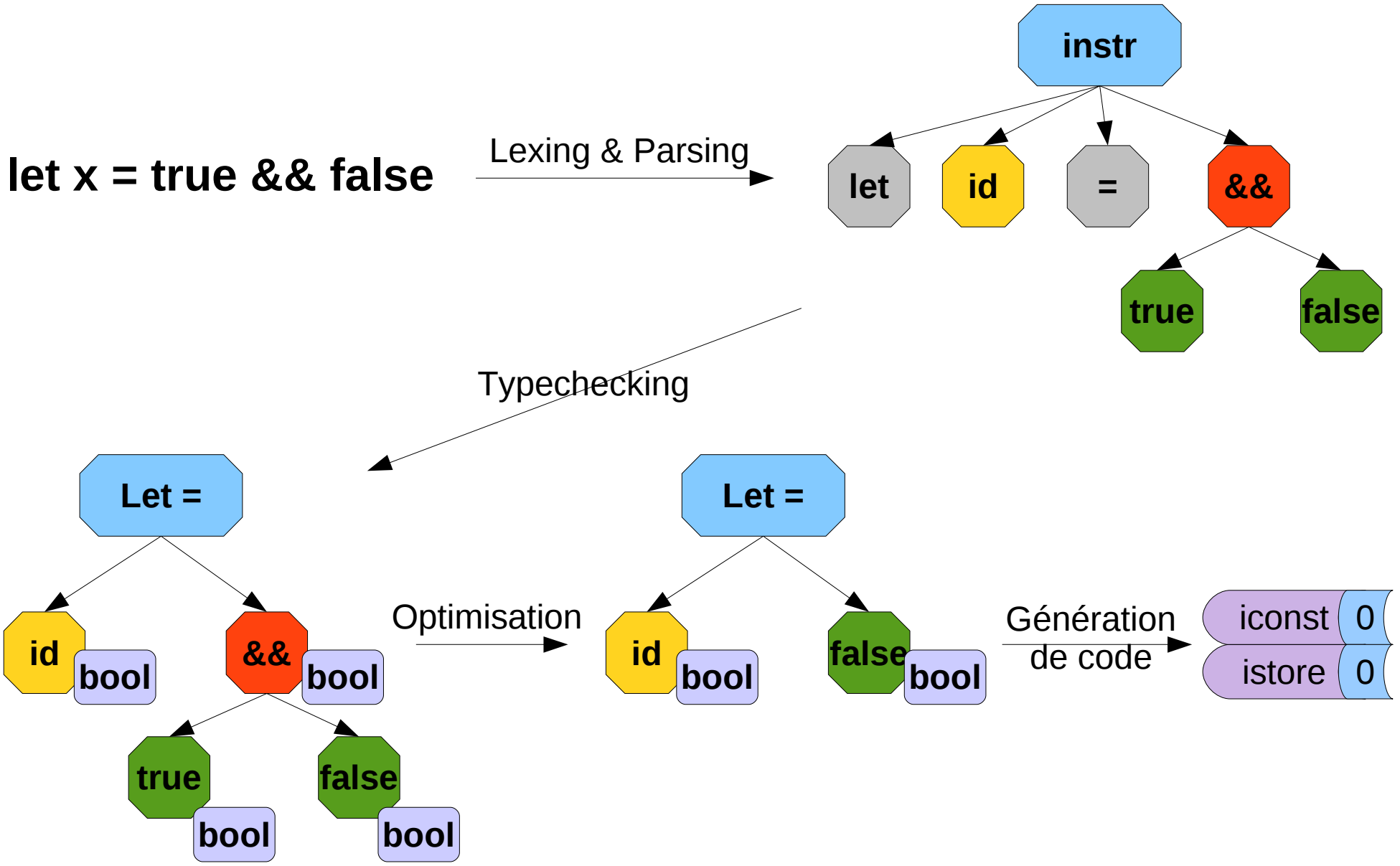
# Analyse Sémantique

Rémi Forax

# Plan

- Analyse Sémantique
- Construction de l'AST
- Design pattern visiteur
- Différentes passes
  - Enter
  - Typage
  - Control flow
  - Desugarisation

# Rappel


let x = true && false
Lexing & Parsing
instr
let
id
=
&&
true
false
Typechecking
Let =
id
bool
&&
bool
true
bool
false
bool
Optimisation
Let =
id
bool
false
bool
Génération
de code
iconst
0
istore
0

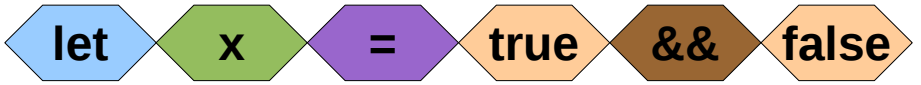
# Analyse en plusieurs passes

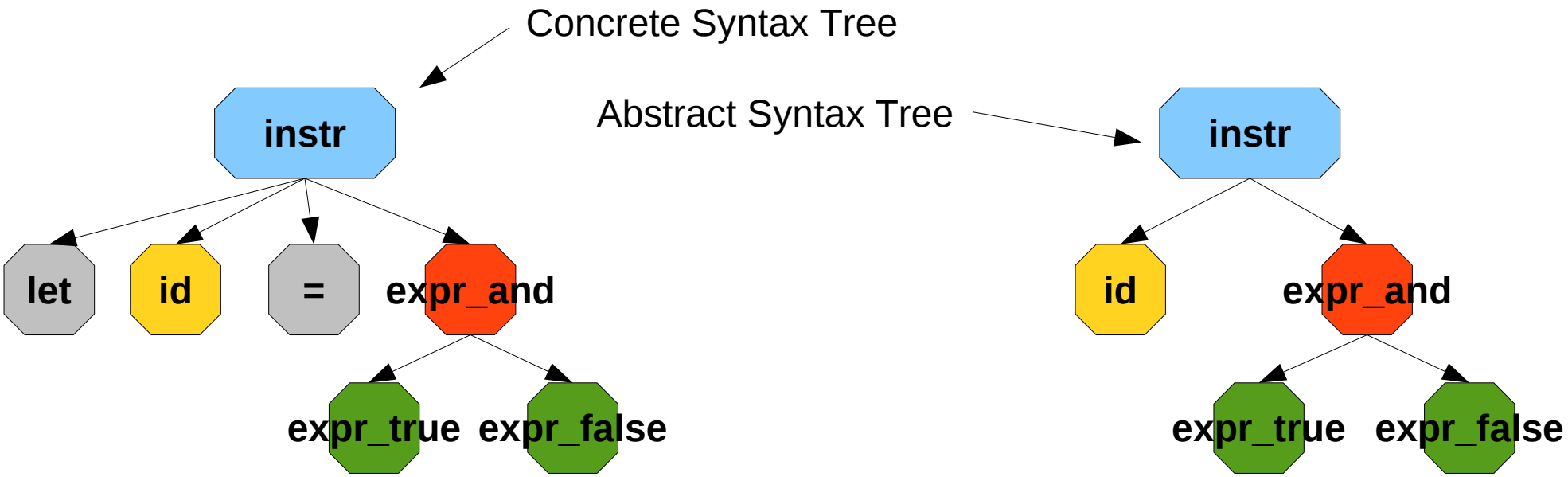
- En spécifiant
  - les terminaux/rules sous forme de regex
  - et les non-terinaux et productions sous forme de grammaire
- Tatoo génère un évaluateur

- Problème: on ne peut faire l'analyse qu'en une passe
- Solution: l'évaluateur génère un arbre sur lequel on fera plusieurs passes

# AST

- L'Abstract Syntax Tree correspond à l'arbre de dérivation auquel on aurait enlevés les terminaux n'ayant pas de valeur



# AST et Tatoo

- Tatoo est capable de générer automatiquement l'AST ainsi que le GrammarEvaluator permettant de créer l'AST

```
<target name="ebnf" depends="tasks">
  <delete dir="${gen-src}"/>
  <ebnf destination="${gen-src}" ebnfFile="${ebnf.file}"  parserType="lalr"
          generateast="true">
    <package lexer="${lexer.package}"/>
    <package parser="${parser.package}"/>
    <package tools="${tools.package}"/>
    <package ast="${ast.package}"/>
  </ebnf>
 </target>
```

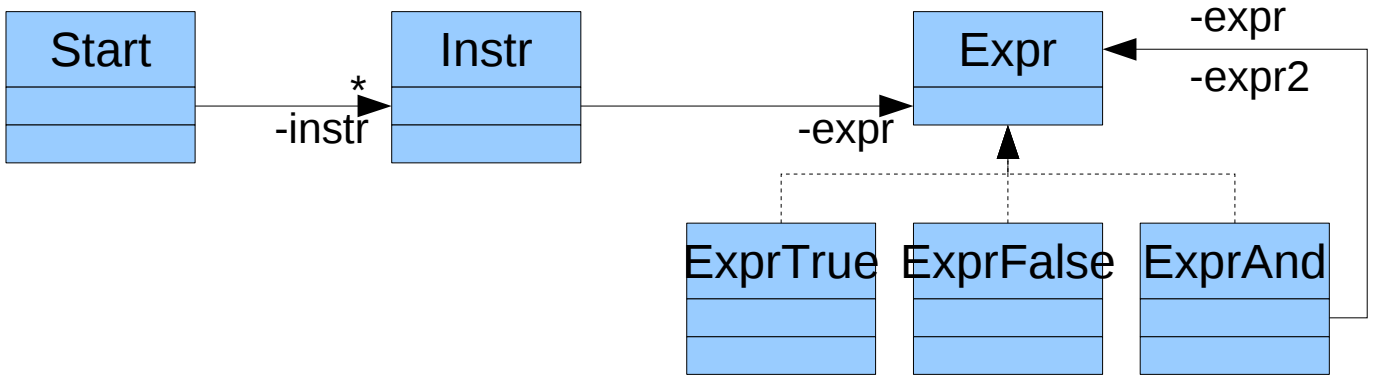
# Tatoo & AST

- Tatoo fourni un évaluateur de production qui construit l'AST

```
Reader reader = ...

TerminalEvaluator<CharSequence> terminalEvaluator = ...
ASTGrammarEvaluator grammarEvaluator =
  new ASTGrammarEvaluator();

Analyzers.run(reader,
                    terminalEvaluator,
                    grammarEvaluator,
                    null,    // start non terminal
                    null);   // version

Start start = grammarEvaluator.getStart();
```

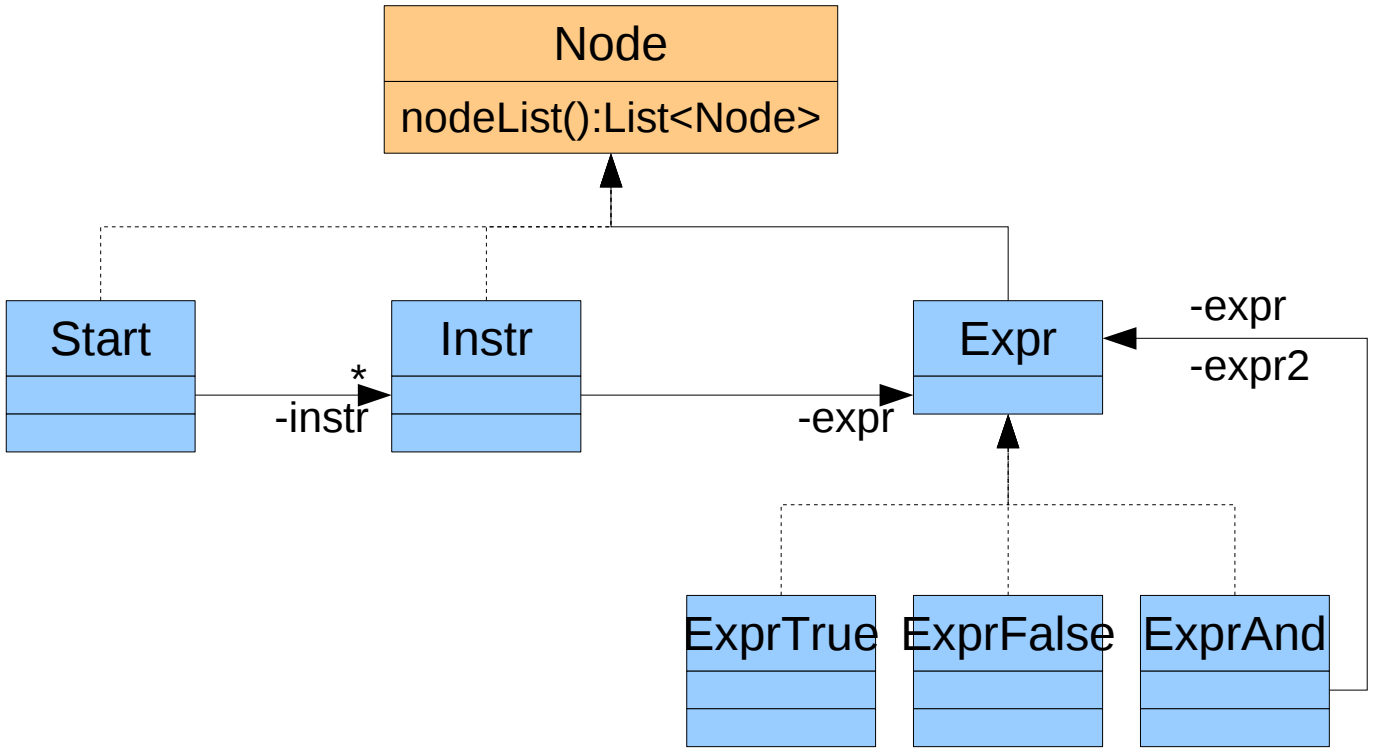
# Tatoo & AST

- AST de Tatoo
  - Une interface par non terminal
  - Une classe par production
    - Implante l'interface du non terminal en partie gauche
  - Règle spécial si un non terminal n'a qu'une seul production, alors l'interface n'est pas généré



# Tatoo & AST

- L'AST fourni aussi une vue générique
- Les classes de l'AST sont mutables, le type des noeuds est vérifié, le parent est calculé automatiquement



# TerminalEvaluator

- Dans le cas de l'AST, le terminal évaluateur doit renvoyer les feuilles de l'arbre

```
Reader reader = ...

TerminalEvaluator<CharSequence> terminalEvaluator = ...
ASTGrammarEvaluator grammarEvaluator =
  new ASTGrammarEvaluator();

Analyzers.run(reader,
                    terminalEvaluator,
                    grammarEvaluator,
                    null,    // start non terminal
                    null);   // version

Start start = grammarEvaluator.getStart();
```

# AST et report d'erreur

- Si une erreur se produit lors d'une des passes sémantiques, on veut indiquer à l'utilisateur où l'erreur s'est produite

- Solution: associer une position à chaque noeud de l'arbre

# AST et Location

- La solution consiste à passer le LocationTracker à l'ASTEvaluator et de remplir une table de hachage qui associe à un noeud sa position

```
final LocationTracker tracker = new LocationTracker();
LexerBuffer buffer = new ReaderWrapper(reader, tracker);

HashMap<Node, Location> locationMap = ...

ASTGrammarEvaluator grammarEvaluator =
  new ASTGrammarEvaluator() {
    protected void computeAnnotation(Node node) {
      locationMap.put(node, new Location(
        tracker.getLineNumber(), tracker.getColumnNumber()));
    }
  };
...
```

# AST généré

- Le problème avec un AST généré est que l'on ne peut pas le modifier sans perdre toutes les modifications lors de la prochaine génération
- Hors, en objet, les calculs se font sous forme de méthodes à l'intérieur des classes
- Solution: le Visitor Pattern

# Visitor Pattern

- Le Visitor Pattern (ou *double-dispatch*, 86) permet de spécifier un algorithme sur une hiérarchie de classes sans modifier celles-ci

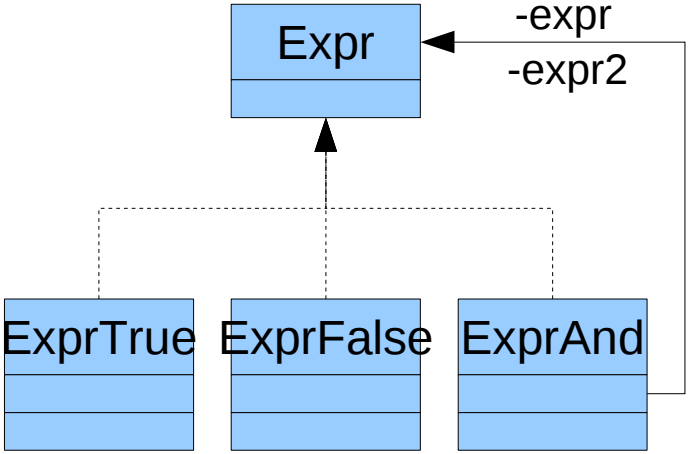
- Le Visitor Pattern requiert que la hiérarchie de classes soit écrite en vue d'utiliser le Visitor Pattern

- L'AST généré par Tatoo permet d'utiliser le Visitor Pattern

# Comprendre le Visiteur Pattern

- Evaluation des expressions booléennes sur l'AST

```
boolean eval(ExprTrue expr) {
  return true;
}

boolean eval(ExprFalse expr) {
  return false;
}

boolean eval(ExprAnd and) {
  return eval(and.getExpr()) &&
         eval(and.getExpr2()));
}
```



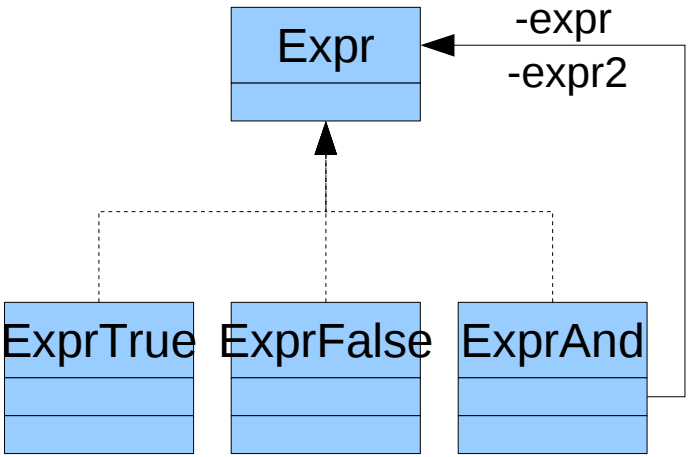

Ne compile pas !!!

eval(Expr) n'existe pas !

# Solution avec des instanceof

- Pas objet, lent, vraiment pas maintenable

```
boolean eval(Expr expr) {
  if (expr instanceof ExprTrue)
    return eval((ExprTrue)expr);
  if (expr instanceof ExprFalse)
    return eval((ExprFalse)expr);
  if (expr instanceof ExprAnd) {
    return eval((ExprAnd)expr);
  }...
}
boolean eval(ExprTrue expr) {
  return true;
}
boolean eval(ExprFalse expr) {
  return false;
}
boolean eval(ExprAnd and) {
  return eval(and.getExpr()) &&
         eval(and.getExpr2()));
}
```



# Visitor Pattern

- Spécifier le Visitor sous forme d'une interface paramétré par le type de retour
- Permet de ré-utiliser la même interface pour plusieurs algorithmes différents

```
public interface Visitor<R> {
  R visit(ExprTrue expr);
  R visit(ExprFalse expr);
  R visit(ExprAnd and);
}
```

```
public class EvalVisitor implements Visitor<Boolean> {
  public boolean eval(Expr expr) {
    return ???
  }
  public Boolean visit(ExprTrue expr) {
    return true;
  }
  public Boolean visit(ExprFalse expr) {
    return false;
  }
  public Boolean visit(ExprAnd and) {
    return eval(and.getExpr()) &&
           eval(and.getExpr2()));
  } }
```

# Visitor Pattern

- Comment appeler la bonne méthode visit ?
- Ajouter une méthode accept déclaré dans l'interface (Expr) et implanter dans les classes concrètes (ExprTrue, ExprFalse, ExprAnd)

```
public interface Expr {
  <R> R accept(Visitor<? extends R> visitor);
}
```

```
public interface Visitor<R> {
  R visit(ExprTrue expr);
  R visit(ExprFalse expr);
  R visit(ExprAnd and);
}
```

```
public class ExprTrue implements Expr {
  public <R> R accept(Visitor<? extends R> visitor) {
    return visitor.visit(this);
  }
}
```

```
public class ExprAnd implements Expr {
  public <R> R accept(Visitor<? extends R> visitor) {
    return visitor.visit(this);
  }
}
```

# Visitor Pattern

- Le code de la méthode accept dans les classes concrète est toujours le même mais ne peut pas être partagé car this est typé différemment.

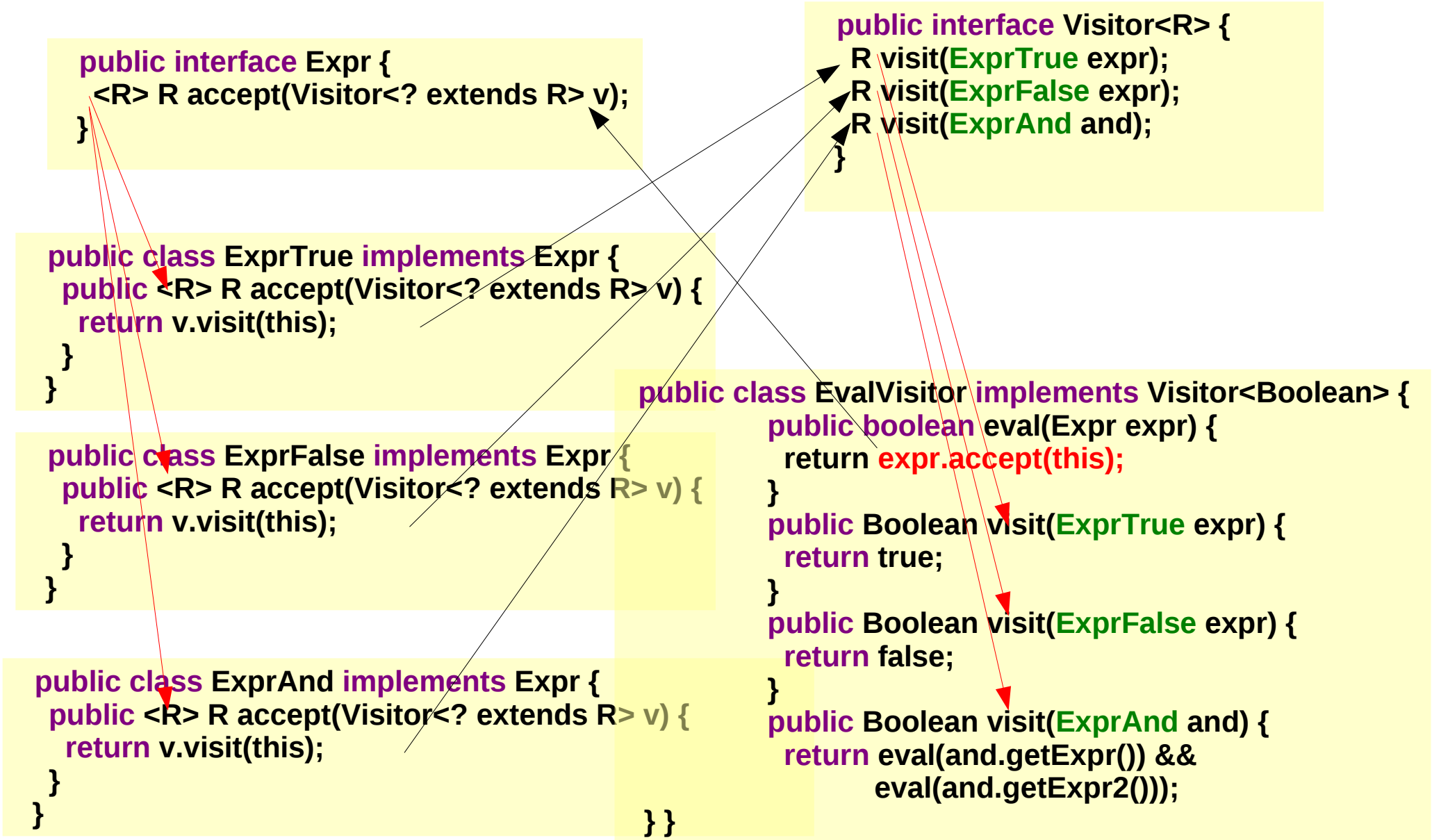
```
public class ExprTrue implements Expr {
  public <R> R accept(Visitor<? extends R> visitor) {
    return visitor.visit(this);
  }
}
```

Appel visit(ExprTrue)

```
public class ExprAnd implements Expr {
  public <R> R accept(Visitor<? extends R> visitor) {
    return visitor.visit(this);
  }
}
```

Appel visit(ExprAnd)

# Double dispatch



# Limitations du Visiteur original

- Le visiteur est spécifier sous forme d'interface
  - Ajout d'une nouvelle production
    - Ajout d'une nouvelle méthode
      - Il faut ré-écrire tous les visiteurs
  - Les méthodes visit ne sont que sur des types concrets, pas possible, on doit écrire le même code plusieurs fois si le traitement est le même pour des sous-types
  - Solution: Visitor Pattern avec des visits par défaut

# Visit par défaut

- Les méthodes visit sur des classes concrètes délégue à une méthode visit sur l'interface correspondante

- Les méthodes visit sur une interface sont protected car elle ne sont pas appelée directement

```
public abstract class Visitor<R> {
  protected R visit(Expr expr) {
    throw new AssertionError();
  }
  public R visit(ExprTrue expr) {
    return visit((Expr)expr);
  }
  public R visit(ExprFalse expr) {
    return visit((Expr)expr);
  }
  public R visit(ExprAnd and) {
    return visit((Expr)expr);
  }
}
```

# Exemple de visit par défaut

- On peut spécifier visit(ExprTrue) et visit(ExprFalse) en une seule visite

```
public abstract class Visitor<R> {
  protected R visit(Expr expr) {
    throw new AssertionError();
  }
  public R visit(ExprTrue expr) {
    return visit((Expr)expr);
  }
  public R visit(ExprFalse expr) {
    return visit((Expr)expr);
  }
  public R visit(ExprAnd and) {
    return visit((Expr)expr);
  }
}
```

```
public class EvalVisitor extends Visitor<Boolean> {
  public boolean eval(Expr expr) {
    return expr.accept(this);
  }
  protected Boolean visit(Expr expr) {
    return expr.getKind() == expr_true;
  }
  public Boolean visit(ExprAnd and) {
    return eval(and.getExpr()) &&
           eval(and.getExpr2()));
  }
}
```

# Attribut Hérité

- Les attributs synthétisés sont renvoyés en utilisant la valeur de retour
- Pour pouvoir spécifier des attributs hérités, il faut ajouter un paramètre

```
public abstract class Visitor<R, P> {
  protected R visit(Expr expr, P param) {
    throw new AssertionError();
  }
  public R visit(ExprTrue expr, P param) {
    return visit((Expr)expr, param);
  }
  public R visit(ExprFalse expr, P param) {
    return visit((Expr)expr, param);
  }
  public R visit(ExprAnd and, P param) {
    return visit((Expr)expr, param);
  }
}
```

```
public class ExprAnd implements Expr {
  public <R, P> R accept(Visitor<? extends R, ? super P> visitor, P param) {
    return visitor.visit(this);
  }
}
```

# Short-circuit

- Certaines passes sémantiques peuvent s'arrêter avant la fin du parcours complet de l'AST
- On utilise pour cela une exception.
  - Si pas besoin, on utilise RuntimeException

```
public abstract class Visitor<R, P, E extends Exception> {
 protected R visit(Expr expr, P param) throws E {
  throw new AssertionError();
 }
 public R visit(ExprTrue expr, P param) throws E {
  return visit((Expr)expr, param);
 }
 public R visit(ExprFalse expr, P param) throws E {
  return visit((Expr)expr, param);
 }
 public R visit(ExprAnd and, P param) throws E {
  return visit((Expr)expr, param);
 } }
```

# Analyze sémantique

- L'analyse sémantique s'effectue traditionnellement en plusieurs passes (même en C)
- Les passes habituelles sont :
  - Enter
  - TypeCheck
  - Flow
  - Desugar
  - Gen

# Enter

- Visite la partie déclarative du language
  - Est composé de plusieurs passes
- Par exemple
  - On doit lire d'abord les types avant les champs et les prototypes des fonctions

```
record Link {
  int value;
  Link next;
}

def display(Link link) {
  for (Link l = link; l != null; l = l.next) {
    print l.value;
  }
}
```

Passe 1: on découvre le type Link

Passe 2: on découvre les champs et les prototypes des fonctions

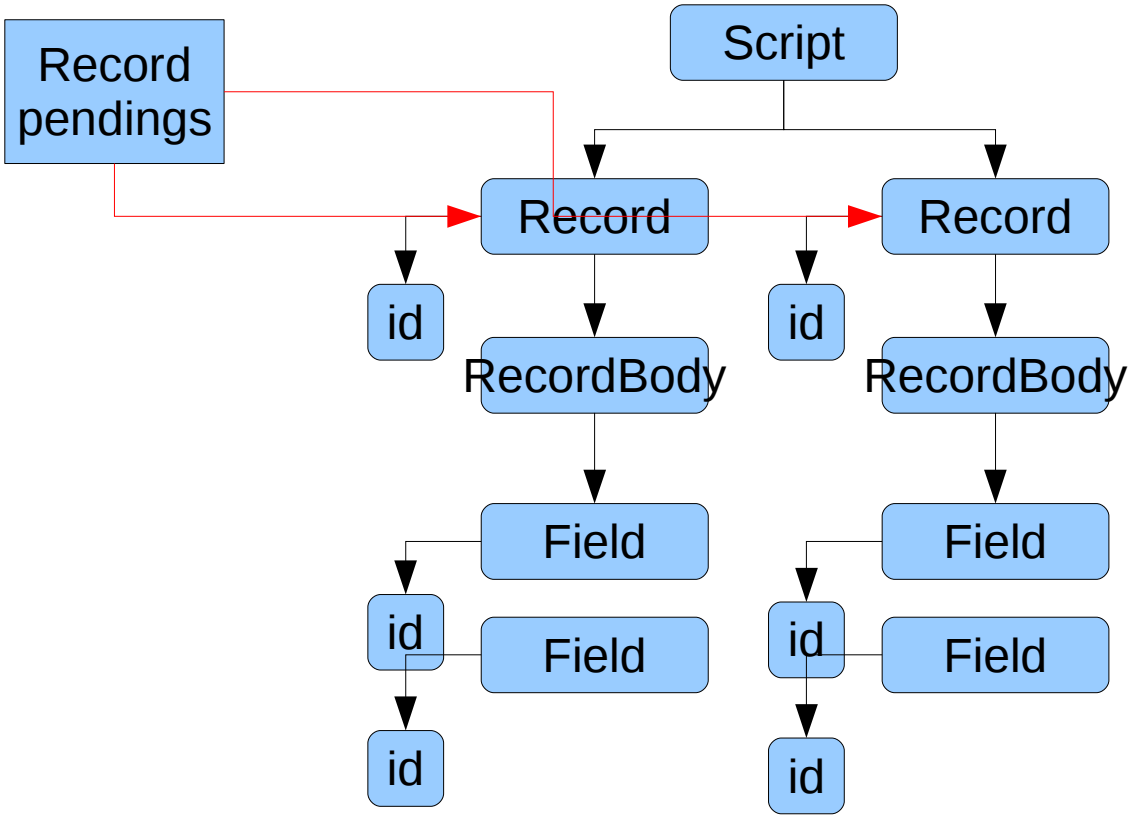
# Astuce pour éviter plusieurs visites

- Lors de la visite, on enregistre dans une collections (appelée pendings) les nœuds de l'arbre que devra visiter plus tard

- Dans l'exemple, on parcours les records et les fonctions et on enregistre les records et les fonctions pour pouvoir les parcourir lors de la seconde passe

# Pendings

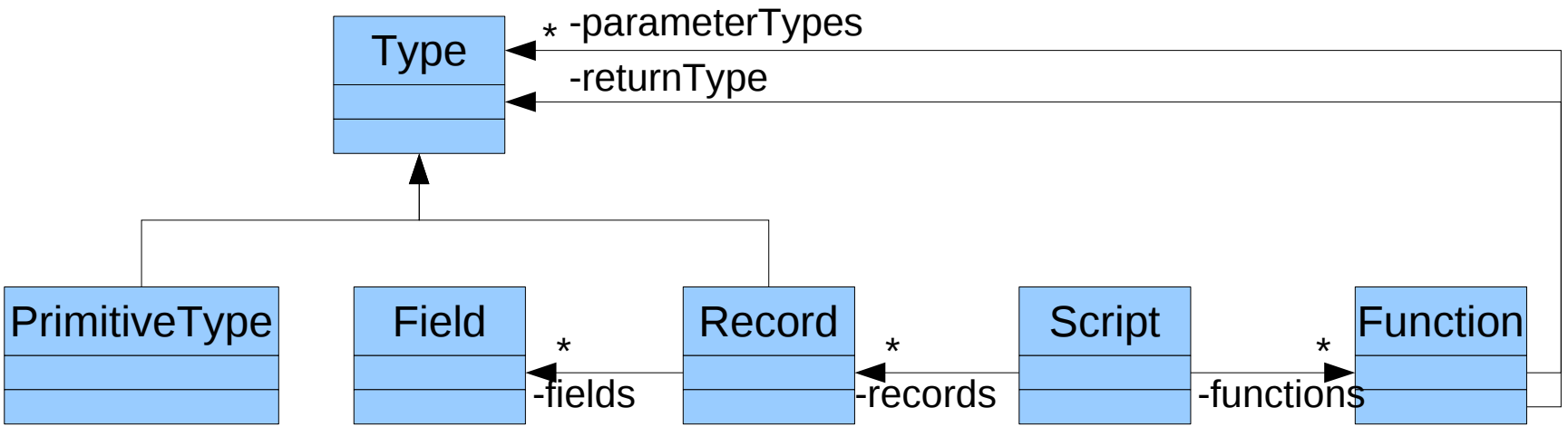
- Lors de la première passe, on sauvegarde les nœuds de l'arbre qu'il faudra traiter ultérieurement




```
record Point {
  int x;
  int y;
}
record Line {
  Point start;
  Point end;
}

def translate(Line line, int dx, int dy) {
  translate(line.start, dx, dy);
  translate(line.end, dx, dy);
}
def translate(Point p, int dx, int dy) {
   p.x = p.x + dx;
   p.y = p.y + dy;
}
```

# Enter

- La passe Enter crée le modèle du langage
- Dans l'exemple sont créés
  - Des objets Record, Field et Function
  - Des objets Type correspondant aux Records (ou l'objet Record peut aussi être un Type)



# TypeCheck

- Passe de vérification de type
  - On vérifie lors des assignations que le lhs (left hand side) est compatible avec le rhs (right hand side)
  - On vérifie qu'une fonction existe suivant le type des paramètres (attention à la surcharge)
  - On vérifie que le type de l'expression de return est bien compatible avec le type de retour de la fonction
  - On vérifie que les conditions du if, for, while sont bien compatible avec des booléens
  - etc.

# Déclaration/Utilisation

- Il faut trouver pour l'utilisation d'un identificateur la déclaration correspondante
- On utilise une table des symboles

```
record Link {
  int value;
  Link next;
}

def display(Link link) {
  for (Link l = link; l != null; l = l.next) {
    print l.value;
  }
}
```

déclaration
utilisation

# Language typé dynamiquement

- Pour les languages typé statiquement (C, Java), la passe de TypeCheck est une passe de vérification de type

- Pour les languages typé dynamiquement (PHP, Python, Ruby), la passe de TypeCheck est une passe d'inférence de type
  (on essaye de trouver les types pour accélérer l'exécution)

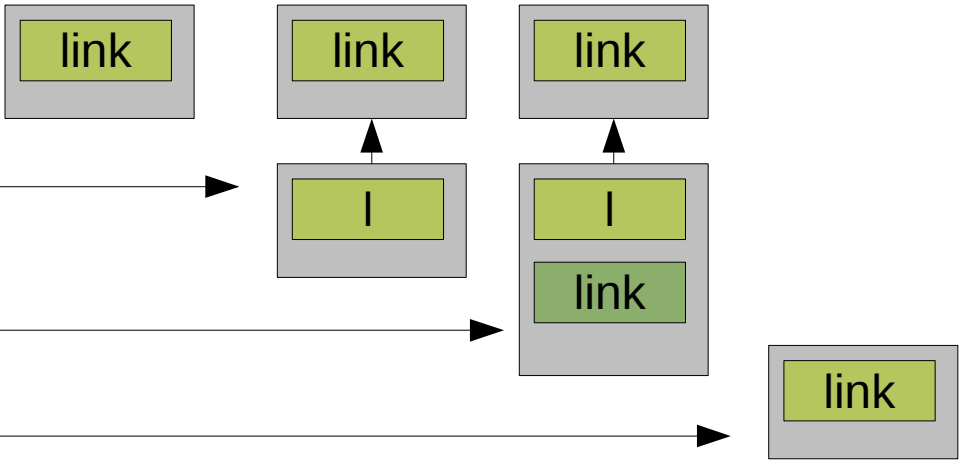
# Table des symboles

- Associe à un nom (ou nom+types) un symbol (une variable)
  - Pile de Scope, un Scope est une table de hachage
  - Algorithme :
    - Rentre dans un block
      - Ajout d'un nouveau scope
    - Déclaration d'une variable
      - Ajout de la variable dans le scope courant
    - Utilisation d'un identificateur
      - Recherche dans le scope courant puis dans les scopes de la pile
    - Sort d'un block
      - Suppression du scope courant

# Exemple

- Evolution de la table des symboles lors du typage d'un fonction

```
def display(Link link) {
  for (Link l = link; l != null; l = l.next) {
    int link = l.value;
    print link;
  }
}
```

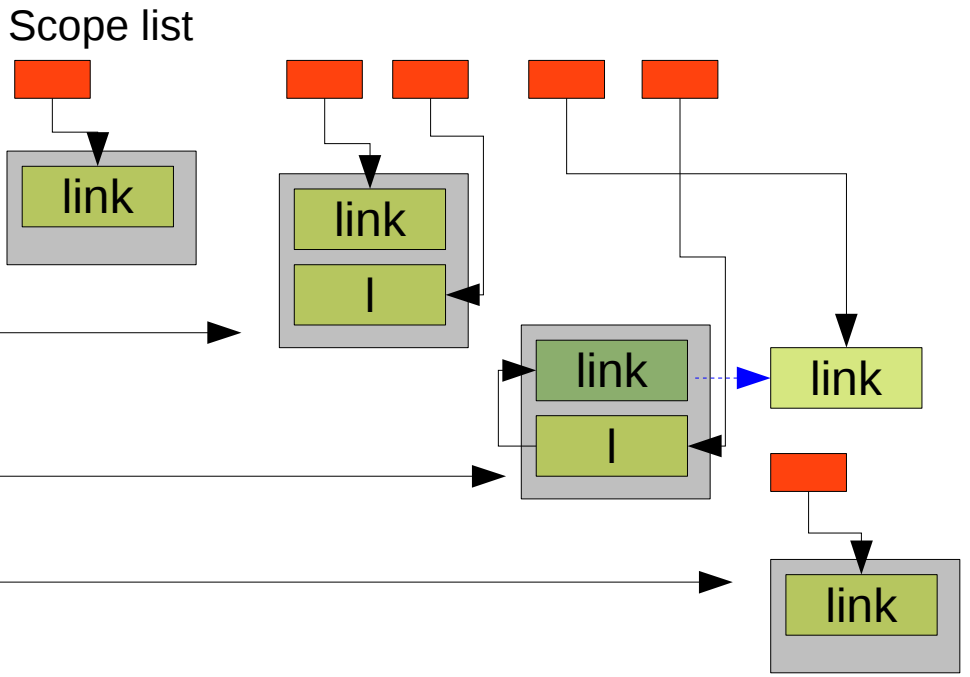



Entrer dans un block: O(1)
Sortir d'un block: O(1)
Rechercher un identificateur: O(taille de la pile de scope)

# Autre implantation

- Il existe une autre implantation un peu plus rapide mais + consommatrice de mémoire

```
• def display(Link link) {
    for (Link l = link; l != null; l = l.next) {
      int link = l.value;
      print link;
    }
  }
```




Entrer dans un block: O(1)
Sortir d'un block: O(nombre de variable du scope courant)
Rechercher un identificateur: O(1)

# Tables des symboles

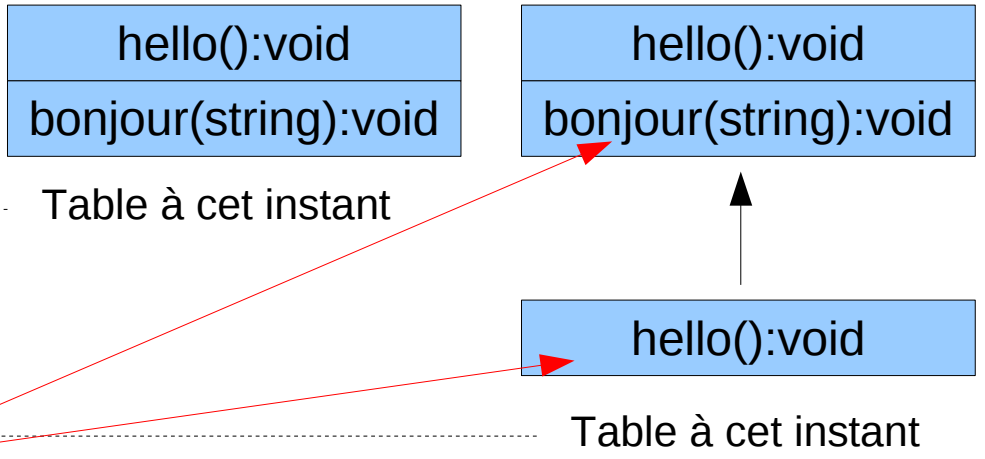
- Il n'y a pas qu'une table des symboles
- En Java, il y a une table
  - pour les types
  - pour les méthodes
  - pour les variables

- On peut alors utiliser le même identificateur pour un type, une méthode et une variable

# Symboles et héritage

- Si il y a héritage (ou mixin, trait etc), la table des symboles du sous-type fait référence à la table des symboles du super types

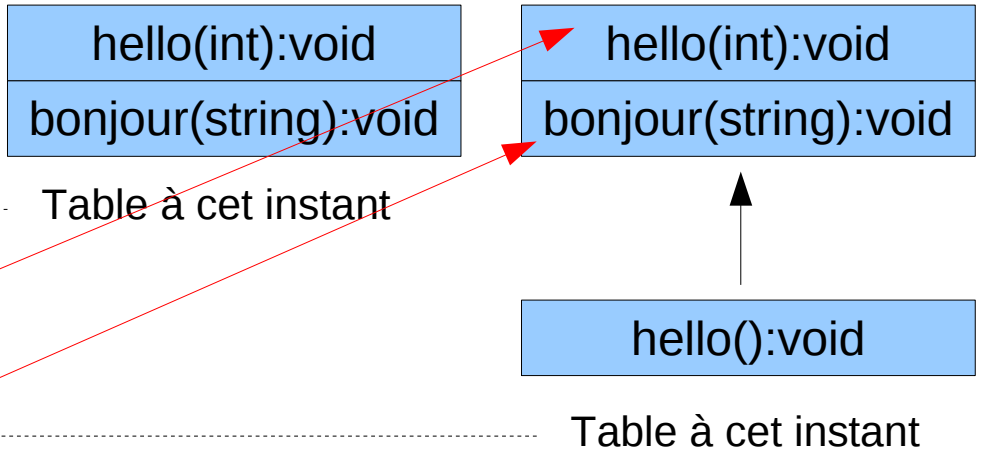
```
• class A {
    def hello() { ... }
    def bonjour(string text) { ... }

  }

•class A <: B {
    def hello() { ... }

    def main([string] args) {
      hello();
      bonjour(args[0]);
    }
  }
```



# Symboles et surcharge

- La surcharge est gérée en permettant d'avoir des symboles ayant le même nom mais des types de paramètre différents

```
• class A {
    def hello(int value) { ... }
    def bonjour(string text) { ... }

  }

•class A <: B {
    def hello() { ... }

    def main([string] args) {
      hello(3);
      bonjour(args[0]);
    }
  }
```



# Bindings (liens)

- Suivant les compilateurs, le lien entre l'utilisation d'une variable et sa déclaration peut être sauvegardé ou pas

- Par exemple pour eclipse ou netbeans le liens est gardé et il est bi-directionnel, cela permet d'effectuer des refactoring simplement

# Le visiteur TypeCheck

- La ou les table des symboles sont des attributs hérités
- Le type de l'expression est un attribut synthétisé

- Comme les passes suivantes auront aussi besoin de savoir le type des expressions, on les stockes dans table de hachage noeud de l'AST => Type

# TypeCheckVisitor

## Déclaration habituel du visiteur TypeCheck

```
public class TypeCheckVisitor extends Visitor<Type, SymbolTable, RuntimeException> {
  private final HashMap<Node, Type> typeMap =
    new HashMap<Node, Type>();

  public Type typeCheck(Node node) {
    Type type = node.accept(this);
    if (type != null)
      typeMap.put(node, type);
    return type;
  }
  ...
}
```

# Enter, TypeCheck et erreurs

- Les passes enter et typecheck doivent faire de la reprise sur erreur
  - Continuer l'algorithme même si celui-ci a détecté une erreur
  - Cela permet à l'utilisateur de corriger plusieurs erreurs d'un coup
    - Il faut éviter de signaler les erreurs causées par une erreur
- Si il y a une erreur
  - Enter ne crée pas l'objet dans le modèle
  - TypeCheck renvoie un type particulier appelé « type errreur »

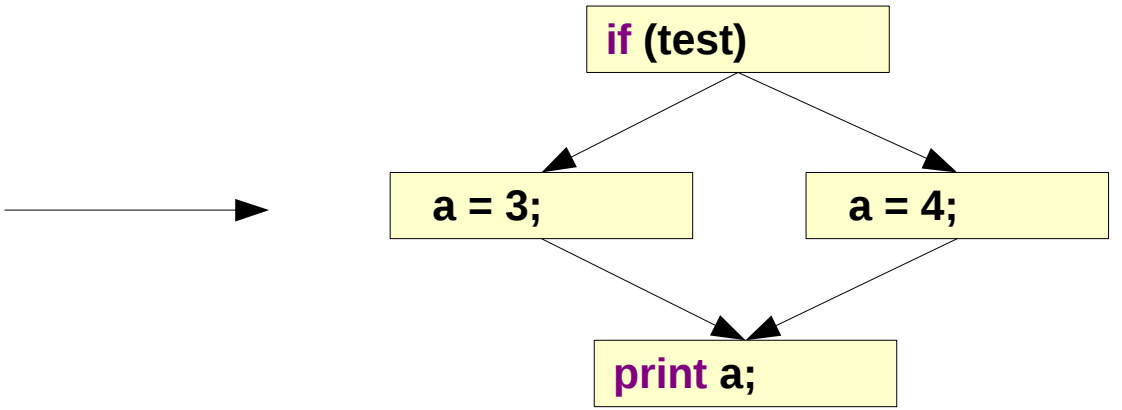
# Flow

- Parcours le code comme un graphe pour calculer/vérifier certaines propriétés
- Propriétés habituelles:
  - Détection de code mort (dead code)
  - Détection d'utilisation de variable non initialisé
  - Détection de modification de variable non-modifiable
  - break est bien dans une boucle, break nommé possède un label qui existe
  - etc.

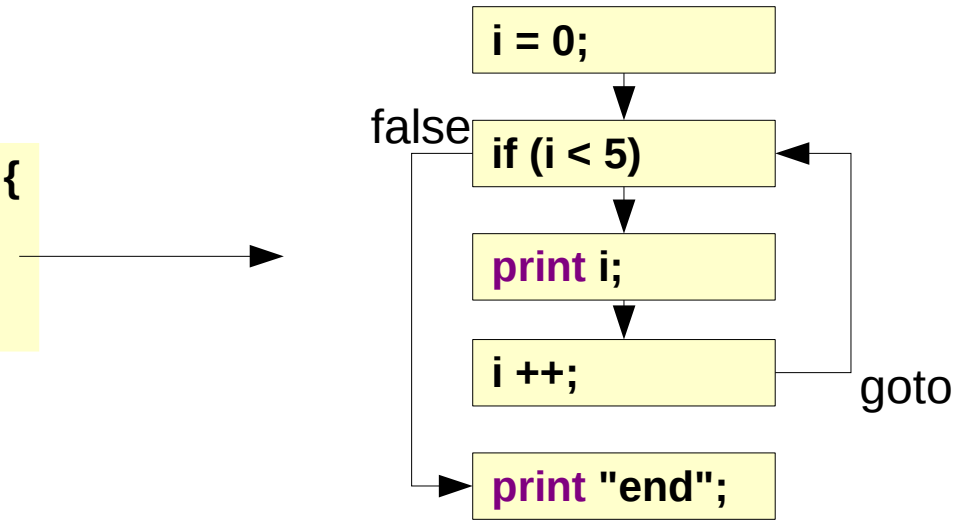
# Code comme un graphe

- Chaque branchement crée un arc dans le graphe

```
int a;
if (test) {
  a = 3;
} else {
  a = 4;
}
print a;
```




```
for (int i = 0;i < 5 ;i = i + 1) {
  print i;
}
print "end";
```



# Détection de code mort

- Après un return, throw, break ou continue, on positionne le flag dead à vrai
- Si on parcours une instruction avec flag dead alors le code n'est pas atteignable
- La jonction de deux arcs revient à faire un « ou » sur les deux flags dead

```
if (test) {
  throw "argh"
} else {
  return;
}
print "i'm dead";
```

```
while (condition) {
  break;
  print "i'm dead";
}
```

```
while (condition) {
  return;
}
print "i'm not dead";
```

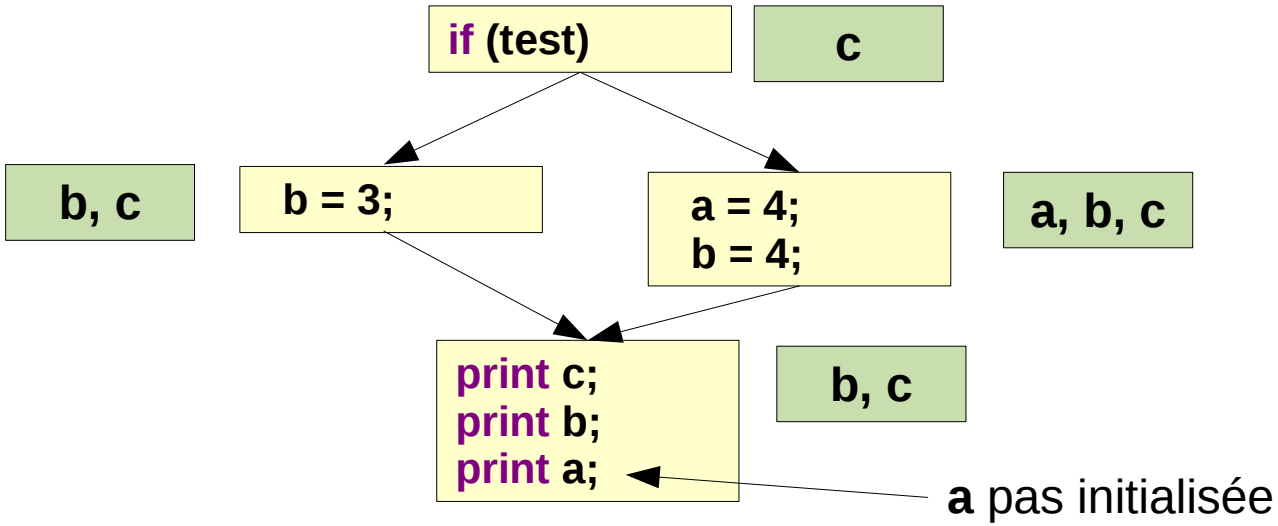
Dead code

```
for (;condition; i = i + 1) {
  return;
}
```

# Utilisation de variable non initialisée

- On maintient un ensemble de toutes les variables initialisées
- Lors d'un branchement, on duplique l'ensemble
- Lors d'une jonction, on fait l'intersection des ensembles

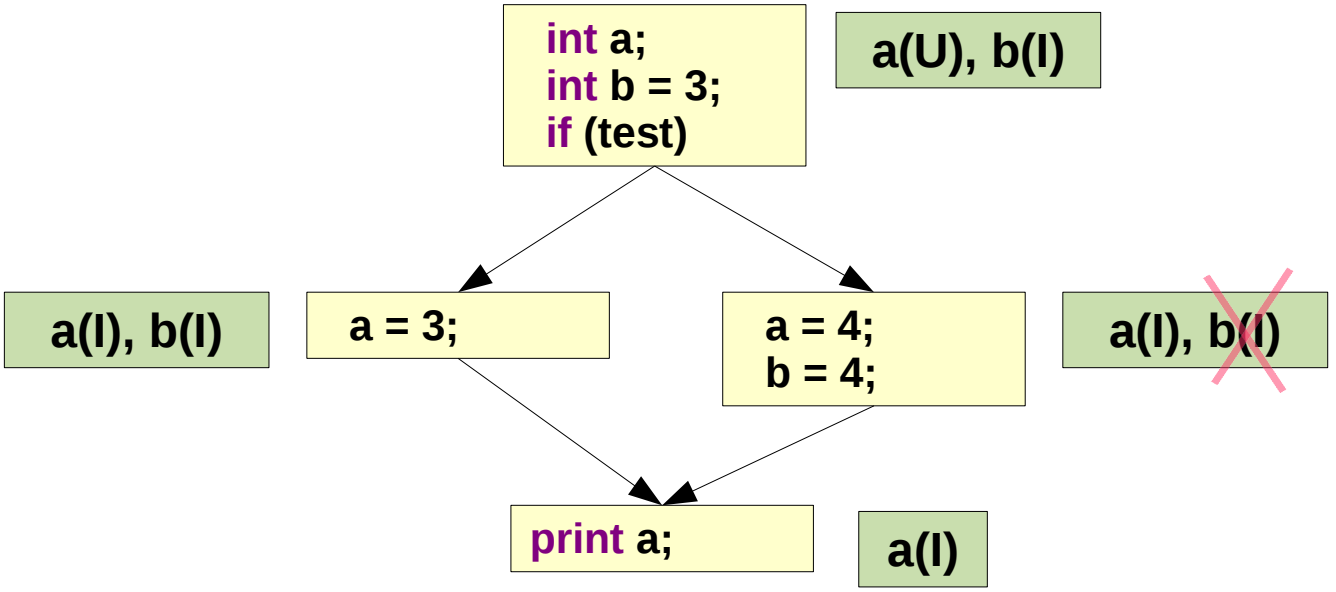
```
int a, b, c=5;
if (test) {
  b = 3;
} else {
  a = 4;
  b = 4;
}
print c;
print b;
print a;
```



# Modification de variable non-modifiable

- On maintient pour chaque variable readonly un état (U=unitialized, I=initialized)
- Un branchement duplique l'ensemble
- Si lors d'une jonction l'état n'est pas identique => erreur

```
readonly int a;
readonly int b = 3;
if (test) {
  a = 3;
} else {
  a = 4;
  b = 4;
}
print a;
```



# Break/Continue => Loop

- Pile des boucles + hashmap label => boucle
  - A chaque boucle, on empile, si il y a un label, on stocke dans la hashmap
  - A chaque sortie de boucle, on dépile, on enlève de la hashmap

```
bar: while (condition) {
  continue foo;
}
```

Erreur

- Si on voit :
  - Un break/continue, on prend le sommet de pile
  - Un break/continue label, on recherche dans la hashmap

```
if (test) {
  break;
}
```

Erreur

# Desugar

- Transforme certaines syntaxes de haut niveau en syntaxes de bas niveau
- Exemples en Java:
  - Foreach => for(;;) loop
  - Classe anonyme/classe interne en classe top-level
  - Enumération en classe classique
  - Erasure
    - Type paramétré => type classique
  - Switch sur les String en switch classique
  - etc.

# Desugar Visitor

- Desugar effectue des ré-écritures de l'AST, c'est un visiteur qui renvoie un nouvel AST.

- Tatoo fourni un visiteur VisitorCopier qui par défaut duplique les noeuds de l'AST.

```
public class DesugarVisitor extends VisitorCopier<DesugarEnv, RuntimeException> {
  public Node desugar(Node node) {
    return node.accept(this);
  }
  ...
}
```

# Exemple: Desugar foreach

- Desugar change une partie de l'AST en la remplaçant par un autre arbre utilisant des instructions plus simple.

```
int[] array = ...
foreach (int value in array) {
  ...
}
```


```
int[] array = ...
{
  int $length = array.length;
  for (int $index=0; $index<$length;$index++) {
    int value = array[$index];
    ...
  }
}
```

```
public Node visit(Foreach foreach, DesugarEnv env) {
  Instr lengthDecl = new DeclInstr(new TokenId("$length"),
    new FieldAccess(new ExprId(foreach.getId()),
    new TokenId("length")));
  Block block = desugar(foreach.getBlock(), env);
  block.add(0, new DeclInstr(...));
  Instr forInstr = new ForInstr(new DeclInstr(...), new ExprLt(...), new InstrExpr(...), block);
  return new Block(Arrays.asList(lengthDecl, block));
}
```

# Desugar et typage

- Desugar intervient après la passe de typage donc les informations de type ne sont pas calculé sur l'AST généré

- Il faut ajouter les informations de type pendant la passe de desugar sinon elle ne seront pas disponible pour la passe de génération

# Switch sur les String

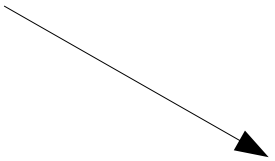
- Plusieurs implantations possible
  - If equals...else
    - pas efficace si beaucoup de cases)
  - Table de hachage
    - plus générique mais gourmand en mémoire
  - Switch sur les hashcode
    - attention aux collisions

```
String s = ...
switch(s) {
  case "toto":
    print "hello toto";  //fallthrough
  case "titi":
    print "hello zorblug";
    break;
}
```

# Switch sur les String

- If equals...else
  - en cas de fallthrough, on fait un goto vers le case suivant

```
String s = ...
switch(s) {
  case "toto":
    print "hello toto";  //fallthrough
  case "titi":
    print "hello zorblug";
    break;
}
```

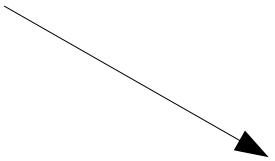


```
String s = ...
if (s.equals("toto")) {
  print "hello toto";
  goto nextcase; //fallthrough
} else {
  if (s.equals("titi")) {
    nextcase:
    print "hello zorblug";
    goto end;
  }
}
end:
```

# Switch sur les String

- Table de hachage
  - Permet de stocker n'importe qu'elle object **constant**

```
String s = ...
switch(s) {
  case "toto":
    print "hello toto";  //fallthrough
  case "titi":
    print "hello zorblug";
    break;
}
```



```
global map = {
  "toto": 1,
  "titi": 2
};

{
  String s = ...
  nullcheck(s);
  int index = map.get(s);
  switch(index) {
    case 1:
      print "hello toto";  //fallthrough
    case 2:
      print "hello zorblug";
      break;
  }
```

# Switch sur les String

- Switch sur le hashcode
  - Le compilateur et la VM doivent utiliser la même fonction de hachage
  - On fait un double-switch

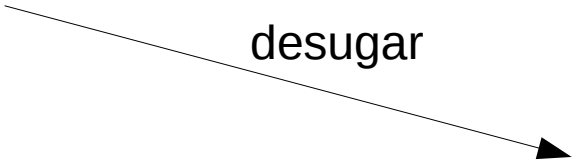
```
String s = ...
switch(s) {
  case "toto":
    print "hello toto";  //fallthrough
  case "titi":
    print "hello zorblug";
    break;
}
```

```
String s = ...
int index = -1;
switch(s.hashCode()) {
  case 145673: // "toto".hashCode()
    if ("toto".equals(s))
      index = 1;
    break;
  case 788966: // "titi".hashCode()
    if ("titi".equals(s))
      index = 2;
    break;
}
switch(index) {
  case 1:
    print "hello toto";  //fallthrough
  case 2:
    print "hello zorblug";
    break;
}
```

# Desugar des closures

- Une closure est une expression qui capture des variables locales sous forme d'une fonction
- Voici une façon de faire la traduction

```
int a = 2;
for (int i =0; i < 5; i = i + 1) {
  (int,int):int f = lambda(int x, int y) x * a + y * i;
  print f.invoke(2, 3);
}
```




```
class Lambda {
  readonly int a;
  readonly int i;
  Lambda(int a, int i) {
    this.a = a;
    this.i = i;
  }
  int invoke(int x, int y) {
    return x * a + y * i;
  }
}
...
int a = 2;
for (int i =0; i < 5; i = i + 1) {
  XXX f = new Lambda(a, i);
  print f.invoke(2, 3);
}
```

# Desugar des closures

- Le visiteur substitue à la construction lambda, l'allocation de la classe correspondante

```
int a = 2;
for (int i =0; i < 5; i = i + 1) {
  int,int:int f = lambda(int x, int y) x * a + y * i;
  print f.invoke(2, 3);
}
```

```
expr = 'lambda' '(' parameter/','* ')' expr
       | ...
       ;
parameter = type 'id'
                ;
```

```
public Node visit(ExprLambda lambda, DesugarEnv env) {
  Expr expr = desugar(lambda.getExpr(), env);
  List<Var> vars = collectFreeVariable(expr);
  String className = generateLambdaClass(vars, expr, env);
  List<Expr> arguments = new ArrayList<Expr>();
  for(Var var: vars) {
    arguments.add(new ExprId(new IdToken(var.getName())));
  }
  return new ExprNew(new IdToken(className), arguments);
}
```

# Desugar des fonctions types

- Le type fonction peut être traduit en un type abstrait ayant une méthode qui prend deux ints (II) et qui renvoie un int I

```
class Lambda extends (II)I {
  readonly int a;
  readonly int i;
  ...
  int invoke(int x, int y) {
    return x * a + y * i;
  }
}
...
int a = 2;
for (int i =0; i < 5; i = i + 1) {
  III f = new Lambda(a, i);
  print f.invoke(2, 3);
}
```

```
trait (II)I {
  int invoke(int x, int y);
}
```

# Optimisations

- Certaines optimisations peuvent être faite pendant la passe de desugarisation
  - Constant Folding
  - Dead-code elimination
  - Common sub-expression removal
  - Loop hoisting
  - etc

  cf cours sur les optimisations

# Generation

- La passe de génération de code est vu dans le cours suivant.