# Conception d'un front-end Python pour *BOLDR*

Stage de fin de License – Juin-Juillet 2017
LRI d'Orsay, sous la supervision de Kim Nguyen

Romain Liautaud

*École Normale Supérieure de Lyon*

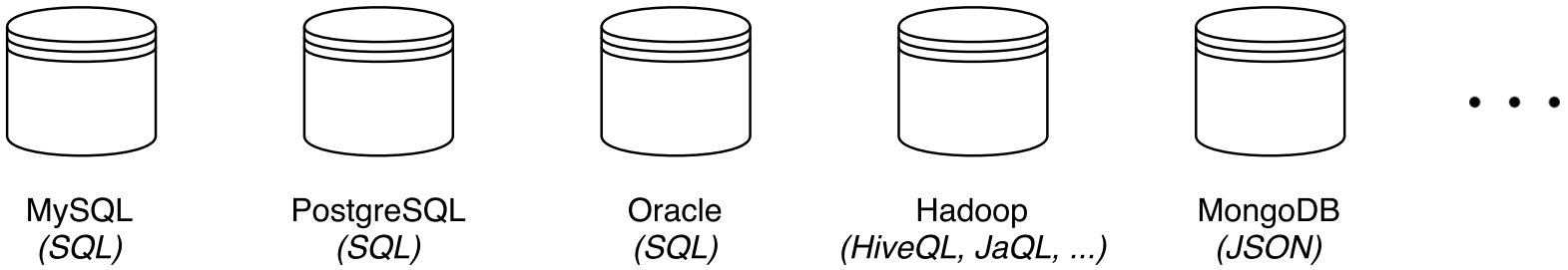
# Plan de l'exposé


1. Introduction
2. Choix de la syntaxe
3. Choix d'un mécanisme d'introspection
4. Du bytecode au QIR
   - Construction du graphe de flot de contrôle
   - Exécution de la machine à pile symbolique
   - Transformation des boucles `while`
5. Conclusion

# Un problème d'impédance

Il y a une séparation entre langages de programmation généralistes *(Python, R, ...)* et langages de requête *(SQL, HiveQL, ...)*.

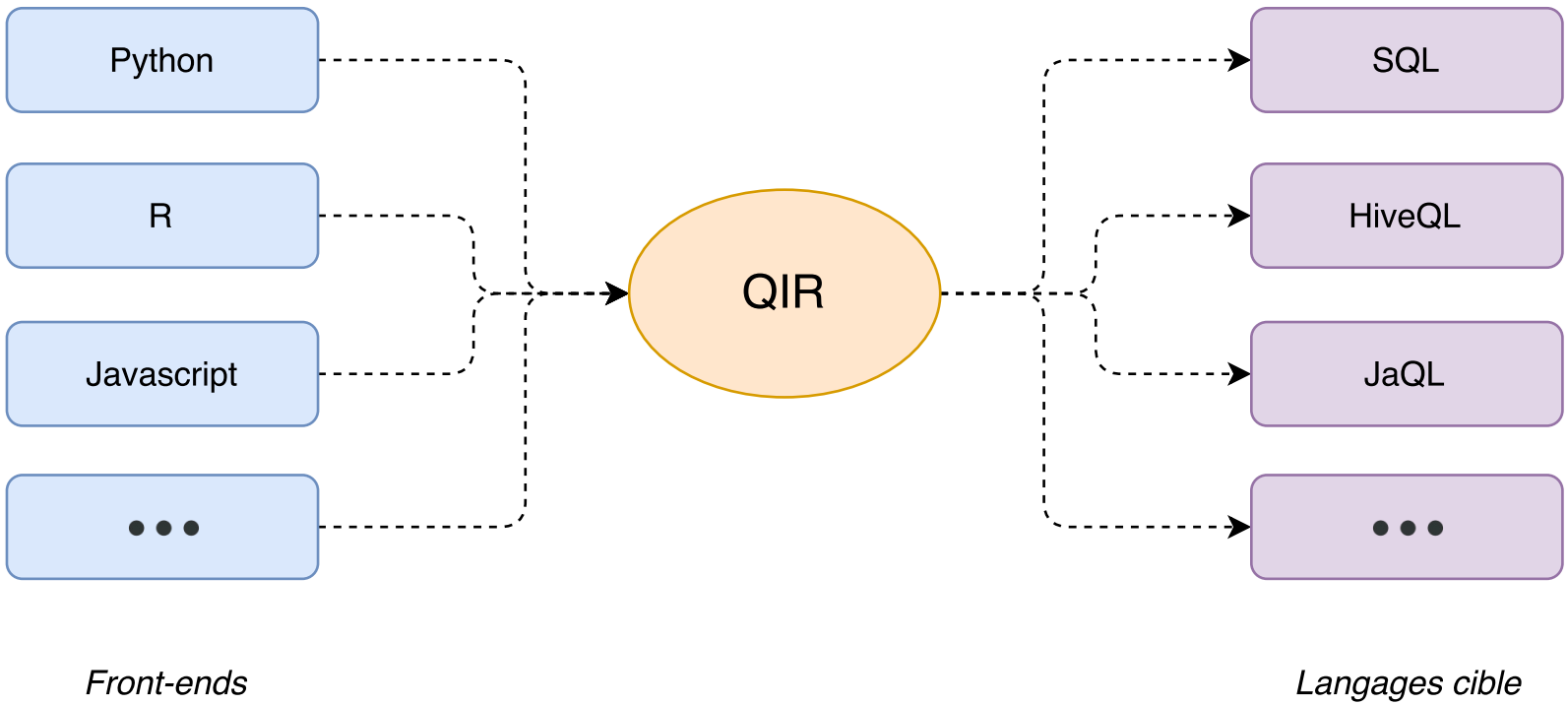
- Deux langages (au moins) à connaître.




- Besoin de réécrire les requêtes si l'on change de SGBD.
- Impossibilité de réutiliser la logique métier dans les requêtes.

# Le projet *BOLDR*

*BOLDR*, un framework de *language-integrated querying* développé par l'équipe VALS du LRI d'Orsay.



# La représentation intermédiaire

Le *QIR*, un lambda-calcul augmenté de :

- Représentation des types scalaires *(entiers, chaînes, ...)* ;
- Représentation des listes ;
- Représentation des tuples nommés ;
- Opérateurs de l'algèbre relationnelle.

# La représentation intermédiaire

$$
\begin{array}{rcl}
\textit{expr} & ::= & \textit{const} \mid \textit{id} \\
& \mid & \textit{func} \mid \textit{list} \\
& \mid & \textit{tuple} \mid \textit{op} \\
& \mid & \textit{alg} \mid \textit{special} \\
\\
\textit{const} & ::= & \mathsf{Null}() \\
& \mid & \mathsf{Number}(\texttt{int}) \\
& \mid & \mathsf{Double}(\texttt{float}) \\
& \mid & \mathsf{String}(\texttt{str}) \\
& \mid & \mathsf{Boolean}(\texttt{bool}) \\
\\
\textit{id} & ::= & \mathsf{Identifier}(\texttt{str}) \\
\\
\textit{func} & ::= & \mathsf{Lambda}(\textit{id}, \textit{expr}) \\
& \mid & \mathsf{Application}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Conditional}(\textit{expr}, \textit{expr}, \textit{expr}) \\
\\
\textit{list} & ::= & \mathsf{ListNil}() \\
& \mid & \mathsf{ListCons}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{ListDestr}(\textit{expr}, \textit{expr}, \textit{expr}) \\
\\
\textit{tuple} & ::= & \mathsf{TupleNil}() \\
& \mid & \mathsf{TupleCons}(\textit{expr}, \textit{expr}, \textit{expr}) \\
& \mid & \mathsf{TupleDestr}(\textit{expr}, \textit{expr})
\end{array}
$$

$$
\begin{array}{rcl}
\textit{op} & ::= & \mathsf{Scan}(\textit{expr}) \\
& \mid & \mathsf{Filter}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Sort}(\textit{expr}, \textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Limit}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Group}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Join}(\textit{expr}, \textit{expr}, \textit{expr}) \\
\\
\textit{alg} & ::= & \mathsf{Not}(\textit{expr}) \\
& \mid & \mathsf{Div}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Minus}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Mod}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Plus}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Star}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Power}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{And}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Or}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{Equal}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{LowerOrEqual}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{LowerThan}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{GreaterOrEqual}(\textit{expr}, \textit{expr}) \\
& \mid & \mathsf{GreaterThan}(\textit{expr}, \textit{expr}) \\
\\
\textit{special} & ::= & \mathsf{Builtin}(\texttt{str}) \\
& \mid & \mathsf{Bytecode}(\texttt{bytes}) \\
& \mid & \mathsf{Database}(\cdots) \\
& \mid & \mathsf{Table}(\cdots)
\end{array}
$$

# Un exemple du fonctionnement de *BOLDR*

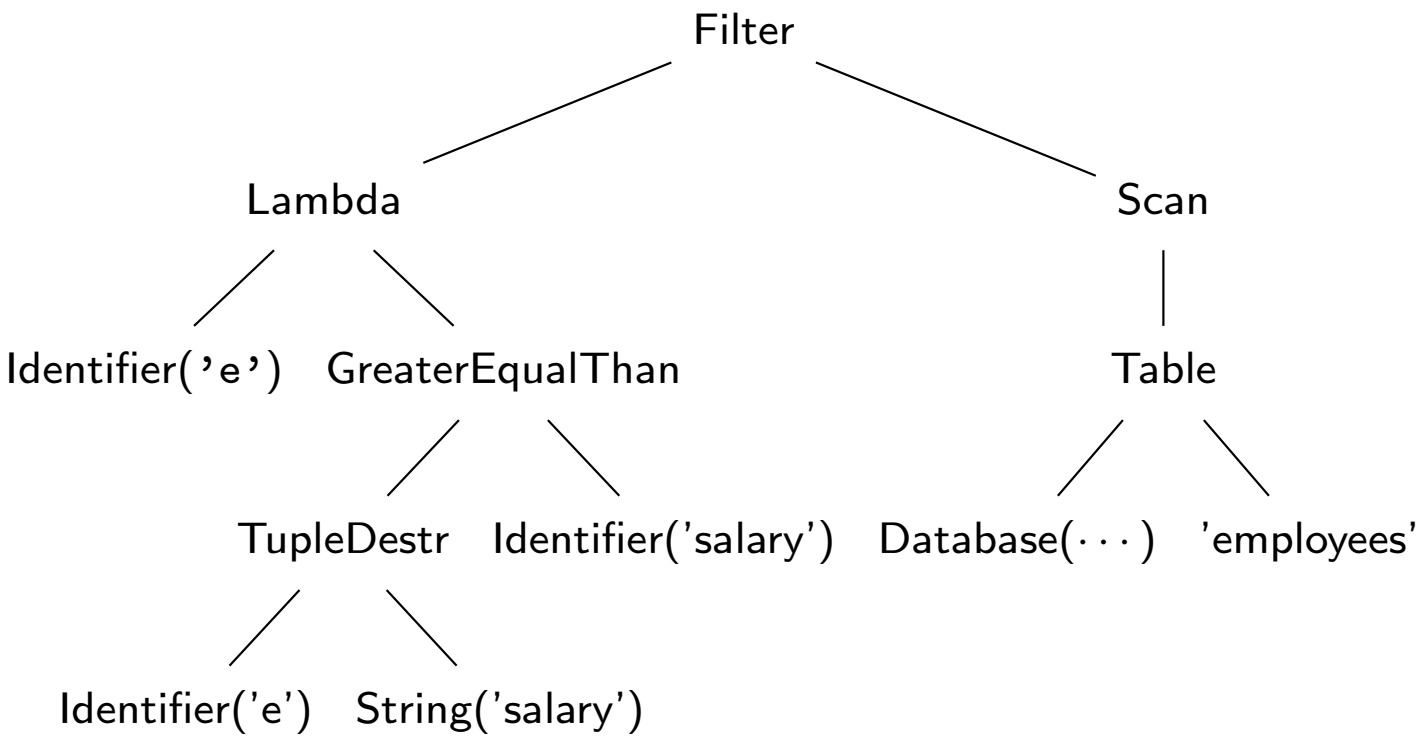
```
def at_least(salary):
    return (e for e in employees if e.salary >= salary)
```




```
SELECT * FROM employees AS e WHERE e.salary >= 2000
```

# Un exemple du fonctionnement de *BOLDR*

```
def at_least(salary):
    return (e for e in employees if e.salary >= salary)
```




```
SELECT * FROM employees AS e WHERE e.salary >= 2000
```

# Un exemple du fonctionnement de *BOLDR*

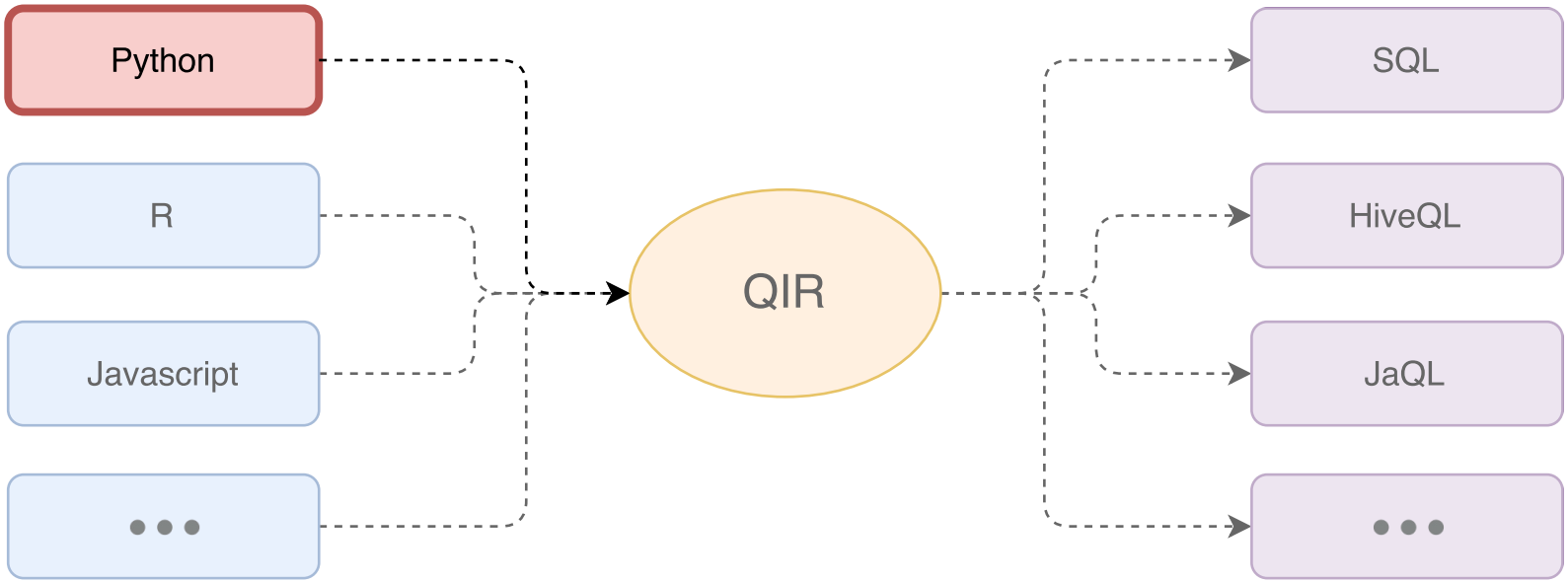
```
def at_least(salary):
    return (e for e in employees if e.salary >= salary)
```




```
SELECT * FROM employees AS e WHERE e.salary >= 2000
```

# À propos du stage

Mon stage : concevoir un front-end de BOLDR pour Python qui ne nécessite pas de modifier le compilateur ou l'interpréteur CPython *(comme le font les front-ends actuels)*.



# Choix de la syntaxe

```
1   from boldr import local, batch
2
3   db = boldr.db(
4       driver='postgresql',
5       name='app',
6       host='localhost',
7       port=3306)
8
9   query1 = db.table('users')
10              .filter(lambda u: u.verified)
11              .project(lambda u: {'score': u.age * u.points})
12              .filter(lambda u: u.score >= 100)
13
14  query2 = (max(u.score, 0) for u in query1 if u.score < 50)
15
16  print(local % query1)
17  print(batch % query2)
```

# Les mécanismes d'introspection en Python

**Introspection** : capacité d'un programme à examiner, et éventuellement à modifier, ses propres structures internes lors de son exécution.

1. Le module `inspect`.

```
def f(x):
    return x + 1

import inspect
inspect.getsource(f)
>>> 'def f(x):\n\treturn x + 1\n'
```

Permet de récupérer, lors de l'exécution, le code ayant servi à définir une fonction donnée. Ne fonctionne pas avec toutes les fonctions *(définies en ligne de commande, anonymes, ...)*.

# Les mécanismes d'introspection en Python

2. Le module `dis` *(pour disassembler)*.

```
def f(x):
    return x + 1

import dis
dis.dis(f)
>>>   0 LOAD_FAST                0 (x)
>>>   3 LOAD_CONST               1 (1)
>>>   6 BINARY_ADD
>>>   7 RETURN_VALUE
```

Permet de récupérer, lors de l'exécution, le bytecode généré pour une fonction donnée. *(Représentation interne des programmes dans la machine virtuelle de CPython)*.

# Le bytecode CPython

Un programme écrit en bytecode CPython est une suite d'instructions *(numéro de ligne, nom d'opération, argument optionnel)* qui agit sur une pile contenant des objets Python.

$$
\begin{aligned}
\mathcal{O}_{stack} &= \texttt{NOP, POP\_TOP, ROT\_TWO, ROT\_THREE, DUP\_TOP, CALL\_FUNCTION} \\
&\quad \texttt{LOAD\_ATTR, STORE\_ATTR, DELETE\_ATTR,} \\
&\quad \texttt{BINARY\_SUBSCR, STORE\_SUBSCR, DELETE\_SUBSCR} \\
\mathcal{O}_{env} &= \texttt{LOAD\_CONST, LOAD\_NAME, LOAD\_FAST, STORE\_FAST, DELETE\_FAST,} \\
&\quad \texttt{LOAD\_GLOBAL, STORE\_GLOBAL, DELETE\_GLOBAL,} \\
\mathcal{O}_{bool} &= \texttt{COMPARE\_OP} \\
\mathcal{O}_{unary} &= \texttt{UNARY\_POSITIVE, UNARY\_NEGATIVE, UNARY\_NOT, ...} \\
\mathcal{O}_{binary} &= \texttt{BINARY\_POWER, BINARY\_MULTIPLY, BINARY\_ADD, ...} \\
\mathcal{O}_{inplace} &= \texttt{INPLACE\_POWER, INPLACE\_MULTIPLY, INPLACE\_ADD, ...} \\
\mathcal{O}_{branch} &= \texttt{POP\_JUMP\_IF\_TRUE, POP\_JUMP\_IF\_FALSE,} \\
&\quad \texttt{JUMP\_IF\_TRUE\_OR\_POP, JUMP\_IF\_FALSE\_OR\_POP} \\
\mathcal{O}_{jump} &= \texttt{JUMP\_FORWARD, JUMP\_ABSOLUTE}
\end{aligned}
$$

$$\mathcal{O} = \mathcal{O}_{stack} \cup \mathcal{O}_{env} \cup \mathcal{O}_{bool} \cup \mathcal{O}_{unary} \cup \mathcal{O}_{binary} \cup \mathcal{O}_{inplace} \cup \mathcal{O}_{branch} \cup \mathcal{O}_{jump}$$

# Traduire du bytecode en terme du QIR

Pour transformer une requête écrite en Python en sa représentation intermédiaire, il nous faut traduire du bytecode en termes du QIR.

```
def at_least(salary):
    return employees.scan().filter(lambda e.salary >= salary)
```

```
 0 LOAD_FAST      0 (e)
 3 LOAD_ATTR      0 (salary)
 6 LOAD_GLOBAL    0 (salary)
 9 COMPARE_OP     5 (>=)
12 RETURN_VALUE
```

# Traduire du bytecode en terme du QIR

Pour transformer une requête écrite en Python en sa représentation intermédiaire, il nous faut traduire du bytecode en termes du QIR.

```
def at_least(salary):
    return employees.scan().filter(lambda e.salary >= salary)
```

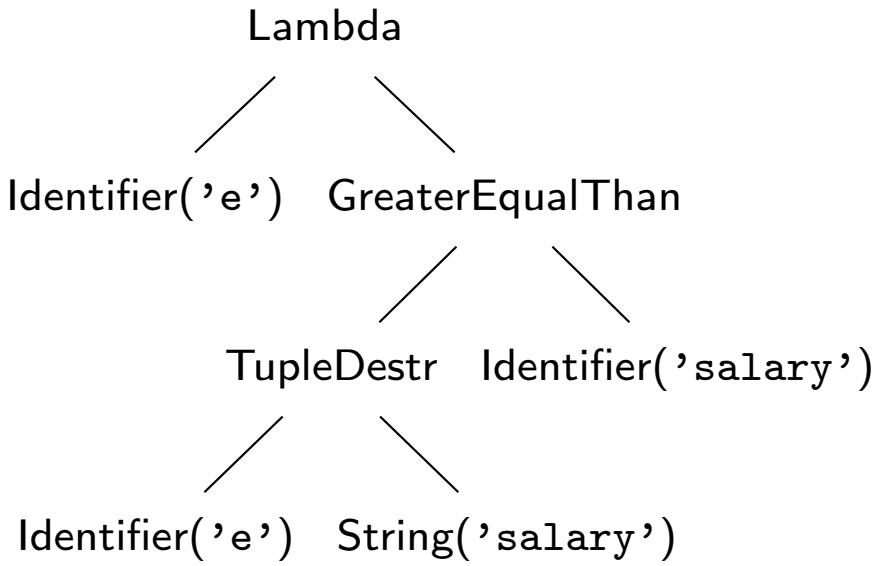
```
 0 LOAD_FAST      0 (e)
 3 LOAD_ATTR      0 (salary)
 6 LOAD_GLOBAL    0 (salary)
 9 COMPARE_OP     5 (>=)
12 RETURN_VALUE
```

# Traduire du bytecode en terme du QIR

Pour transformer une requête écrite en Python en sa représentation intermédiaire, il nous faut traduire du bytecode en termes du QIR.

```
def at_least(salary):
    return employees.scan().filter(lambda e.salary >= salary)
```

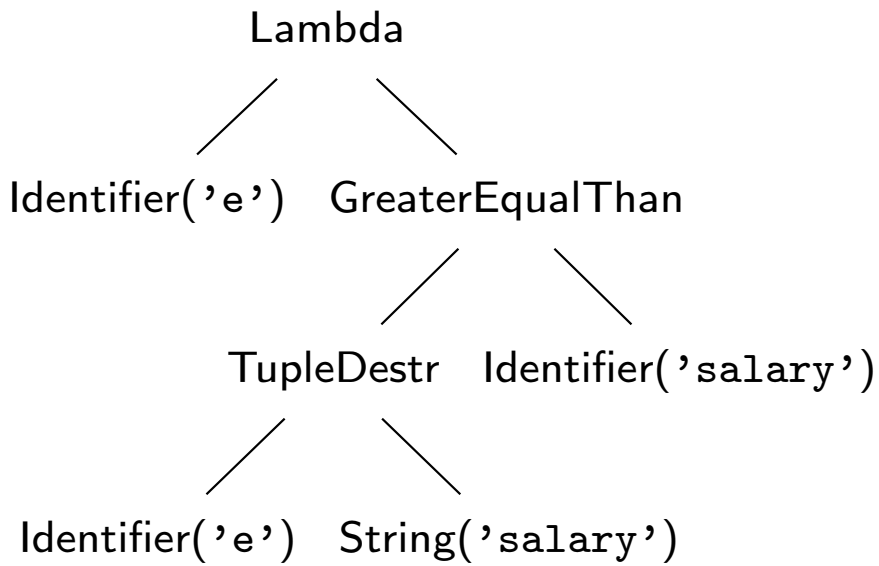
```
 0 LOAD_FAST      0 (e)
 3 LOAD_ATTR      0 (salary)
 6 LOAD_GLOBAL    0 (salary)
 9 COMPARE_OP     5 (>=)
12 RETURN_VALUE
```



# Principe de la transformation

Exécuter le bytecode, au moment de la traduction, sur une machine à pile **symbolique** qui manipule des termes du *QIR* au lieu d'objets Python.
*(Exemple d'exécution au tableau.)*

```
 0 LOAD_FAST      0 (e)
 3 LOAD_ATTR      0 (salary)
 6 LOAD_GLOBAL    0 (salary)
 9 COMPARE_OP     5 (>=)
12 RETURN_VALUE
```



# Le graphe de flot de contrôle

Le bytecode de certaines fonctions *(avec des conditions, boucles, ...)* contient des sauts *(conditionnels ou non)*.

```
def is_even(x):
    if x % 2 == 0:
        return True

    return False
```

```
 0 LOAD_FAST                0 (x)
 3 LOAD_CONST               1 (2)
 6 BINARY_MODULO
 7 LOAD_CONST               2 (0)
10 COMPARE_OP               2 (==)

13 POP_JUMP_IF_FALSE       20

16 LOAD_CONST               3 (True)
19 RETURN_VALUE

20 LOAD_CONST               4 (False)
23 RETURN_VALUE
```

# Le graphe de flot de contrôle

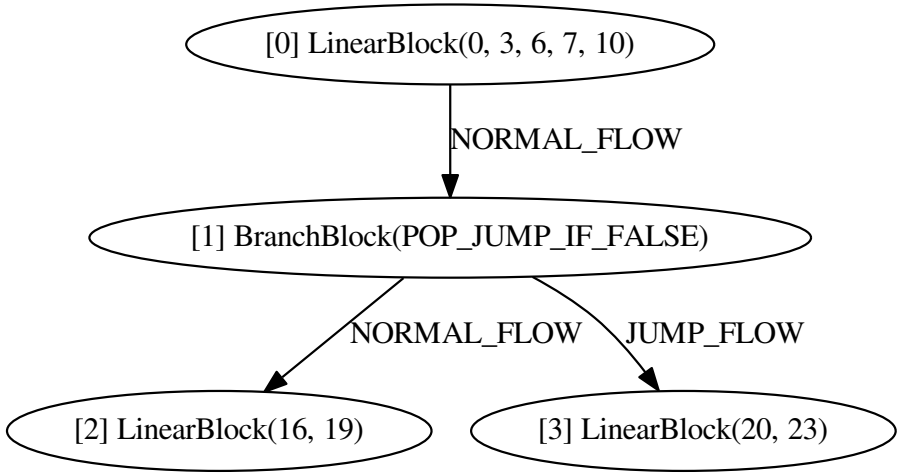
## Définition

Le **graphe de flot de contrôle** $(V_{\mathcal{P}}, E_{\mathcal{P}})$ d'un programme $\mathcal{P}$, avec $V_{\mathcal{P}}$ une partition des numéros de ligne et $E_{\mathcal{P}} \in {V_{\mathcal{P}}}^2$, est le graphe orienté :

- qui minimise $|V_{\mathcal{P}}|$ ;
- tel que pour tout $v \in V_{\mathcal{P}}$, les instructions correspondant à $v$ se suivent dans chaque chemin d'exécution possible de $\mathcal{P}$ ;
- tel que $(v, v') \in E_{\mathcal{P}}$ ssi. il est possible de sauter de la dernière instruction de $v$ à la première instruction de $v'$ dans $\mathcal{P}$.

# Exemple de construction du graphe

```
 0 LOAD_FAST                0 (x)
 3 LOAD_CONST               1 (2)
 6 BINARY_MODULO
 7 LOAD_CONST               2 (0)
10 COMPARE_OP               2 (==)

13 POP_JUMP_IF_FALSE       20

16 LOAD_CONST               3 (True)
19 RETURN_VALUE

20 LOAD_CONST               4 (False)
23 RETURN_VALUE
```




L'algorithme précis est fourni en Annexe E du rapport.

# La machine à pile symbolique

On adapte la sémantique de la machine à pile de CPython pour lui faire travailler symboliquement sur des termes du QIR.

## Définition

Un **état** $\langle n,\ S,\ B \rangle$ de la machine à pile symbolique est la donnée :

- du numéro de ligne $n$ de l'instruction courante ;
- d'une pile $S$ de termes du QIR ;
- d'une liste $B$ d'affectations de variables, i.e. de couples $(nom, valeur)$.

# Sémantique de la machine à pile symbolique

## Machine à pile CPython

$$
\begin{array}{rcll}
\sigma\langle n,\ S,\ E\rangle & = & \langle s(n),\ c\cdot S,\ E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_CONST},\ c) \\
\sigma\langle n,\ S,\ E\rangle & = & \langle s(n),\ E[id]\cdot S,\ E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_FAST},\ id) \\
\sigma\langle n,\ s\cdot S,\ E\rangle & = & \langle s(n),\ S,\ E[id\mapsto s]\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_FAST},\ id) \\
\sigma\langle n,\ s\cdot S,\ E\rangle & = & \langle s(n),\ s[id]\cdot S,\ E\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_ATTR},\ id) \\
\sigma\langle n,\ s_1\cdot s_2\cdot S,\ E\rangle & = & \langle s(n),\ s_1[id\mapsto s_2]\cdot S,\ E\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_ATTR},\ id)
\end{array}
$$

## Machine à pile symbolique

$$
\begin{array}{rcll}
\sigma'\langle n,\ S,\ B\rangle & = & \langle s(n),\ \mathsf{encode}(c)\cdot S,\ B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_CONST},\ c) \\
\sigma'\langle n,\ S,\ B\rangle & = & \langle s(n),\ \mathsf{Identifier}(id)\cdot S,\ B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_FAST},\ id) \\
\sigma\langle n,\ s\cdot S,\ B\rangle & = & \langle s(n),\ S,\ (id,s)\cdot B\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_FAST},\ id) \\
\sigma'\langle n,\ s\cdot S,\ B\rangle & = & \langle s(n),\ \mathsf{TupleDestr}(s,\ \mathsf{String}(id))\cdot S,\ B\rangle & \text{if } \mathcal{C}_n = (\texttt{LOAD\_ATTR},\ id) \\
\sigma'\langle n,\ s_1\cdot s_2\cdot S,\ B\rangle & = & \langle s(n),\ \mathsf{TupleCons}(\mathsf{String}(id),\ s_2,\ s_1)\cdot S,\ B\rangle & \text{if } \mathcal{C}_n = (\texttt{STORE\_ATTR},\ id)
\end{array}
$$

# Description de l'algorithme de transformation

1. Calcul d'un ordre topologique du graphe de flot de contrôle tel que le bloc contenant la première instruction ait l'indice 1 *(on suppose le graphe acyclique)*.
2. Première passe d'*exécution* : on exécute le code de chaque bloc, dans l'ordre topologique, sur la machine à pile symbolique. On démarre avec la pile finale des parents, et une liste d'assignations vides. On stocke la pile $S_f$ et la liste d'affectations $B_f$ en fin d'exécution.
3. Seconde passe d'*expression* : on assigne un terme du QIR à chaque bloc, dans l'ordre inverse topologique.

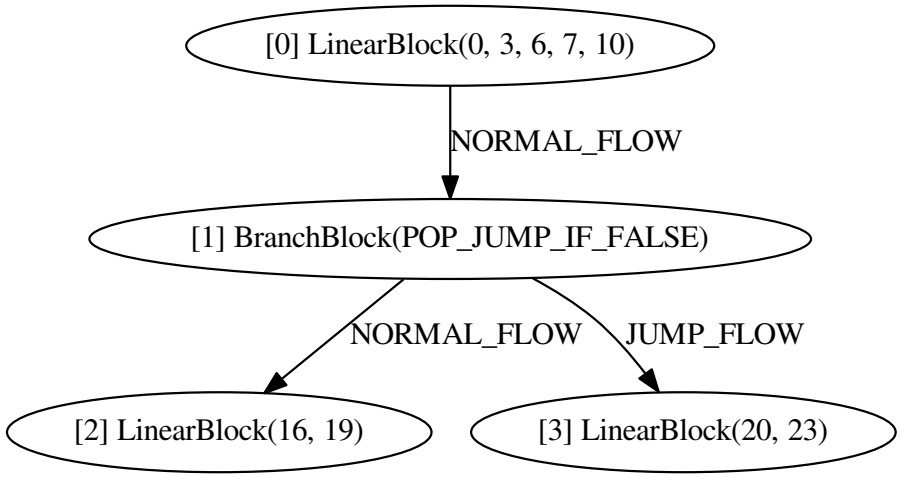
    - Feuille qui contient `RETURN_VALUE` : terme en haut de $S_f$ ;
    - Feuille sans `RETURN_VALUE` : Null() ;
    - Bloc de type `BRANCH` : Conditional(*cond*, *true*, *false*) ;
    - Sinon : terme assigné à l'unique successeur.

    Pour chaque (*nom*, *valeur*) dans la liste d'affectations $B_f$, on "entoure" le terme assigné dans Application(Lambda(Identifier(*nom*), ...), *valeur*).

L'algorithme formel est fourni en Annexe K du rapport.

# Démonstration de l'algorithme

```
 0 LOAD_FAST                0 (x)
 3 LOAD_CONST               1 (2)
 6 BINARY_MODULO
 7 LOAD_CONST               2 (0)
10 COMPARE_OP               2 (==)

13 POP_JUMP_IF_FALSE       20

16 LOAD_CONST               3 (True)
19 RETURN_VALUE

20 LOAD_CONST               4 (False)
23 RETURN_VALUE
```

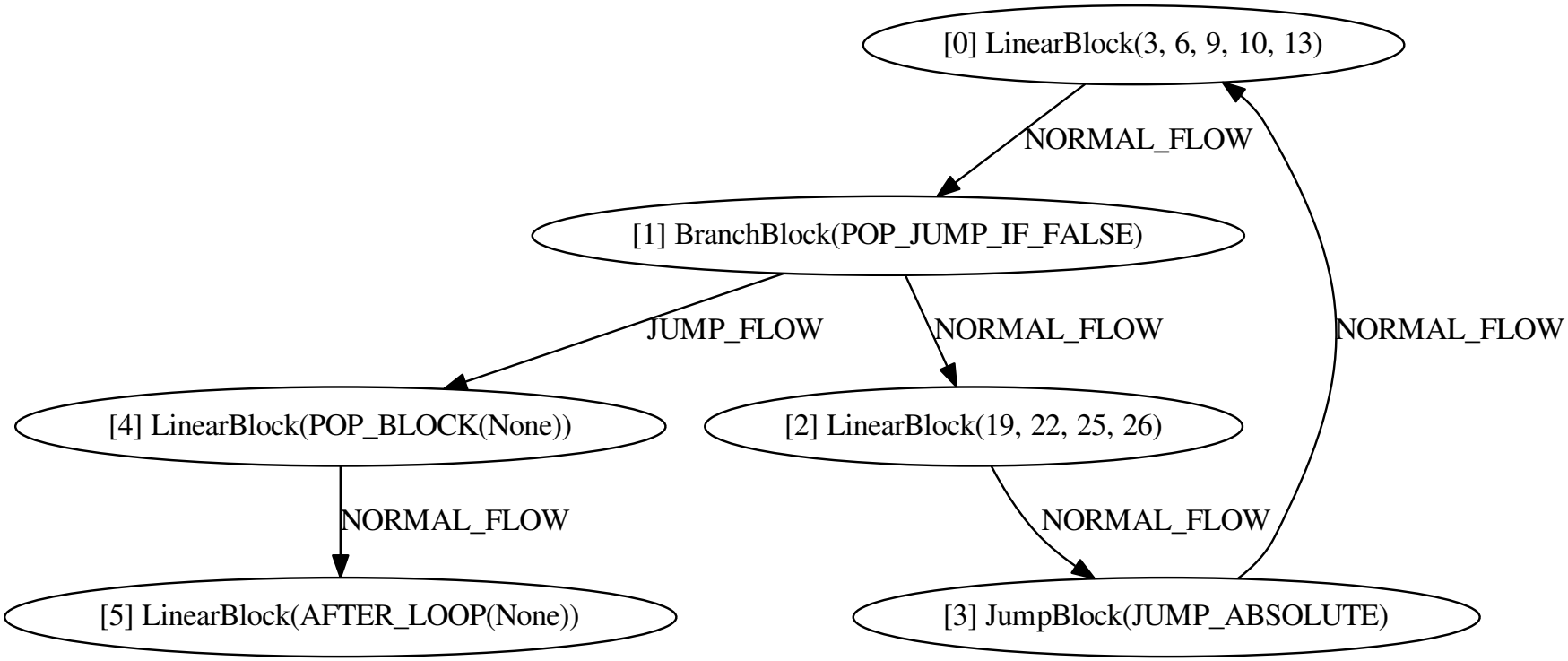


# Bytecode d'une fonction avec une boucle while

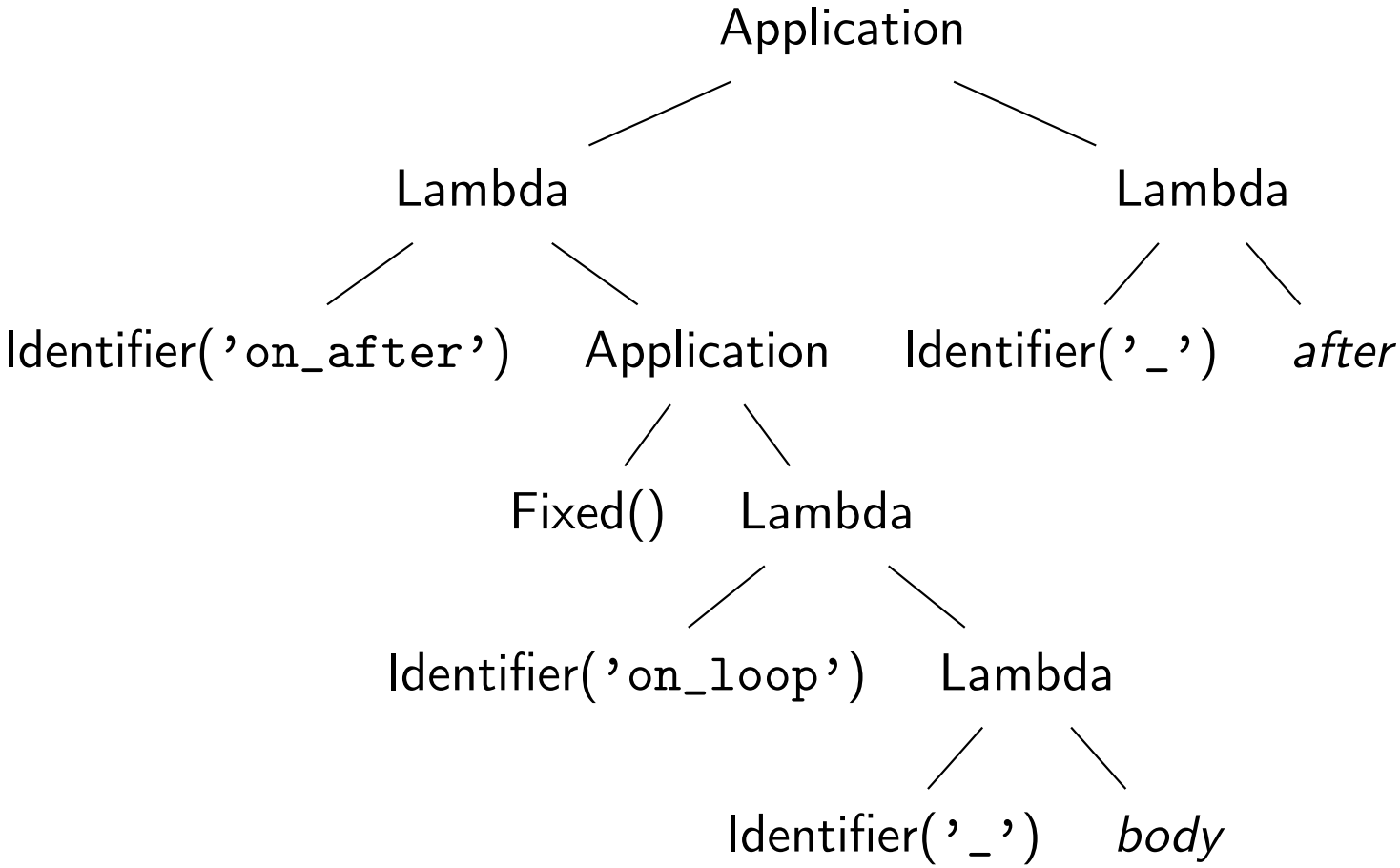
```
def closest_multiple(x, n):
    while x % n != 0:
            x = x + 1
    return x
```

```
 0 SETUP_LOOP              30 (to 33)
 3 LOAD_FAST                0 (x)
 6 LOAD_FAST                1 (n)
 9 BINARY_MODULO
10 LOAD_CONST               1 (0)
13 COMPARE_OP               3 (!=)
16 POP_JUMP_IF_FALSE       32

19 LOAD_FAST                0 (x)
22 LOAD_CONST               2 (1)
25 BINARY_ADD
26 STORE_FAST               0 (x)
29 JUMP_ABSOLUTE            3
32 POP_BLOCK

33 LOAD_FAST                0 (x)
36 RETURN_VALUE
```

# Graphe de l'intérieur d'une boucle `while`



# Graphe de l'intérieur d'une boucle `while`




Fixed() est un alias de l'opérateur de point-fixe $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

# Conclusion

Au cours du stage, extension de l'algorithme pour supporter :

- La transformation de fonctions contenant des boucles `for` ;
- La transformation de fonctions contenant des compréhensions de liste ou de générateur.

Implémentation du QIR et de l'algorithme en Python *(voir Annexe M)*, et de la communication avec le serveur QIR déjà développé par l'équipe.

Exploration d'évolutions possibles de BOLDR :

- Bonne gestion des effets de bord ;
- Système de type et meilleure prise en compte des fonctions supportées par chaque base de données.