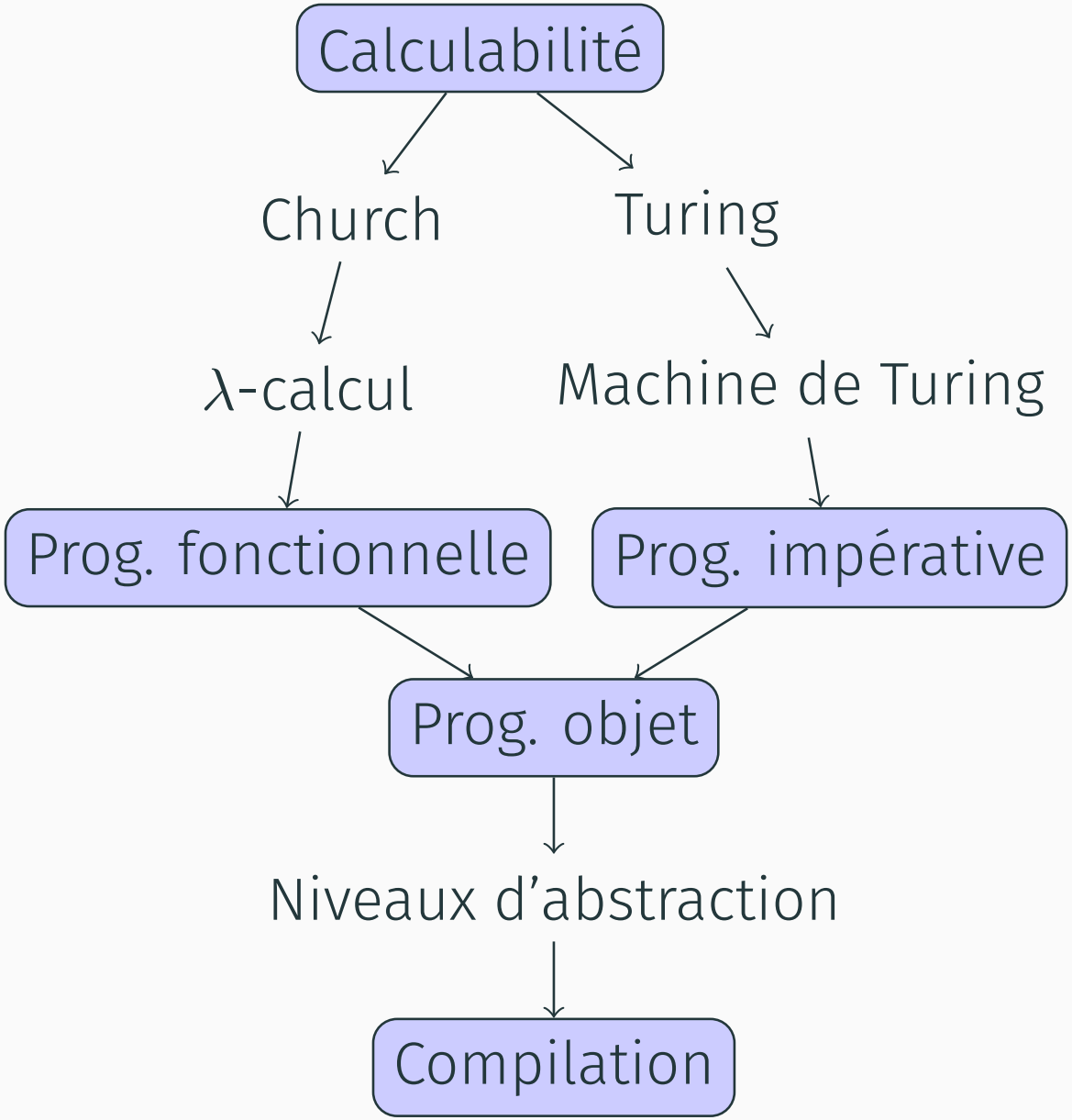
Une introduction brève et incomplète

Antoine Bertout

28/03/2018

Université de Poitiers

Plan


Calculabilité
Church
Turing
λ-calcul
Machine de Turing
Prog. fonctionnelle
Prog. impérative
Prog. objet
Niveaux d'abstraction
Compilation

Calculabilité

- On va s'intéresser à des moyens d'expressions des algorithmes, les langages, mais que calcule-t-on?
- Les scientifiques ont cherché à formaliser la notion intuitive d'algorithme à travers celle de *fonction calculable*.

Fonction calculable

Une fonction calculable est une fonction f par laquelle, l'application d'un ensemble fini d'instructions (un algorithme) pour une entrée x, produit la sortie $f(x)$.

Calculabilité

Plusieurs personnages de renom se sont penchés dans les années 1930 sur des caractérisations mathématiques des fonctions calculables, notamment :

- **Alan Turing** pour qui les fonctions calculables sont toutes celles qui peuvent être calculées par une machine abstraite éponyme, dite *machine de Turing*.
- **Alonzo Church** qui définit un modèle de calcul appelé *λ-calcul*, dans lequel toutes les fonctions calculables sont représentables et...calculables.

La thèse de Church-Turing[1] unifie ces deux approches, en montrant que tout ce qui est exprimable dans l'une est exprimable dans l'autre, i.e. qu'elles sont équivalentes.

[1] exprimée par Kleene :)

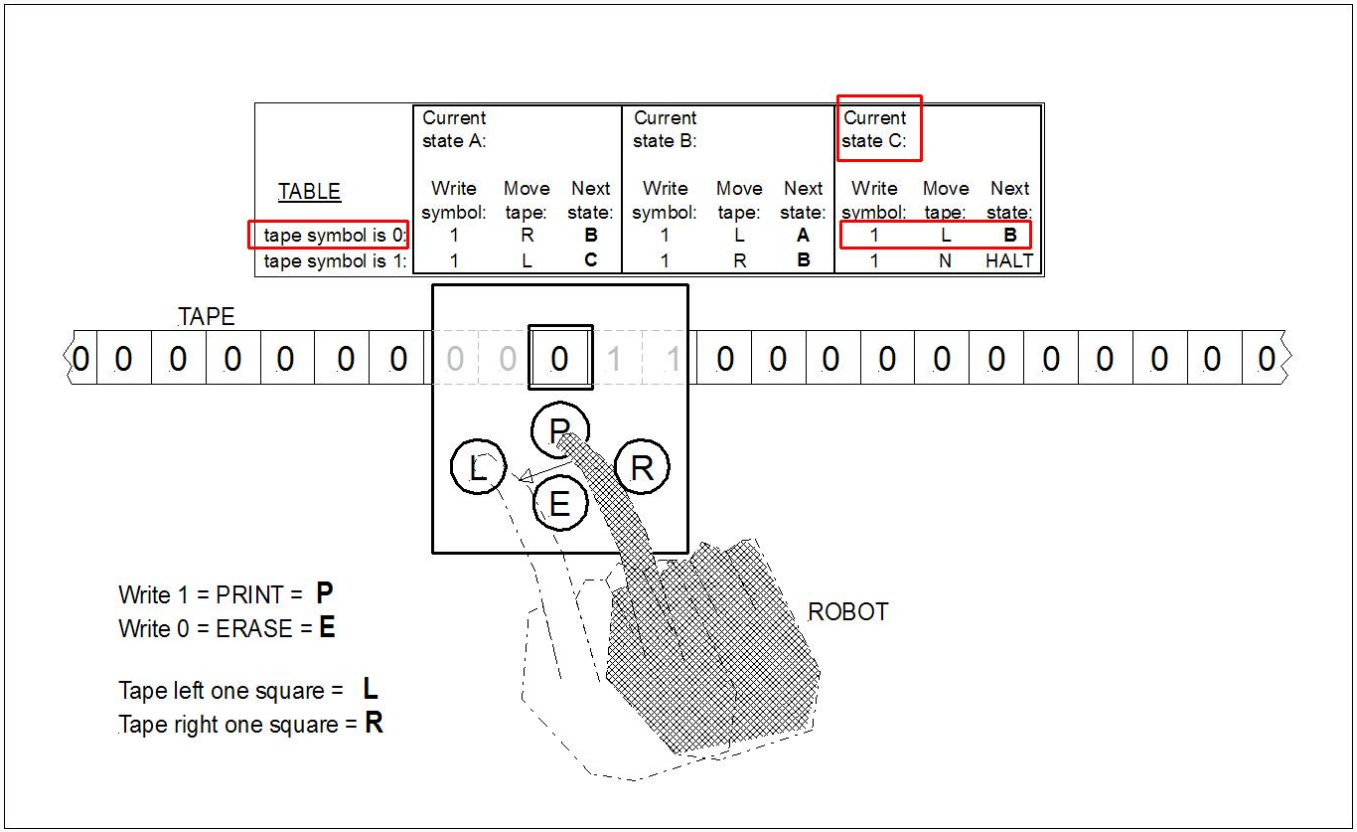
Machine de Turing

Alan Turing (1912-1954)

Figure 1: 2h46 au marathon.

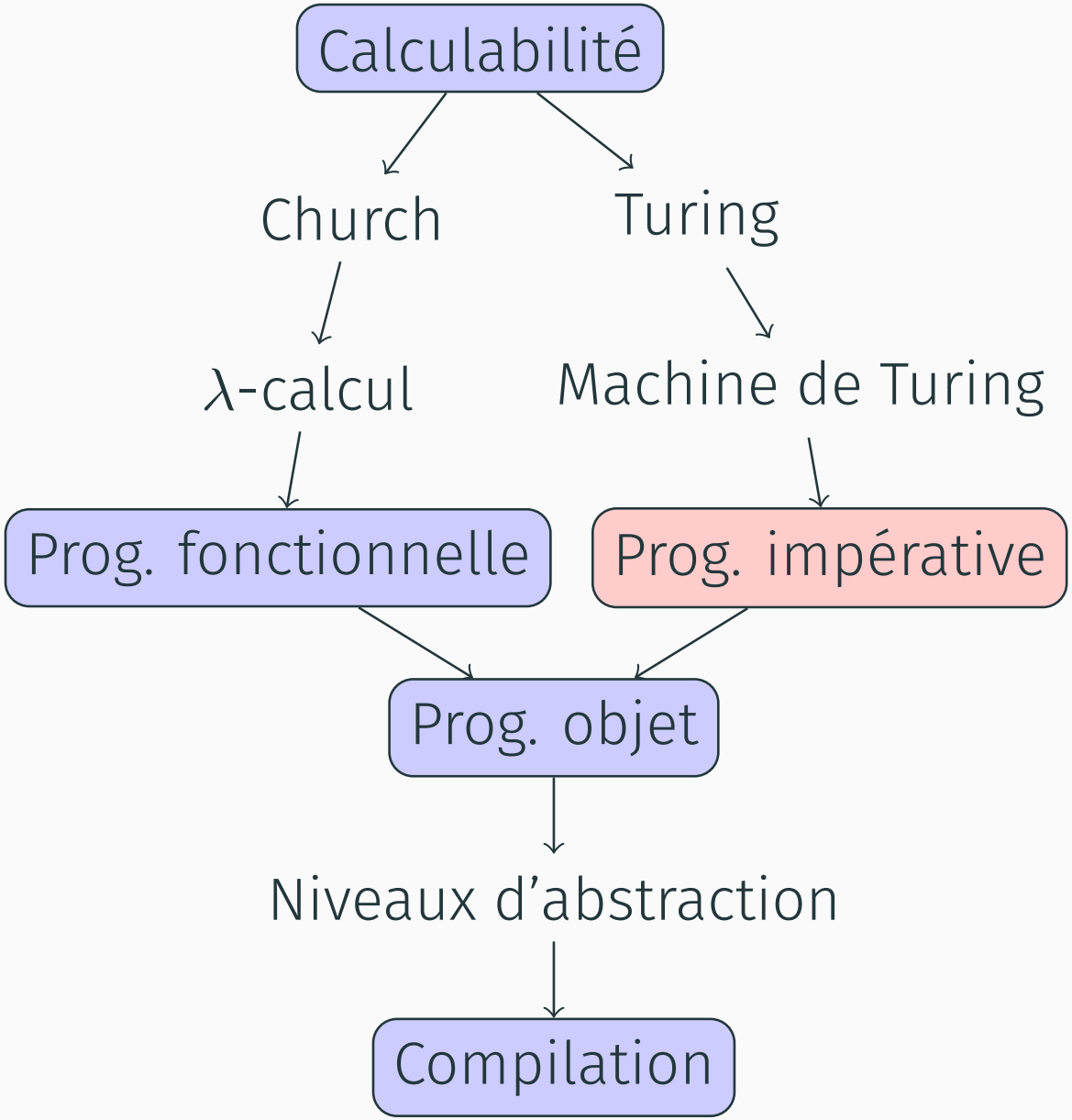
Machine de Turing

La machine de Turing est mécanique…mais abstraite!



La programmation impérative s'inspire du fonctionnement de la machine de Turing. On parle d'ailleurs de langages Turing-Complet.

Plan


Calculabilité
Church
Turing
λ-calcul
Machine de Turing
Prog. fonctionnelle
Prog. impérative
Prog. objet
Niveaux d'abstraction
Compilation

Noyau impératif

En programmation impérative (Python par ex.), on décrit **comment** le calcul va être réalisé, étape par étape à l'aide :

- d'affectations, de boucles, de déclarations, de séquences, de tests, etc.
- de la récursivité

Ces opérations reviennent à écrire/lire des valeurs, transiter d'états à d'autres.

Quelques langages impératifs : FORTRAN, COBOL, Pascal, C...

Lambda calcul

Le λ-calcul [2] est un système formel de calcul minimaliste fait de définitions et d'applications de fonctions. C'est un modèle abstrait d'un langage de programmation fonctionnelle.

- carre(n) = n * n $\Rightarrow$ fonction qui, pour chaque n retourne n^2
- en raccourci : carre = λn. n * n
- carre(2) est la valeur qui résulte du remplacement de la variable n dans λn. n * n qui donne 2 * 2

En λ-calcul, **tout est fonction**, les arguments sont des fonctions et le résultat retourné est une autre fonction.

[2] ici non typé et non pure

Lambda calcul

Les termes du lambda-calcul sont de la forme :

- x (variable)
- $(\lambda x.M)$ où M est un terme (abstraction)
- (MN) où M et N est un terme (application)

Dans le terme $\lambda x.M$, x est la variable liée (*argument*) M le *corps* de la fonction.

Fonctions simples :

- La fonction identité $\lambda x.x$ ($\Leftrightarrow \lambda o.o$)
- La fonction constante $\lambda x.c$

Lambda calcul

En lambda-calcul, on calcule en effectuant des **réductions**.

Exemple : la fonction qui a x associe 2x + 5

$$(\lambda x.2x + 5)6$$
$$(12 + 5)$$
$$17$$

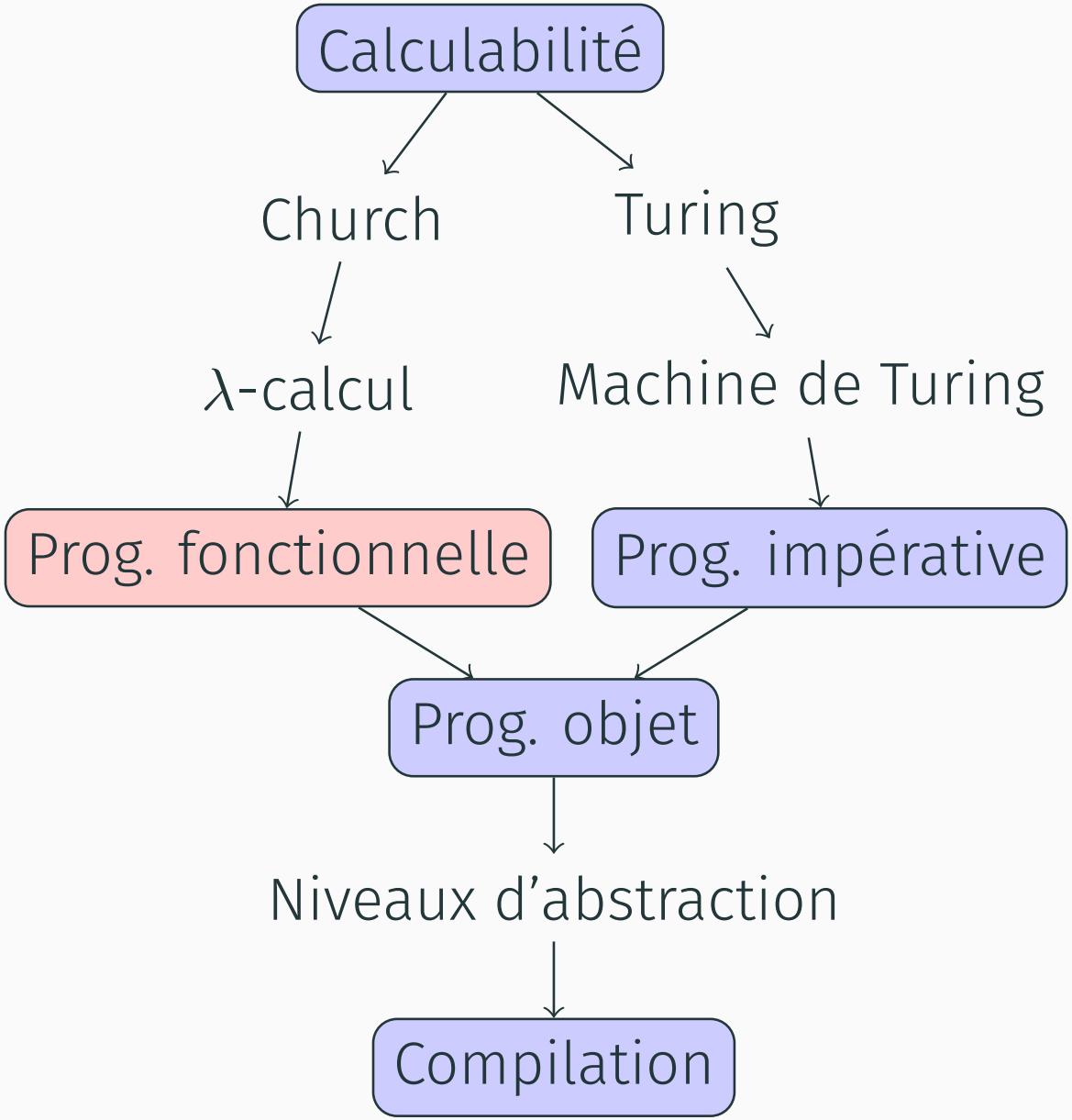
En λ-calcul, les fonctions ne prennent qu'un argument.

Pour "simuler" plusieurs arguments, l'application de la première fonction renvoie une autre fonction qui prend le deuxième argument et produit le résultat.

Exemple : une fonction add(x,y) $\Rightarrow x + y$

$$(\lambda.x\lambda.y(x + y))(3)(5)$$
$$\lambda.y(3 + y)5$$
$$(3 + 5)$$
$$8$$

Plan


Calculabilité
Church
Turing
λ-calcul
Machine de Turing
Prog. fonctionnelle
Prog. impérative
Prog. objet
Niveaux d'abstraction
Compilation

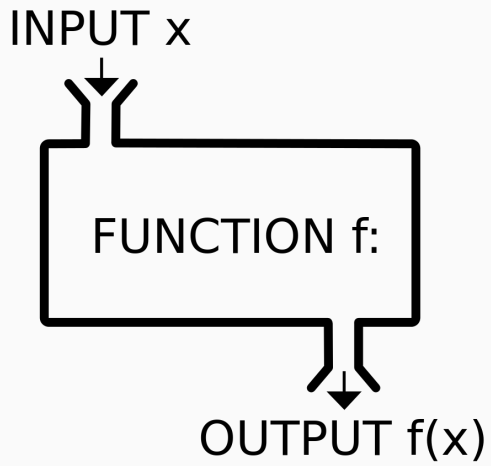
Programmation fonctionnelle

On peut classer la programmation fonctionnelle dans la famille de la programmation déclarative : on ne décrit pas **comment** on va calculer mais ce que l'on (**quoi**) calcule.

La logique de la programmation fonctionnelle est basée sur le λ-calcul de Church.

Programmation fonctionnelle : Philosophie

Elle privilégie les fonctions dites *pures*, qui ne font que prendre des valeurs en *entrée*, retourner des valeurs en *sortie* et … rien d'autre!



$\Rightarrow$ on évite les effets de bord, par ex. :

- Modifier une variable non locale à la fonction
- Réassigner des valeurs à des variables (a = 5; a = a -3)
- Modifier une structure de données sur place (lst.append() en Python!)

Programmation fonctionnelle

Quelques concepts liés à la programmation fonctionnelle :

- Les fonctions sont des valeurs de première classe : passables en paramètre de fonction
- Immuabilité des ”variables” et des structures de données
- Filtrage par motif
- Fonctions d’ordre supérieur (map, fold, etc.)

Quelques langages fonctionnels : Lisp, ML, Haskell, OCaml, Scala...

Fonctions comme argument

```
def cri(text):
    return text.upper()

def murmure(text):
    return text.lower()

def parler(f,text):
    return f(text)

>>>print(parler(cri,"Coucou"))
COUCOU
```

Immuabilité

Comment écrire un code sans modifier de variable?

Avec la récursivité par exemple.

```
def factoriel(n):
    i = 1
    acc = 1
    while i<=n:
        acc = acc * i
        i += 1
    return acc
```

```
def factoriel(n):
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

Ou avec les fonctions d'ordre supérieur.

Fonctions d'ordre supérieur : Exemples

map : construit une nouvelle liste en appliquant une fonction sur tous les éléments de la liste.

```
>> val nombres = List(1,2,3,4,5)
>> nombres.map(x => x + 2)
res0: List[Int] = List(3, 4, 5, 6, 7)
```

fold : préfixe la liste par 1 (valeur de départ) et applique une opération binaire sur la valeur accumulée et la valeur courante parcourue.

```
>> nombres.fold(1)((x, y) => x * y)
res2: Int = 120
```

$((((1 \times 1) \times 2) \times 3) \times 4) \times 5$

Filtrage par motif

- Le filtrage par motif (pattern matching) permet de reconnaître une structure particulière dans un argument et d'opérer en conséquence.
- Ressemble aux structures de type *switch...case* mais en plus puissant

Filtrage par motif

Exemple : une fonction qui convertit un entier valant 1 ou 0 en sa représentation booléenne sous forme de chaîne de caractères.

```python
def binToStr(binar):
    if binar == 0:
        return "False"
    elif binar == 1:
        return "True"
    else return "Error"  # else inutile
```

```scala
def binToStr(binar:Int): String = binar match {
  case 0 => "False"
  case 1 => "True"
  case _ => "Error"
}
```

Filtrage par motif

Très utile pour les définir les fonctions récursives, par exemple pour calculer la longueur d'une liste.

Sans pattern matching

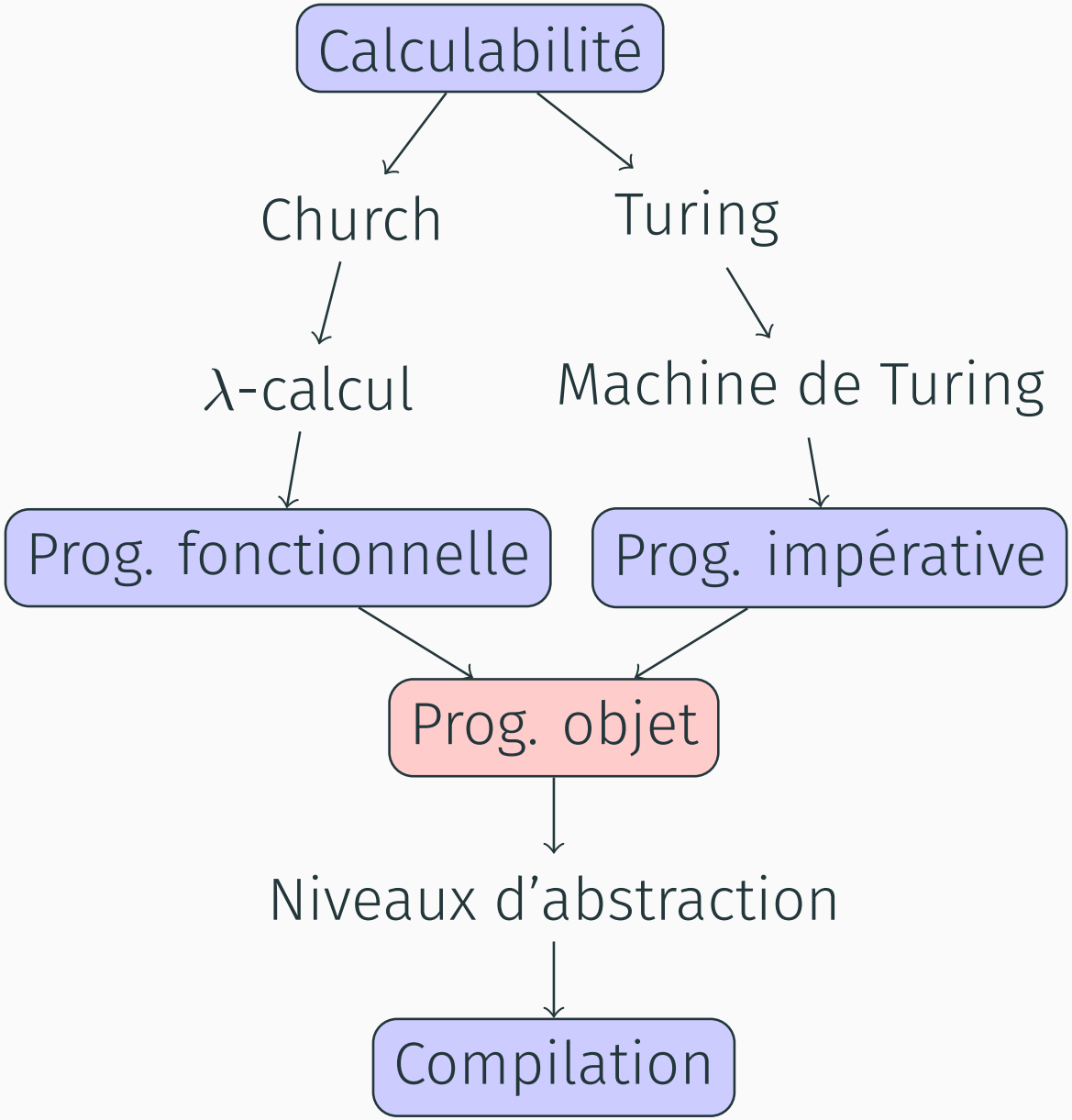
```
def len(lst: Seq[Int]): Int = {
    if(lst.isEmpty) return 0 //return
    return len(lst.tail) + 1
}
```

Avec...

```
def len(lst: Seq[Int]): Int = lst match{
  case Nil => 0
  case head::tail => len(tail) + 1
}
```

Peut même être directement utilisé sur les types des objets.

Plan


Calculabilité
Church
Turing
λ-calcul
Machine de Turing
Prog. fonctionnelle
Prog. impérative
Prog. objet
Niveaux d'abstraction
Compilation

Programmation objet

Une définition

La programmation objet est une manière d'organiser, de structurer le code. Un objet est défini par ses **attributs** et des **comportements**.

Avantages

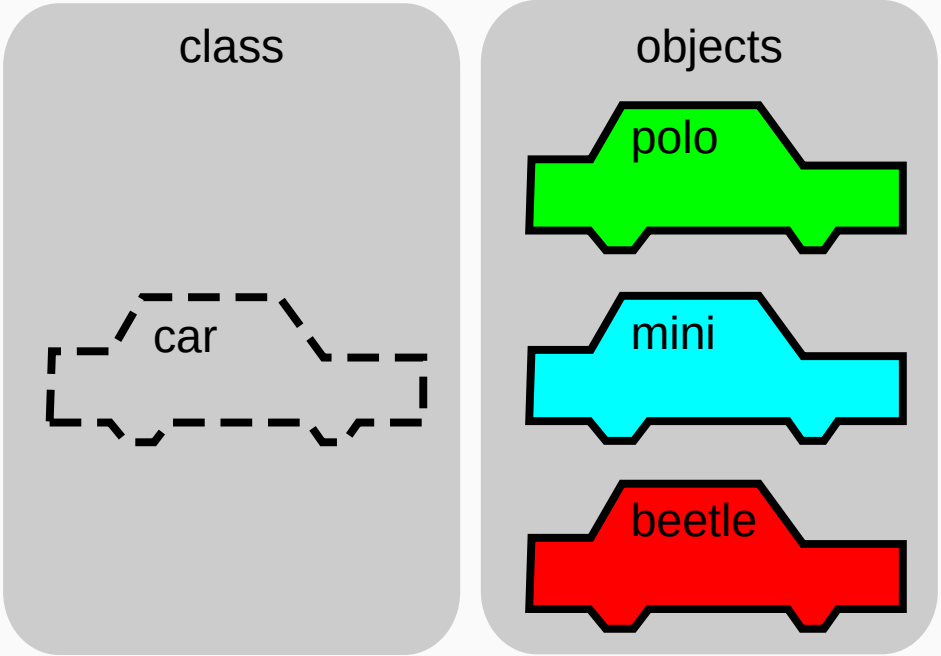
- Meilleure organisation du code (modularité)
- Modification et ré-utilisation du code facilitées
- Manière de penser le problème plus "naturelle"
- etc.

Programmation objet : Classes et objets

- Une **classe** : une entité qui contient des variables (attributs) et des méthodes (comportements)
- Un **object** : une instance de classe

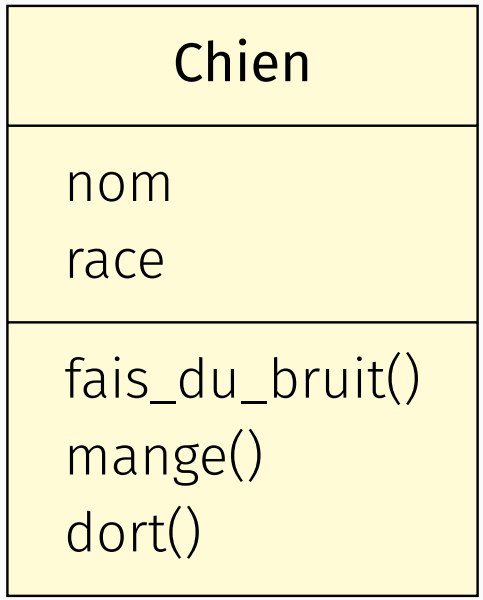
La classe : une usine à objets

La *classe* peut être vue comme un modèle, un patron que l'on va concrétiser par la fabrication d'*objets*.



Programmation objet

Représentation du concept de **Chien**.



- *nom* et *race* sont les **attributs** du chien.
- Les méthodes *mange()*, *dort()* et *fais_du_bruit()* décrivent des **comportements** du chien.

Programmation objet : Classes et objets

Implémentation en Python et *instanciation* de ”vrais” chiens.

```
class Chien:
    def __init__(self, nom, race): #constructeur
        self.nom = nom
        self.race = race

    def fait_du_bruit(self):
        print('Je suis '+self.nom + ': Ouaf Ouaf!')

    def mange(self):
        print('Miam!')

    def dort(self):
        print('ZZzzzzzzz!')

chien1 = Chien('Laïka','bâtard');chien1.fait_du_bruit()
chien2 = Chien('Milou','fox-terrier');chien2.fait_du_bruit()
print(chien2.race)
```

Programmation objet : Classes et objets

Un chat n'est pas si différent...

```
class Chat:
    def __init__(self, nom, race):
        self.nom = nom
        self.race = race

    def fait_du_bruit(self):
        print('Je suis '+self.nom + ': Miaou!')

    def mange(self):
        print('Miam!')

    def dort(self):
        print('ZZzzzzzzz!')

chat1 = Chat('Tombili','turc')
chat2 = Chat('Azraël','roux-blanc-noir-rouge')
```

Programmation objet : Héritage

Les classes Chien et Chat partagent des attributs et des comportements communs.

Le concept d'**héritage** est une relation qui permet d'étendre les comportements et les attributs d'une classe existante.

- Les deux sous-classes vont hériter de la classe *Animal*.
- Les implémentations des méthodes *dort()* et *mange()* sont communes, elles sont réutilisées.
- On va juste redéfinir la méthode *fais_du_bruit()* pour les classes *Chat* et *Chien*. On peut implémenter un comportement par défaut dans *Animal* et redéfinir la méthode quand cela est nécessaire.
- Toutes les classes héritent de la classe *Object*.

Programmation objet : Héritage

Les classes *Chat* et *Chien* héritent de la classe *Animal*.

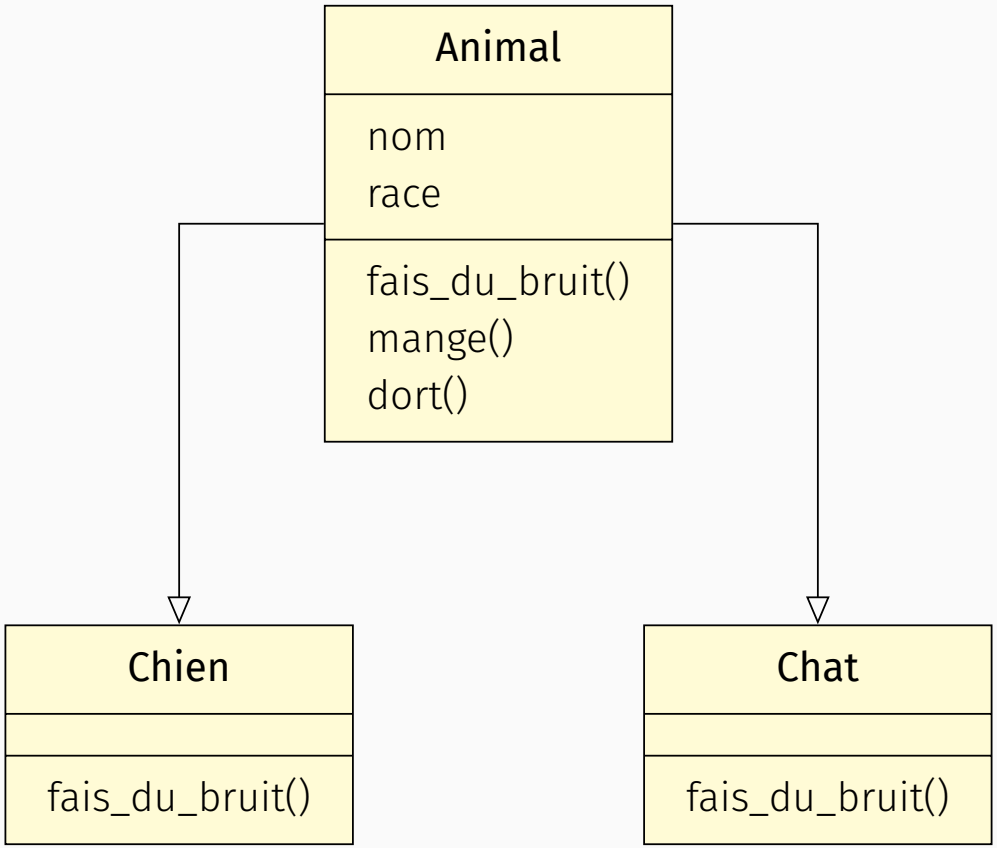


Figure 2: Diagramme de classes.

Programmation objet : Héritage

```
class Animal:

    def __init__(self, nom, race):
        self.nom = nom
        self.race = race

    def mange(self):
        print('Miam!')

    def dort(self):
        print('ZZzzzzzzz!')

    def fait_du_bruit(self):
        pass
```

Programmation objet : Héritage

```
class Chat(Animal):

    def fait_du_bruit(self):
        print('Je suis '+self.nom + ': Miaou!')

chatherite = Chat('Tombili','turc')
chatherite.mange()
```

Programmation objet : Polymorphisme

L'héritage introduit ici la notion de **polymorphisme** (ou généricité).

Intuitivement

Je peux écrire une méthode générique à tous les animaux sans me soucier de l'implémentation détaillée, à savoir s'il s'agit d'un chat, d'un chien ou d'un boa.

```
def bruit_ambiant(liste):
    for anim in liste:
      anim.fait_du_bruit()

zoo = [chien1, chien2, chat] #liste d'objets de type Animal
bruit_ambiant(zoo)
```

Programmation objet : Méthodes et attributs de classe

Il est possible de définir des attributs et des méthodes propres à la classe et partagés par toutes les instances de la classe.

```
class Animal:
    population = 0

    def __init__(self, nom, race):
        self.nom = nom
        self.race = race
        Animal.population += 1
    #en sus de mange(),dort(), etc.

    @classmethod
    def toomuch(cls):
        return cls.population > 10

    @staticmethod
    def convert_age(birth):
        return date.today().year - birth.year
```

Méthodes d'instances ou statiques?

Le nettoyage des chaînes de caractères dans le TP de cryptographie faisait intervenir des méthodes statiques et des méthodes d'instance de classe.

```
def clean(t):
  s = t.lower()
  table = str.maketrans("àéïôùÿç","aeiouyc")
  trans = s.translate(table)
  up = trans.upper()
  return "".join(_ for _ in up if _ in "ABCDEFHSTUVWYZ")
```

L'appel à la méthode d'instance est préfixée par le nom de la variable alors que celui de la méthode statique est préfixée par le nom de la classe (le type).

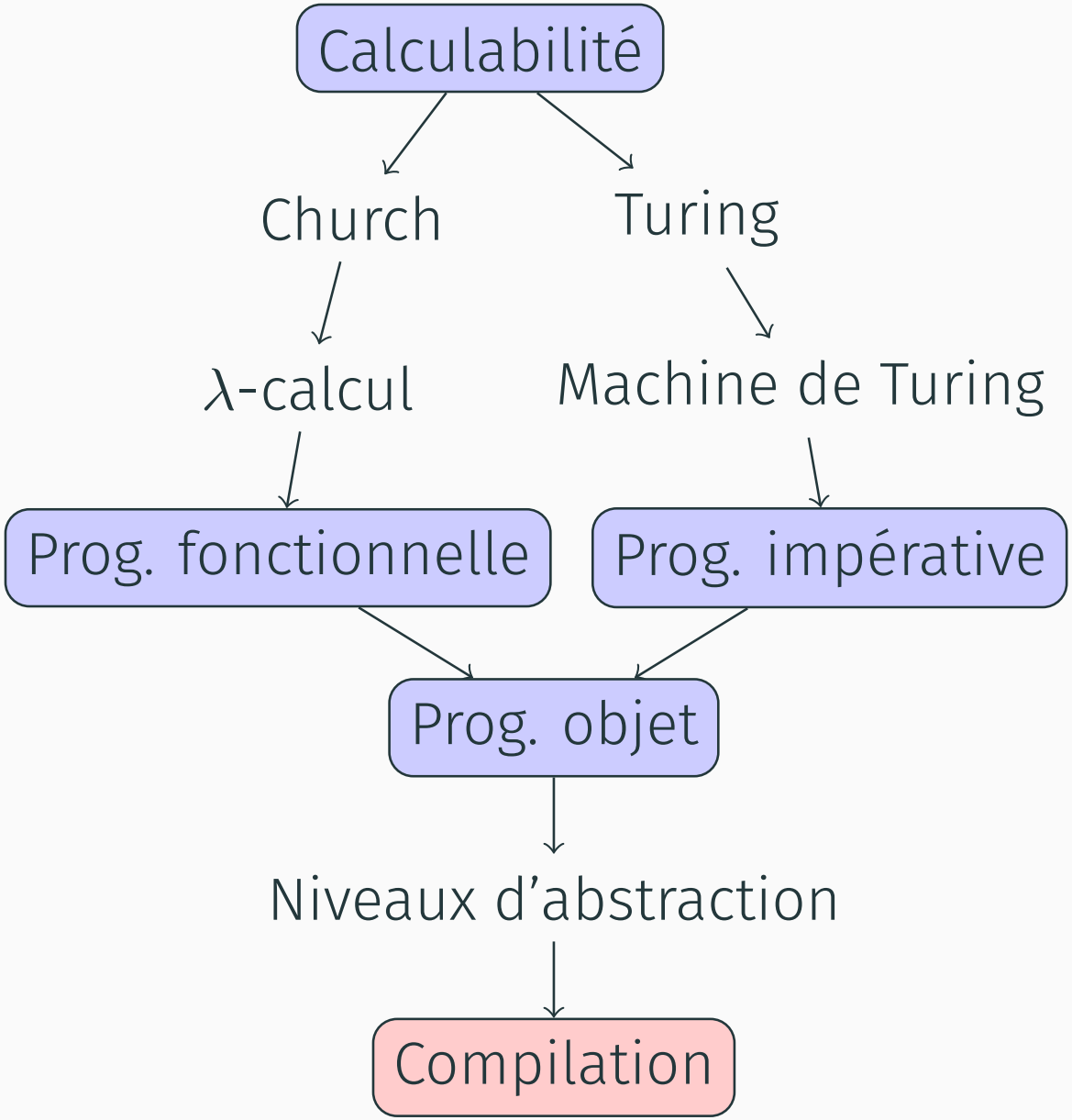
Programmation objet : Notions complémentaires

- Héritage simple et multiple
- Visibilité : concepts d'attributs privés ou publiques
- Classes abstraites, interfaces, etc.

L'implémentation des concepts de la programmation orientée objet varie en fonction des langages.

Quelques langages orientés objets : Smalltalk, Java, C++...

Plan


Calculabilité
Church
Turing
λ-calcul
Machine de Turing
Prog. fonctionnelle
Prog. impérative
Prog. objet
Niveaux d'abstraction
Compilation

Bas-niveau et haut-niveau

Les langages que nous avons vus permettent d'exprimer des programmes compréhensibles par l'homme...mais

Bas-Niveau

Les machines exécutent des instructions écrites en **langage machine**, c'est-à-dire en binaire.

La signification de celles-ci dépend du processeur de la machine, on parle de jeu d'instructions.

Exemple

Sur une machine équipée d'un processeur x86-64 **10000011110000000000001110** en binaire ou 83C00E en hexadécimal : ajoute 14 au registre EAX du processeur.

- manière la plus directe d'exécuter des instructions sur une machine
- requiert une connaissance approfondie des détails de bas niveau de la machine
- très fastidieux à écrire

Bas-Niveau

Depuis les années 50, les programmeurs ont utilisé un premier langage de plus haut-niveau : **le langage assembleur**. Il a déjà le mérite d'être lisible par un humain

Exemple

Sur une machine équipée d'un processeur x86-64
ADD EAX 14 : ajoute 14 au registre EAX du processeur.

Il y a un programme nommé **assembleur** qui traduit ce langage machine.

Problème :

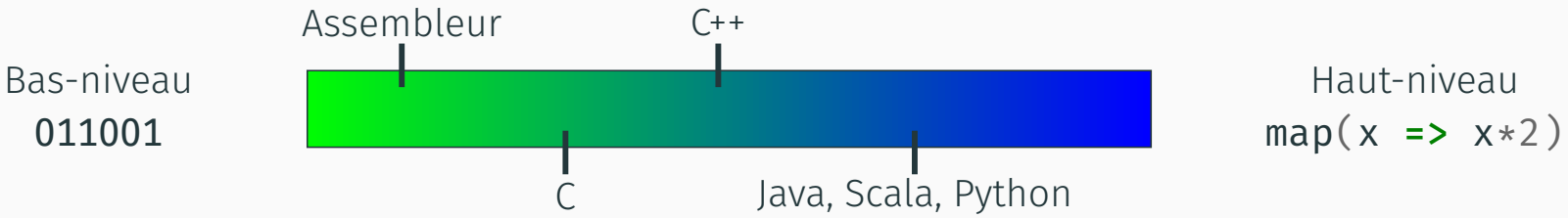
- C'est encore assez proche des détails de bas-niveau.
- C'est également relativement fastidieux à utiliser.
- Ça n'est pas *portable* : dépend de la machine utilisée.

Besoin d’abstraction

Avec la montée en flèche de la complexité et de la taille des programmes, il a fallu écrire des langages de plus haut-niveau pour s’abstraire des détails de la machine.

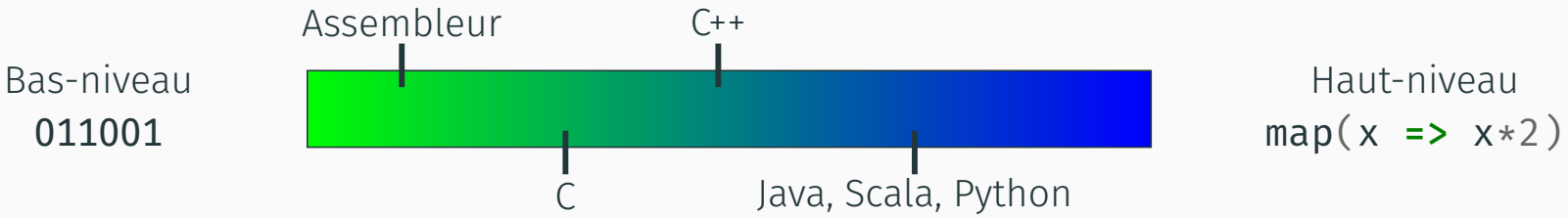
Bas-niveau et haut-niveau

On peut classer les langages par chronologie, paradigmes et **niveau d'abstraction** :



Bas-niveau et haut-niveau

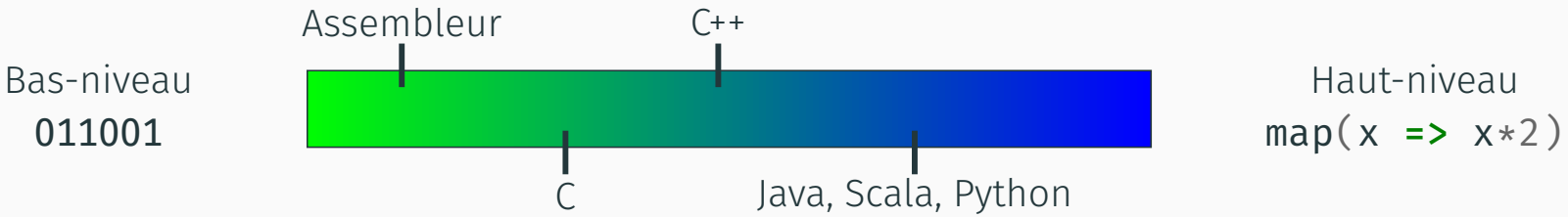
On peut classer les langages par chronologie, paradigmes et **niveau d'abstraction** :



Le travail d'écriture d'un programme est grandement facilité mais la machine exécute toujours du code machine.

Bas-niveau et haut-niveau

On peut classer les langages par chronologie, paradigmes et **niveau d'abstraction** :

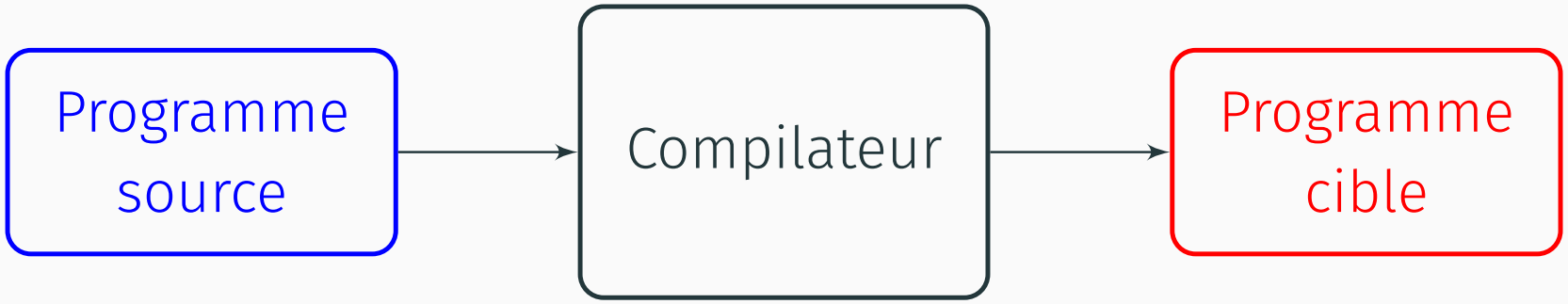


Le travail d'écriture d'un programme est grandement facilité mais la machine exécute toujours du code machine.

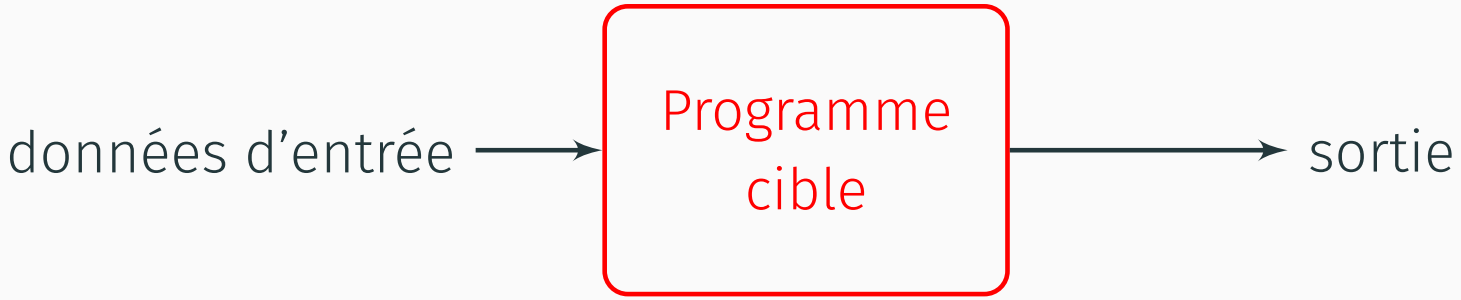
C'est le **compilateur** qui va réaliser cette tâche.

La compilation

La compilation est une **traduction**. Un compilateur est un programme qui prend un programme dans un langage **source** et produit un programme dans un langage **cible**.

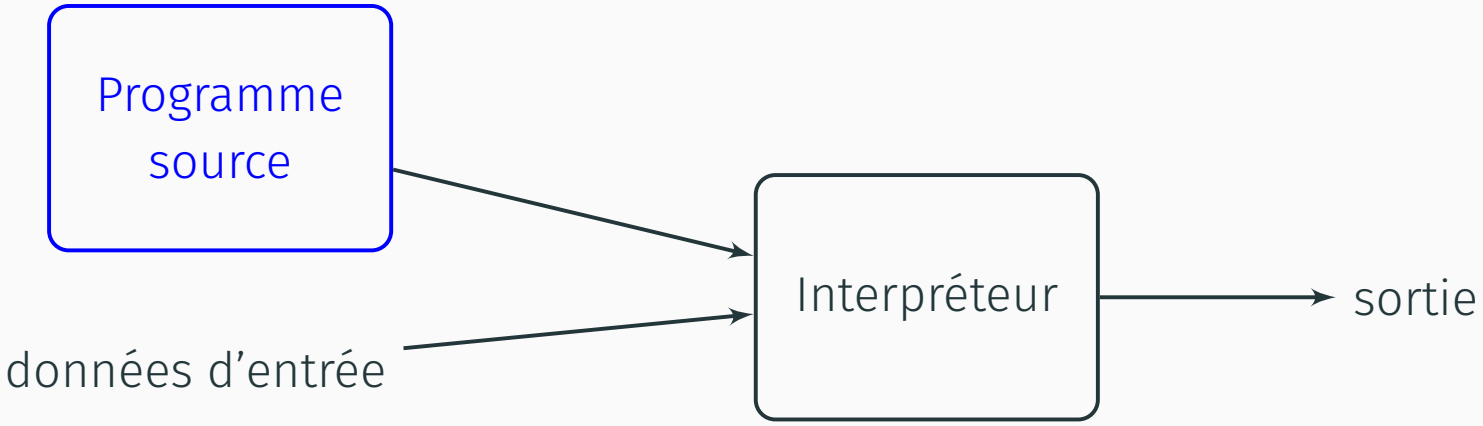


Une fois le programme compilé en langage (exécutable), on lui fournit des données en **entrée** et il produit un résultat en **sortie** :



Interpréteur

L'interpréteur est un programme qui exécute les opérations d'un programme **source** sur des données fournies en **entrée** et qui fournit un résultat en **sortie**.



- Le programme cible en langage machine obtenu par compilation est en général beaucoup plus rapide.
- L'exécution pas à pas de l'interprète permet un diagnostic de l'erreur plus précis.

Langages compilés ou interprétés?

La frontière entre les deux n'est pas toujours très nette.

- Un code source Python est traditionnellement interprété mais après une compilation vers un code intermédiaire : le *bytecode*
- Un code source Java est compilé vers du *bytecode* puis exécuté par une machine virtuelle.
- Les codes sources des langages C, C++ sont compilés.

Il faut distinguer le langage et son implémentation standard $\Rightarrow$ on pourrait écrire un compilateur pour du code Python.

Compilation

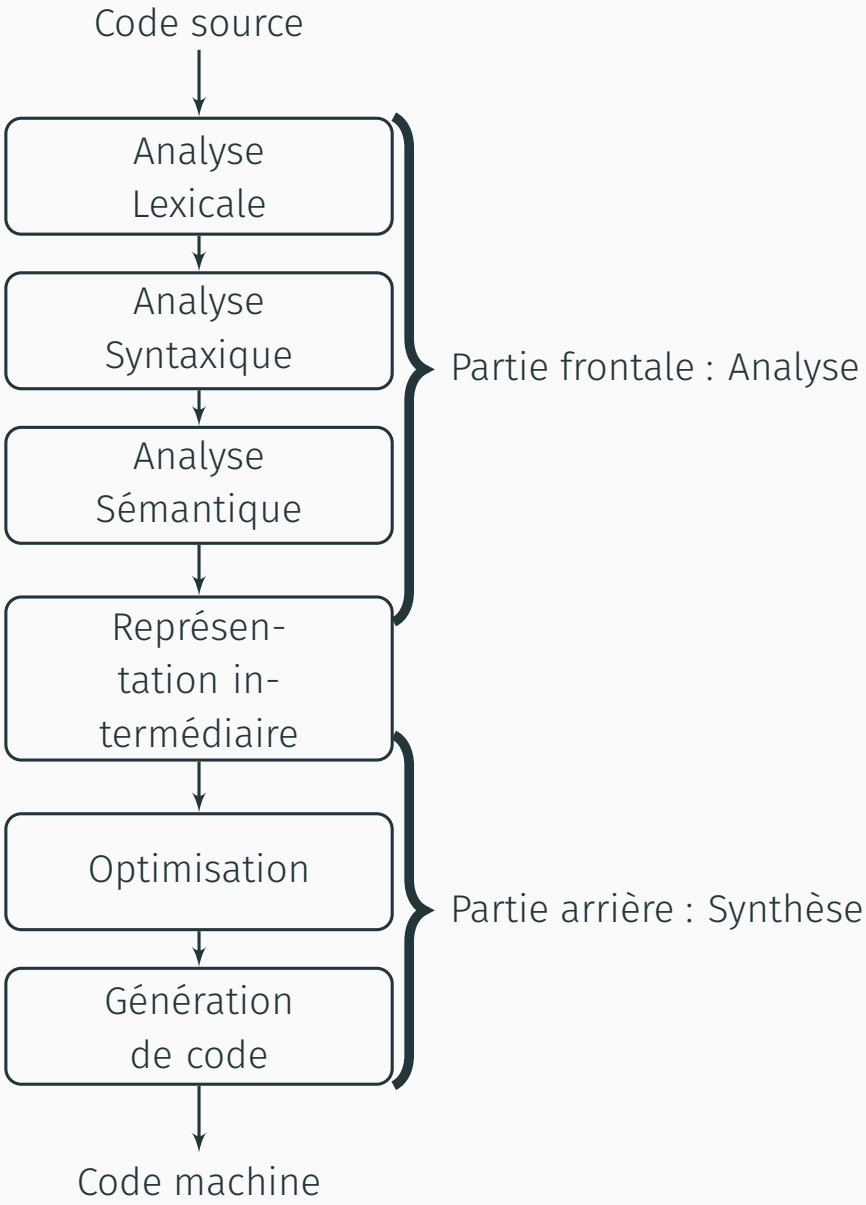
Challenges de la compilation :

- Produire un exécutable au comportement similaire au code source
- Produire un exécutable efficace

La compilation est décomposée en une série de phases modulables regroupées en deux parties principales :

1. Une partie analyse (dite frontale, front-end) : de l'analyse du code source à une représentation intermédiaire (indépendant de l'architecture).
2. Une partie synthèse (back-end) : de la représentation intermédiaire à la production du code machine.

Compilation

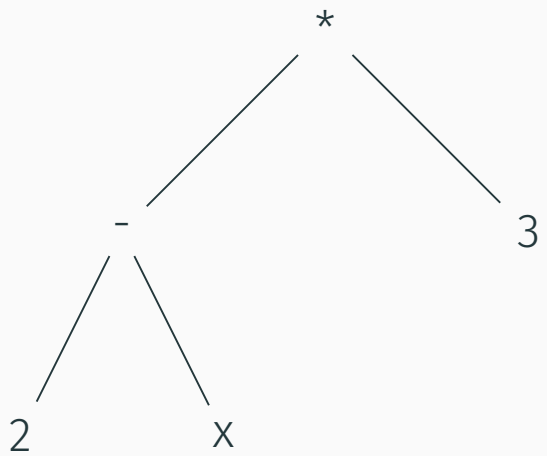


La compilation : front-end

L'analyse lexicale est la première étape de la compilation :

- Elle collabore avec l'analyse syntaxique pour passer d'une syntaxe concrète à une syntaxe abstraite
- Elle transforme le programme source, *une suite de caractères*, en une suite de mots, dit aussi lexèmes (tokens)
- Elle utilise pour ce faire le formalisme des **expressions régulières** (ou rationnelles)

Les expressions `( 2 - x ) * 3` et `(2-x)*(3)` ont la même syntaxe abstraite :



Expressions régulières

Définition

Une expression régulière est une chaîne de caractères qui décrit un motif.

Quelques exemples :

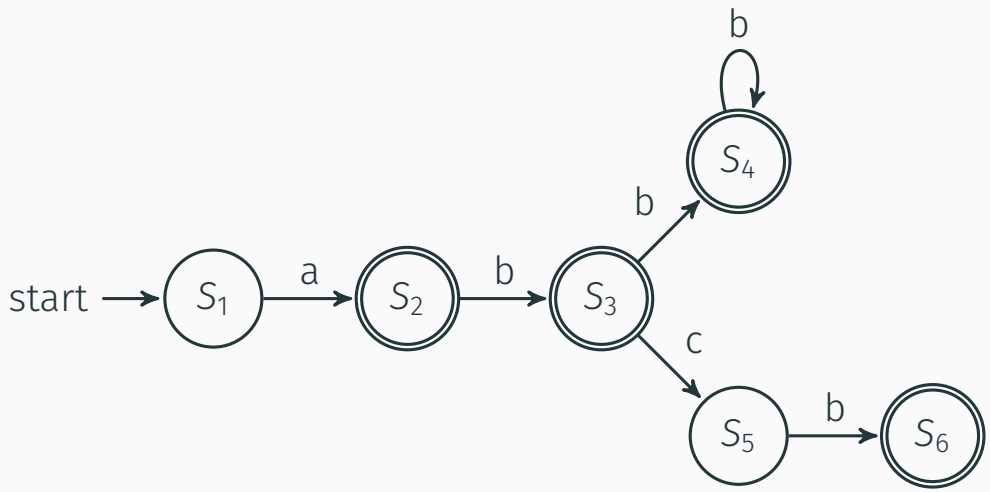
- `[0-9]+,[0-9]+` filtre un nombre flottant
- `[A-Z]{2}-[0-9]{3}-[A-Z]{2}` filtre les plaques d'immatriculations françaises
- `[a-z0-9_-]{3,16}` filtre un nom d'utilisateur d'une certaine taille
- `[a-z0-9._-]+@[a-z0-9._-]+.[com|fr]+` filtre certaines adresses email

Expressions régulières: automates

Équivalence

On peut associer un automate à une expressions régulière. On dit alors que l'automate reconnaît le langage défini par l'expression régulière.

L'automate suivant reconnaît le langage définit par l'expression régulière `a(b*|bcb)`



- reconnus : a, ab, abcb, abbbb, ...
- non reconnus : ba, abccb, abbcb, ...

Expressions régulières

L'analyse lexicale va nous permettre d'identifier les différents types de lexèmes dans le texte :

- Les mots-clés du langage (return, for,...)
- Les noms de variables (toto45, cpt, bar, foo,...)
- Les entiers (42,...)
- Les symboles (,), [,], ., +, *, ...
- Les blancs : retour à la ligne, espace, tabulation...

Analyse syntaxique

La syntaxe d'un langage s'exprime à l'aide d'une grammaire.

Exemple

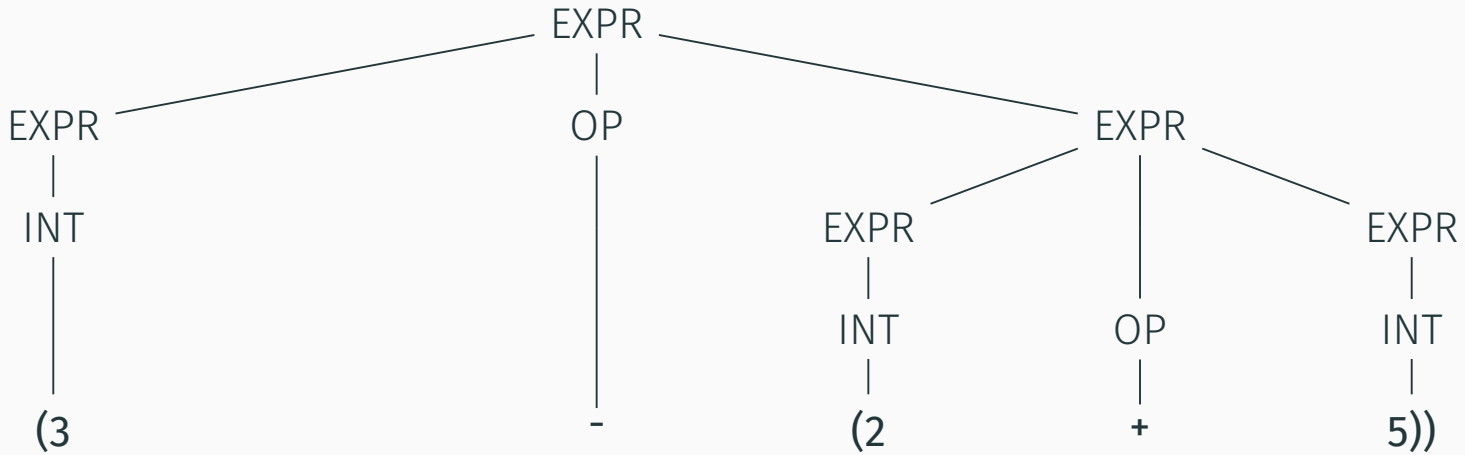
Le langage des expressions arithmétiques avec addition et soustraction uniquement sur les entiers à 1 digit :

$$\langle \text{EXPRESSION} \rangle \models \langle \text{ENTIER} \rangle \mid (\langle \text{EXPRESSION} \rangle \langle \text{OPERATEUR} \rangle \langle \text{EXPRESSION} \rangle)$$

$$\langle \text{ENTIER} \rangle \models 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\langle \text{OPERATEUR} \rangle \models + \mid -$$

Pour vérifier que le texte analysé appartient ou non au langage, on réalise une dérivation, représentable sous la forme d'un arbre. Exemple : (3 - (2 + 5))



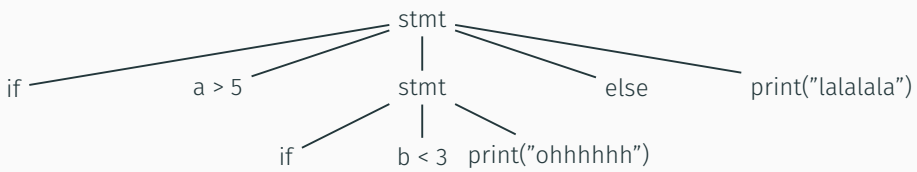
Ambiguïté : Dangling else

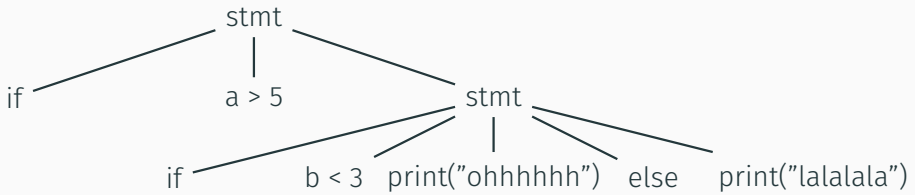
Une grammaire est **ambiguë** si elle peut être interprétée de plusieurs manières différentes (différents arbres de dérivation).

$$\langle \text{stmt} \rangle \quad \models \quad \textit{if}\ \langle \text{stmt} \rangle\ \textit{else}\ \langle \text{stmt} \rangle \ \mid\ \textit{if}\ \langle \text{stmt} \rangle \ \mid\ \ldots$$

```
if (a > 5)
  if( b < 3)
    println("ohhhhhh")
  else println("lalalala")
```

```
if (a > 5)
  if( b < 3)
    println("ohhhhhh")
else println("lalalala")
```





Analyse sémantique

- Après les deux premières étapes, on peut vérifier que le texte est lexicalement et syntaxiquement valide. Mais a-t-il du sens?

```
a = 5.96
b = "coucou"
c = a / b
```

La **vérification de type** est l'étape majeure de l'analyse sémantique.

Les types : Qu'est-ce que c'est?

Système de type

Un **type** est une caractérisation des données, il restreint les valeurs que peuvent prendre ces données et les opérations que l'on peut effectuer dessus.

Quelques exemples :

- le type **booléen** : {True,False} et les opérateurs (and,not,or,etc.) : True and False, True or False
- le type **entier** : les entiers et les opérations arithmétiques
- le type **Animal** : un nom, une race et les comportements (méthodes dort(), mange(), etc.)

Les types : A quoi ça sert?

- le typage permet de circonscrire les valeurs des expressions
- il permet de spécifier un comportement : int $\Rightarrow$ bool
- globalement, la vérification de type consiste à vérifier que les types attribués sont cohérents avec les données manipulées et que les opérations effectuées dessus sont correctes.

Par exemple, la déclaration d'une variable en Scala :

```
var s : Int = "Robert" /* error: type mismatch;
found   : String("robert")  required: Int */
var l = "Marie" // inférence de type
```

ou la concaténation de listes en Python :

```
lst = [1,2,3]
lst = lst + [42] #lst = [1,2,3] + [42]
lst = lst  +  42 #TypeError: can only concatenate
#list (not "int") to list
```

Les types

Il existe des typages statiques ou dynamiques :

- Le typage statique consiste à effectuer les vérifications de types à la *compilation* (C, Java, Scala, Ocaml).
- Le typage dynamique consiste à effecteur ces vérifications à l'*exécution* (Python).

Exemple Scala

```scala
val l1 = Seq(1,3,4)
val l2 = Seq(4,5,6)

def combineAdd(a:Seq[Int],b:Seq[Int],z:Int):Seq[Int] = {
  val x = a.map(t => t + z)
  val y = b.map(t => t + z)
  return Seq(x,y)
}
```

Ce code Scala ne passe pas l'étape de compilation.

Les types : A quoi ça sert?

Le typage statique permet de détecter avant l'exécution de nombreuses erreurs, par ex:

- addition d'une chaîne de caractères et d'un entier
- passer une liste de liste au lieu d'une liste en paramètre d'une fonction

Aussi,

- le type donne de l'information sur le programme
- il facilite le travail du compilateur qui travaille sur une copie propre

Robin Milner

"Well-typed programs can't go wrong."

On parle parfois de *typage fort et faible* mais ces notions ne sont pas bien définies (conversion implicite?).

Inférence de type

En fonction des langages, les types peuvent être :

- Déclarés
  - Le programmeur spécifie explicitement le type des variables, des fonctions...
  - Considéré comme une pratique lourde ou rassurante selon les programmeurs!
  - Exemples : C, Java
- Inférés
  - Le compilateur ou l'interpréteur va "deviner" de quel type il s'agit.
  - Exemples : Python, OCaml

On peut aussi avoir un mélange des deux comme en Scala.

La compilation : back-end

- La phase d'optimisation : l'objectif est généralement de produire un code plus efficace
- Génération de code : allocation mémoire, sélection d'instructions machine

Un exemple complet de compilation

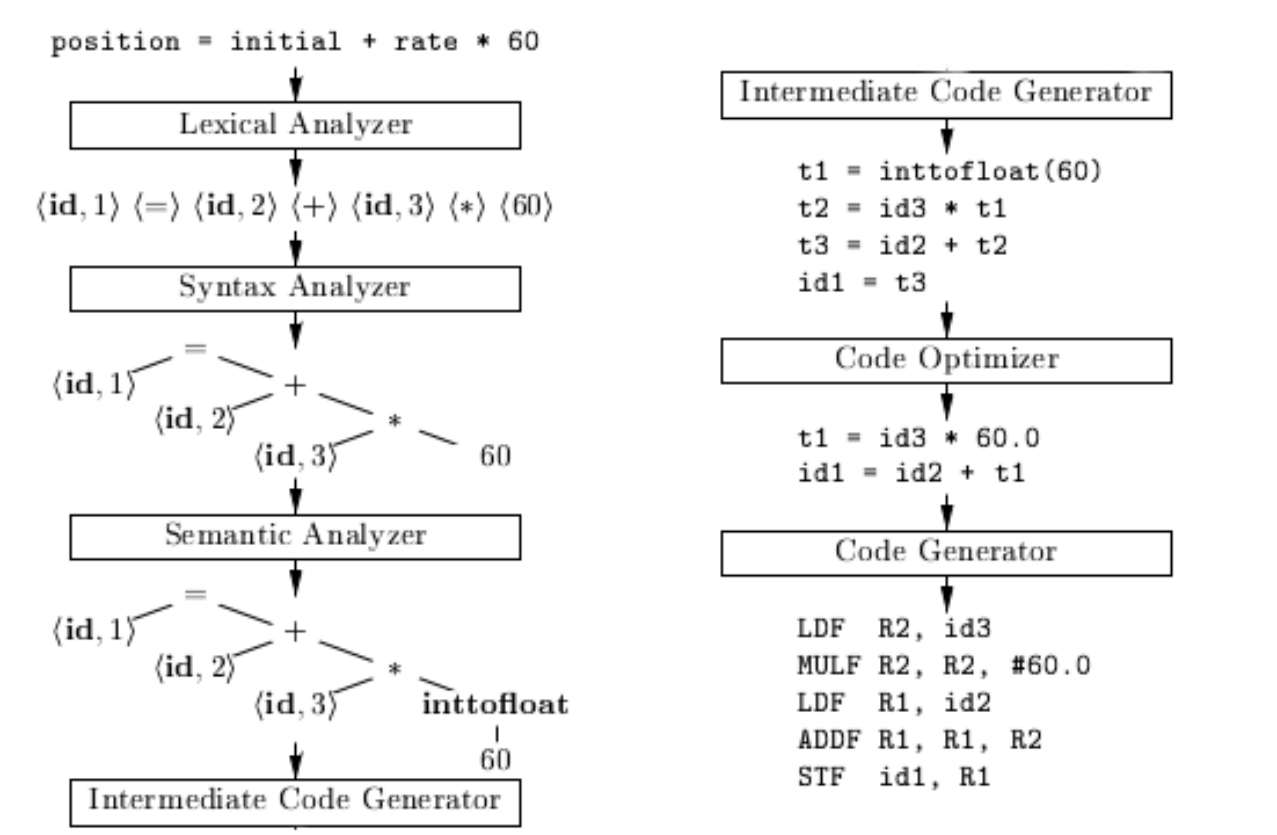


Figure 3: Repris de [4]

Conclusion

- Beaucoup de langages mais beaucoup de concepts en commun, évolutions à la marge. Il y a des mélanges harmonieux, cela n'est pas cloisonné.
- Le futur : la programmation concurrente, s'adapter au parallélisme inhérent aux architectures modernes.

Références i


Gilles Dowek. *Les principes des langages de programmation*. Editions Ecole Polytechnique, 2008.


Gilles Dowek and Jean-Jacques Lévy. *Introduction à la théorie des langages de programmation*. Editions Ecole Polytechnique, 2006.


Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.


Jeffrey Ullman, Alfred V Aho, and Ravi Sethi. *Compilers: Principles, Techniques and Tools, 2nd edition*. 2003.