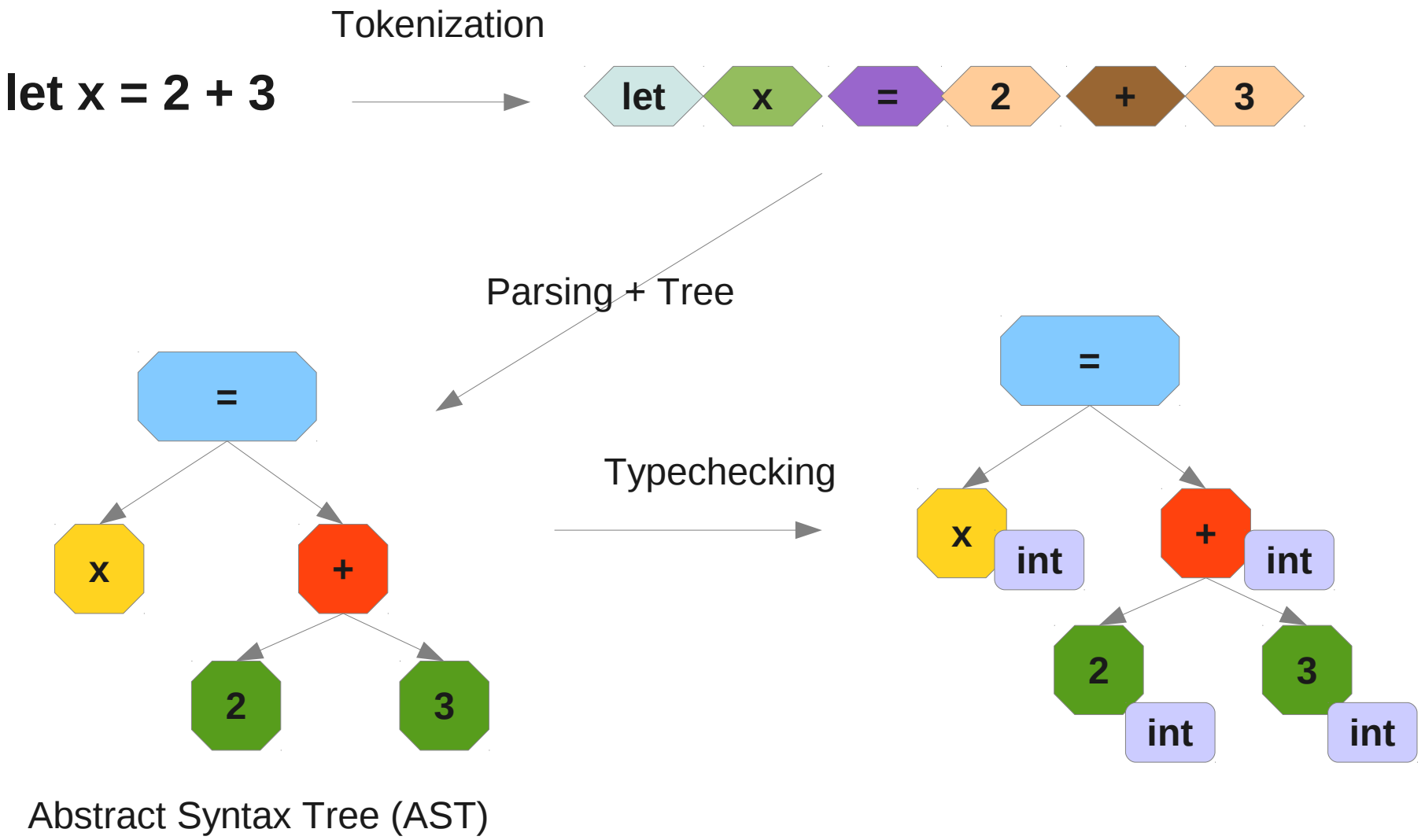
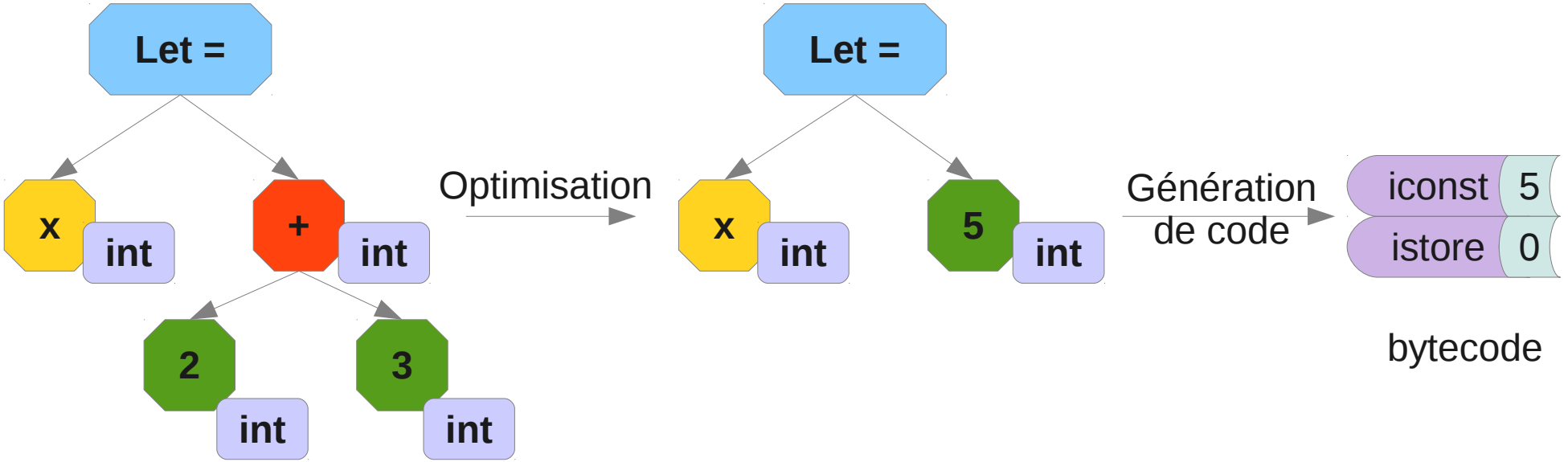
Machines Virtuelles
et bazard autour

Rémi Forax

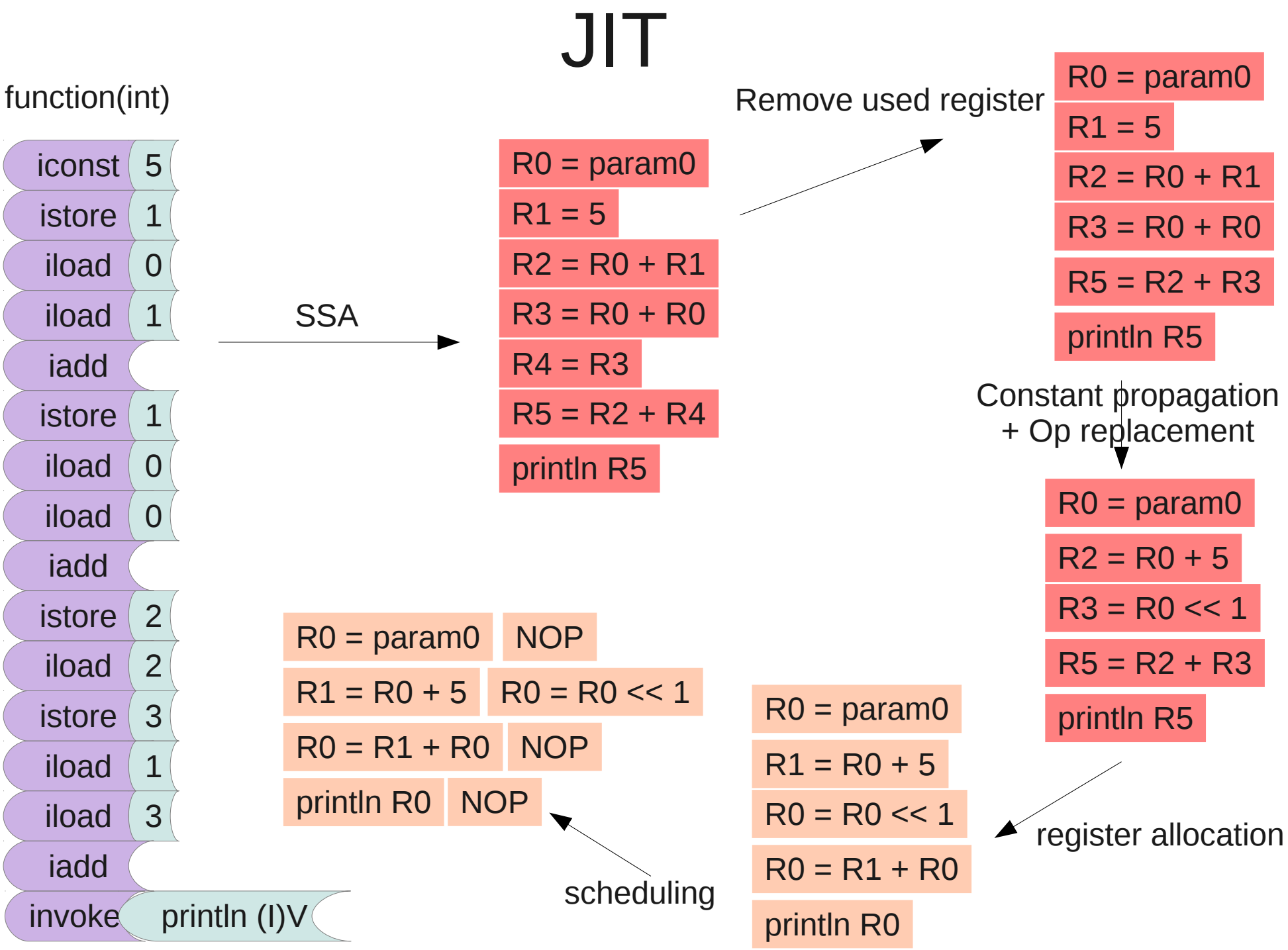
Compilation



Génération de code

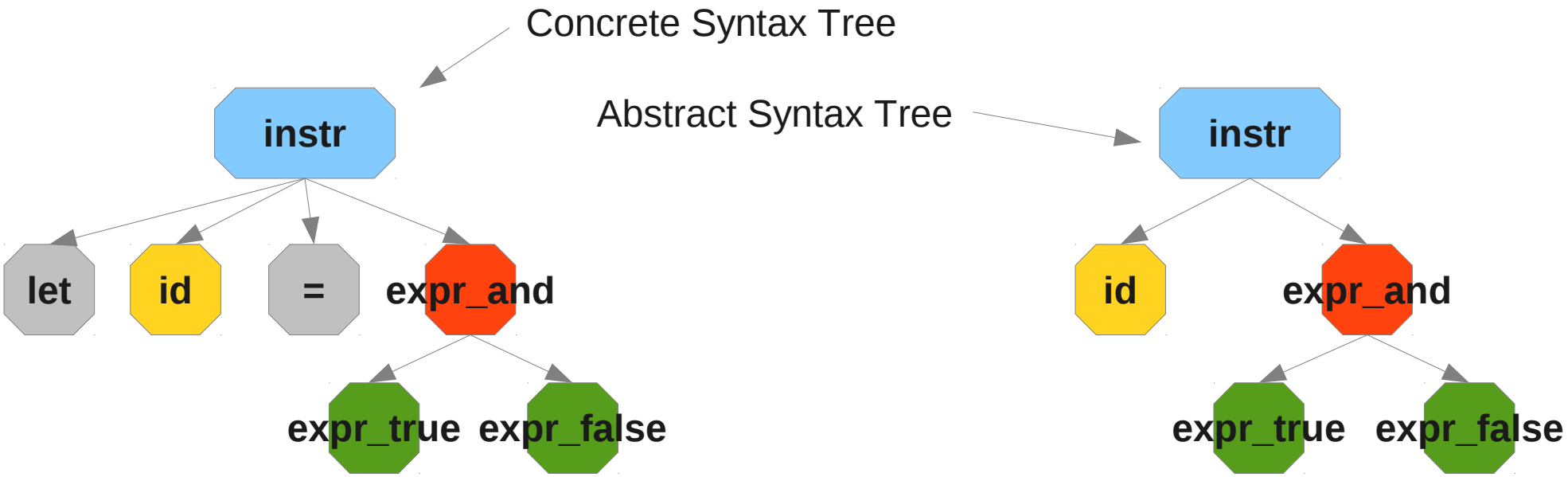


JIT


function(int)
iconst 5
istore 1
iload 0
iload 1
iadd
istore 1
iload 0
iload 0
iadd
istore 2
iload 2
istore 3
iload 1
iload 3
iadd
invoke println (I)V
SSA
R0 = param0
R1 = 5
R2 = R0 + R1
R3 = R0 + R0
R4 = R3
R5 = R2 + R4
println R5
Remove used register
R0 = param0
R1 = 5
R2 = R0 + R1
R3 = R0 + R0
R5 = R2 + R3
println R5
Constant propagation
+ Op replacement
R0 = param0
R2 = R0 + 5
R3 = R0 << 1
R5 = R2 + R3
println R5
register allocation
R0 = param0
R1 = R0 + 5
R0 = R0 << 1
R0 = R1 + R0
println R0
scheduling
R0 = param0
NOP
R1 = R0 + 5
R0 = R0 << 1
R0 = R1 + R0
NOP
println R0
NOP

AST

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



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

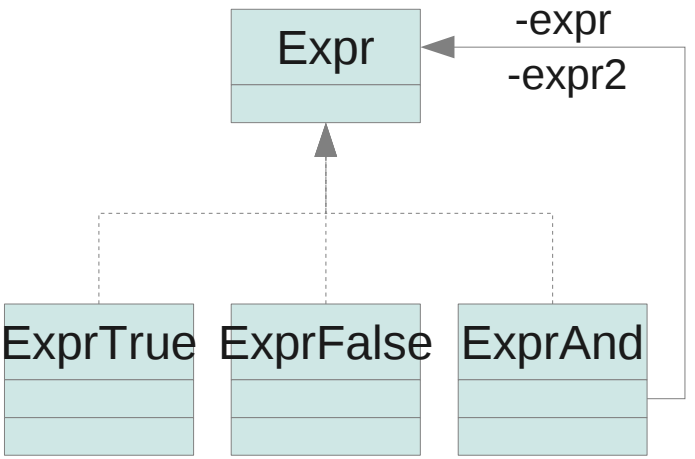
Comprendre le Visiteur Pattern

- 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()));
}
```



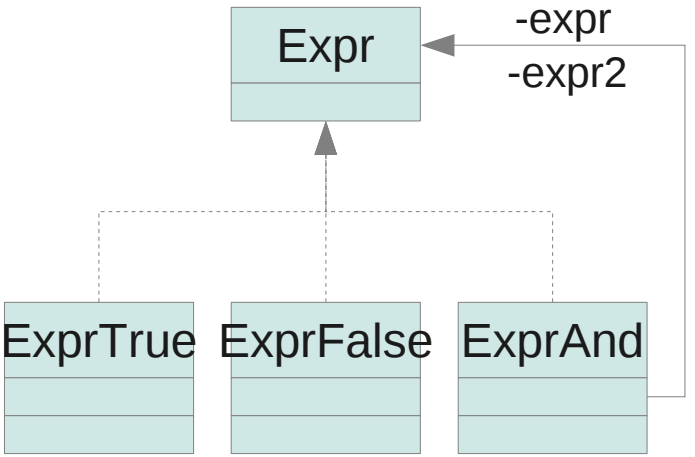
Ne compile pas !!!

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

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

```
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> v) {
    return v.visit(this);
  }
}
```

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

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

```
public class EvalVisitor implements Visitor<Boolean> {
  public boolean eval(Expr expr) {
    return expr.accept(this);
  }
  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()));
  }
}
```

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);
  }
}
```

Analyze sémantique

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

* optionel

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;
}

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

déclaration
utilisation

Exemple

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

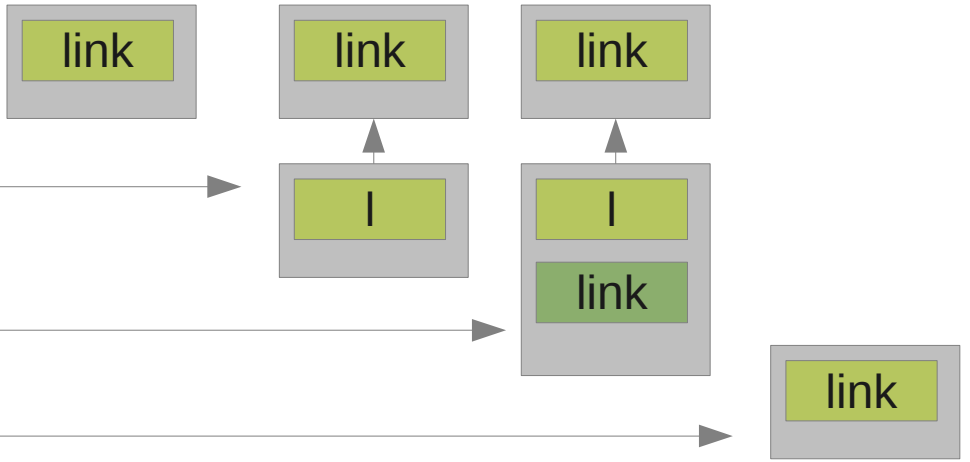
```
fun display(Link link) {

  for (Link l = link; l != null; l = l.next) {

    int link = l.value;
    print link;

  }

}
```



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;
   }
   ...
 }
```

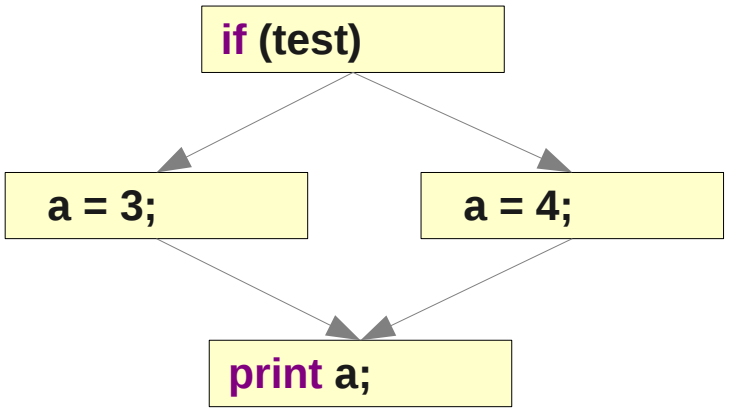
Flow

- 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.

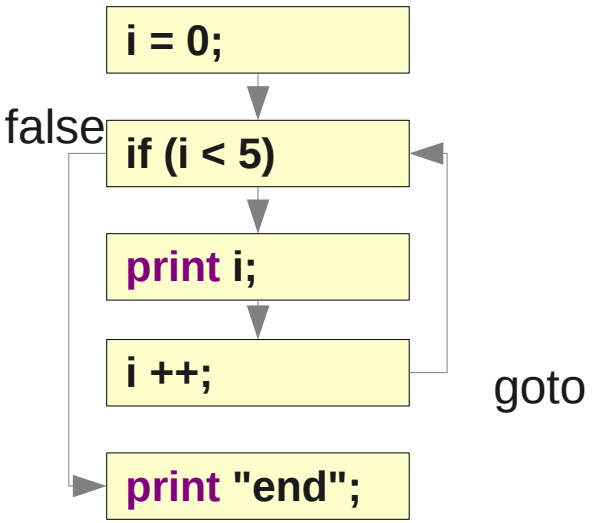
Code comme un graphe

- 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";
```

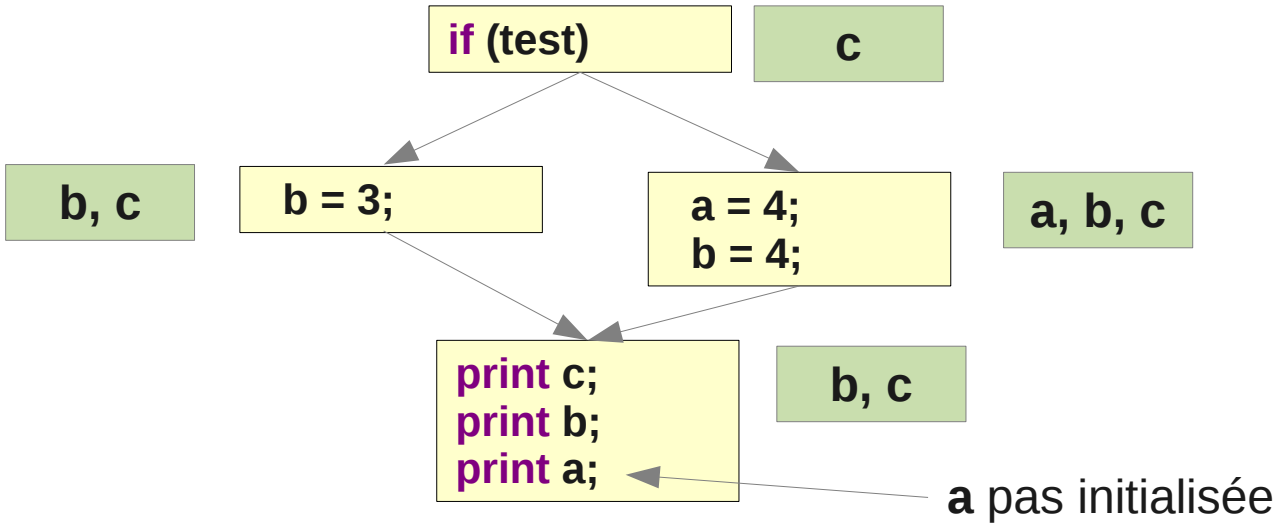
Dead code

```
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

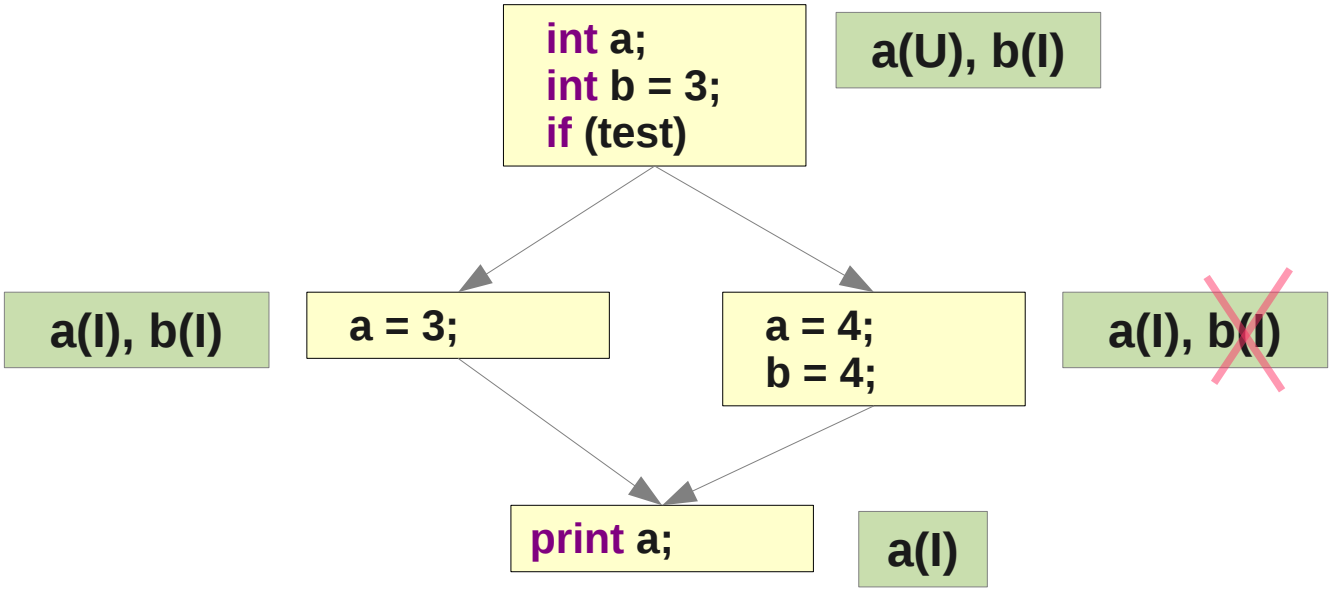
```
int a, b, c=5;
if (test) {
  b = 3;
} else {
  a = 4;
  b = 4;
}
print c;
print b;
print a;
```



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

Generation

- La génération est dépendante de ce que l'on génère
  - Pour du bytecode, il n'est pas nécessaire d'utiliser une représentation intermédiaire car c'est déjà une représentation intermédiaire
  - Pour du code machine, on utilisera la représentation SSA qui permet d'effectuer facilement des optimisations puis une représentation spécifique au code cible

Code intermédiaire

- Code proche de l'assembleur des machines cibles mais pas spécifique à une machine
- Il existe plusieurs sortes de codes intermédiaires

- Caractéristiques communes :
  - Nombre illimité de variables (registres)
  - Instructions de sauts conditionnels ou non pour représenter les tests et boucles

Codes intermédiaires

- Plusieurs sortes de code intermédiaire
  - Code 3 adresses
    - représentation à base de registres
  - Code à base d'arbre,
    - Représentation avec une pile
  - Static Single Assignment form (SSA)
    - 1 variable ne peut être affecté qu'une seul fois
- Dans un vrai compilateur, plusieurs formes peuvent co-exister

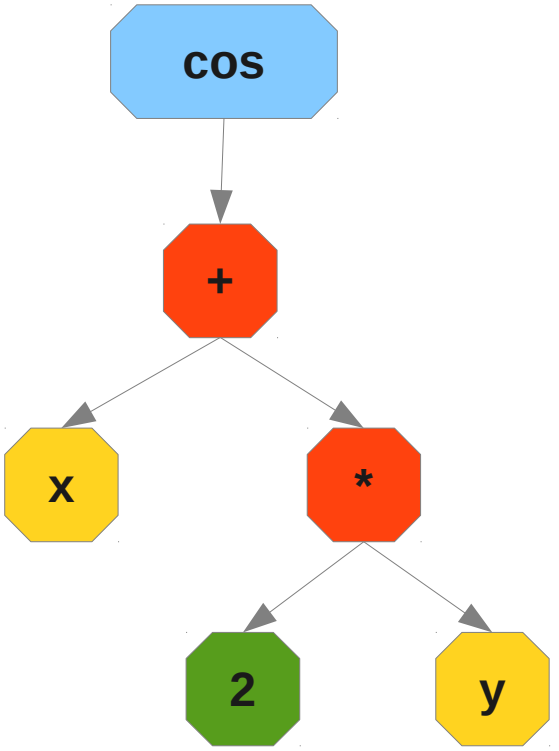
Code à base d'arbre

- Les résultats intermédiaires sont stockés sur la pile

  a = cos(x+2*y)

  devient

  iload x
  iload y
  iconst_2
  imult
  iadd
  invokestatic Math.cos (I)I



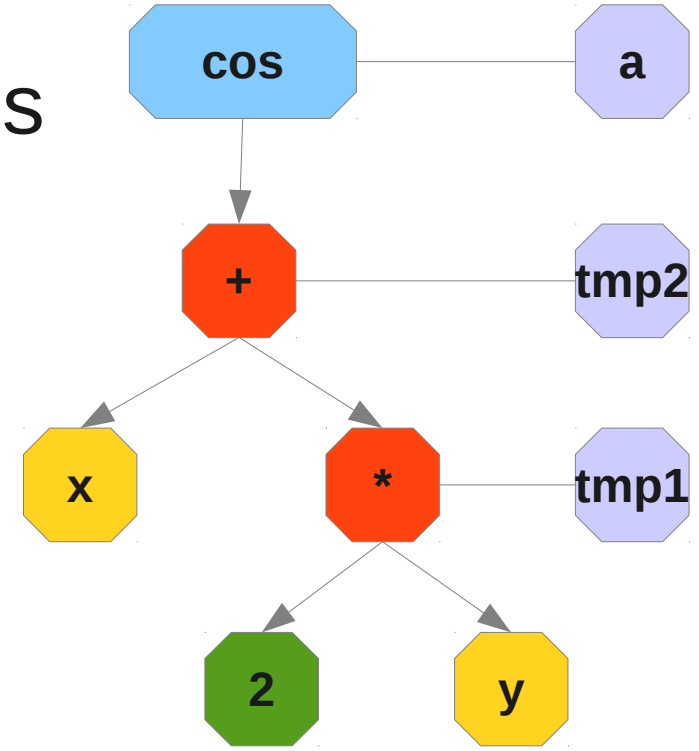
Code 3 adresses

- On doit stocker les résultats intermédiaires des expressions dans une variable temporaire

  a = cos(x+2*y)

  devient

  tmp1 = 2*y
  tmp2 = x + tmp1
  a = cos(tmp2)



- Il faut une variable temporaire par nœud intermédiaire de l'AST

Static Single Assignment

- Code 3 adresses qui permet d'exprimer facilement de nombreuses optimisations

  y = 1
  y = 2
  x = y

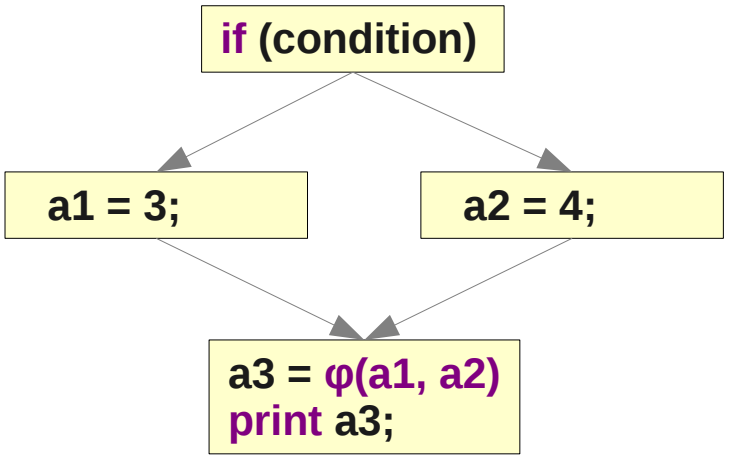
  est transformé en

  y1 = 1
  y2 = 2
  x1 = y2

- Si seul x est utilisé, y1 = 1 peut être supprimé

SSA

- Il existe une opération spéciale φ (phi) qui permet d'indiquer une dépendance entre deux variables
- int a;
  if (condition) { a=3; } else { a=4; }
  print a

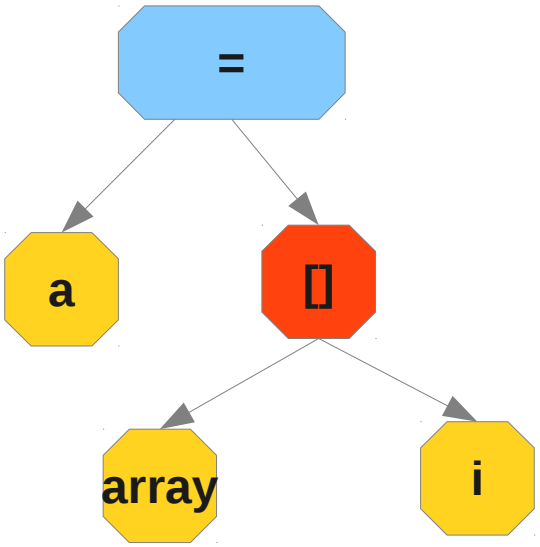


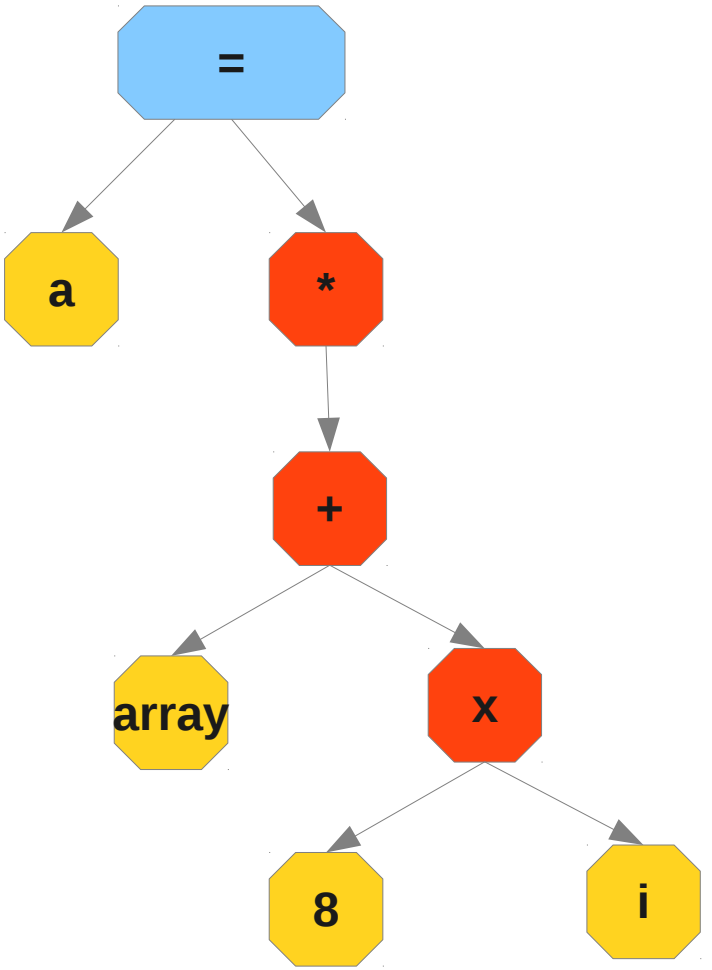
Niveau de langage

- On peut choisir différents niveau de langage intermédiaire
- Exemple :
  - instructions spécifique pour les tableaux
    - On peut tirer partie des procs moderne (MMX, SSE*), ou même générer du bytecode Java
  - Ou transformer en déréférencement de pointeur
    - Le générateur de code est plus simple à écrire car il ne gère qu'une seul abstraction

Exemple

a = array[i]





Structure de contrôle

- Les structures de contrôle sont remplacés par du code basé sur des tests et des goto

- Le if

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

```
if (test) goto test

a = 4;
goto end

test:
a = 3;

end:
print a
```

Structure de contrôle

- Boucle for:

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

```
i = 0

test:
if (i >= 5) goto end

print i;

continue:
i++;

goto test:

end:
print "end";
```

- Comme ici on sait que i est strictement inférieur à 5 on peut faire une petite optimisation

Structure de contrôle

Loop peeling:

On sait que premier test marche toujours

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

```
i = 0

test:
if (i >= 5) goto end

print i;

continue:
i++;

goto test:

end:
print "end";
```

```
i = 0

test:
print i;

continue:
i++;

if (i < 5) goto test

end:
print "end";
```

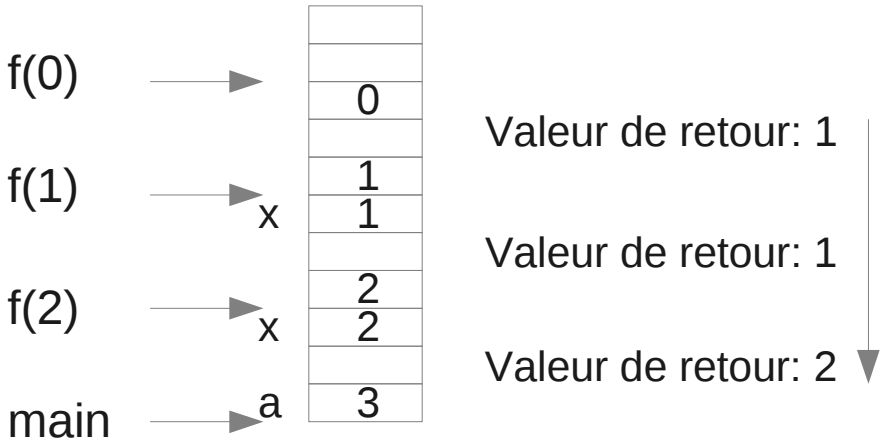
On peut faire mieux en déroulant la boucle

Appel de fonction

- A chaque appel de fonction correspond un morceau de pile appelée stack frame
- Lors d'un appel, l'appelant empile les arguments (+ valeur de retour)
- Au retour soit l'appelant soit l'appelée dépile

```
function f(int x):int {
  return if (x == 0) 1 else x * f(x - 1);
}

function main() {
  int a = 3;
  print f(2);
}
```



Convention d'appel de fonction

- Il existe plein de conventions d'appel
  - Dépend de l'architecture machine
    x86, x64, SPARC, etc
  - De l'OS (Windows, autre)
  - Si l'appel est un appel système (vers le kernel)

- 2 sortes :
  - Caller clean-up (Pascal, Java)
  - Callee clean-up (C)

Différence entre les conventions

- Nombre de paramètre passé en registre
- Convention d'utilisation de registre
  - exemple:
    - ebp doit pointer sur début stack frame
    - esp sur le haut de la pile

- Un même compilateur peut utiliser plusieurs conventions de compilation
  - si on effectue un appel système
  - si on appel à une librairie externe

Le bytecode Java

Assembleur Java

- Le format binaire de la plateforme Java appelé bytecode est un code intermédiaire utilisant une pile
  - Chaque instruction possède un opcode sur un byte
  - Les opérandes sont:
    - soit des entiers
    - soit des constantes partagé dans un dictionnaire (constant pool)

Anatomie d'un .class

- 2 parties
  - Le constant pool
  - Une description de la classe
    - Modificateurs, nom, héritage etc,
    - Description des champs
    - Description des méthodes
      - Un Attribut nommé "Code" qui contient une tableau d'instruction
- On peut mettre des attributs sur la classe, les champs, les méthodes

Format des descripteurs

- Nom d'une classe
  java/lang/Object
- Descripteur de champs
  B byte, C char
  D double, F float
  I integer, J long
  S short, Z boolean

  Ljava/lang/Object; classe java.lang.Object
  [int tableau d'int

Format des descripteurs

- Descripteur de method
  (*descs*)*desc_ou_V*
- Exemple:
  ()V renvoie void
  (I)Z prend un int et renvoie un booléen
  (II)V prend deux ints
  (Ljava/lang/Object;I)V prend Object et un int

Exemple de bytecode

- Exemple de HelloWorld avec javap -c

- public HelloWorld();
  Code:
  0: aload_0
  1: invokespecial #21; //Method **java/lang/Object."<init>":()V**
  4: return

- public static void main(java.lang.String[]);
  Code:
  0: getstatic #14; //Field **java/lang/System.out:Ljava/io/PrintStream;**
  3: ldc #20; //String HelloWorld
  5: invokevirtual #23; //Method **java/io/PrintStream.println:(Ljava/lang/String;)V**
  8: return

Java != bytecode

- En bytecode, le constructeur se nome <init> et <clinit> pour le bloc static
- Les blocks d'initialisations sont recopiés dans les constructeurs
- Les blocs finally sont recopiés dans le cas avec exception et dans le cas sans
- Les appels de méthode indiquent le type de retour
- Il n'y a pas d'exception checked
- Les classes internes n'existe plus. Pour les inners classes un pointeur sur la classe englobante est ajouté et initialisé dans tous les constructeurs (avant l'appel à super ?)
- Les enums sont presque des classes normales
- Il n'y a pas de generics (erasure)

Le constant pool de l'exemple

- Le constant pool est un tableau ou chaque ligne contient un type d'items
  - Class, Fieldref, Methodref, InterfaceMethodref, MethodTyperef, MethodHandleref, NameAndType, Integer, Float, Long, Double, Utf8
  - Un item peut lui-même référencé d'autres items

```
const #1 = NameAndType      #24:#26;//  out:Ljava/io/PrintStream;
const #2 = Asciz      ([Ljava/lang/String;)V;
const #3 = Asciz      java/lang/Object;
const #4 = Asciz      <init>;
const #5 = class      #3;   //  java/lang/Object
const #6 = NameAndType      #4:#9;//  "<init>":()V
const #7 = class      #18; //  java/io/PrintStream
const #8 = Asciz      Helloworld.j;
const #9 = Asciz      ()V;
...
const #21 = Method #5.#6;      //  java/lang/Object."<init>":()V
...
```

Outils

- Assembleur Java
  - Jasmin
- APIs Java de génération de bytecode
  - ASM, BCEL, SERP
- Afficheur de bytecode Java
  - javap -c
  - ASM bytecode outline (plugin eclipse)

.class dans eclipse

Eclipse sait afficher les .class

- public class HelloWorld {
- // Method descriptor #11 ()V
  // Stack: 1, Locals: 1
  public HelloWorld();
  0 aload_0 [this]
  1 invokespecial java.lang.Object() [2]
  4 return
- // Method descriptor #2 ([Ljava/lang/String;)V
  // Stack: 2, Locals: 1
  public static void main(java.lang.String[] arg0);
  0 getstatic java.lang.System.out : java.io.PrintStream [14]
  3 ldc <String "HelloWorld"> [20]
  5 invokevirtual java.io.PrintStream.println(java.lang.String) : void [23]
  8 return
- }

Jasmin

```
.class public HelloWorld
.super java/lang/Object
```

- ```
  ; ceci est un commentaire
  .method public <init>()V
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
  .end method
  ```

- ```
  .method public static main([Ljava/lang/String;)V
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "HelloWorld"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
  .limit stack 2
  .limit locals 1
  .end method
  ```

- Attention au '/' pour accéder au membre ??

Variable locale

- Les paramètres et les variables locales sont stockés au même endroit
  - Les boolean/char/byte/int prenne 1 slot
  - Les doubles et les longs prennent 2 slot
  - Il est possible de réutiliser les slots ?
- Exemple

```
main(String[] args)     slot 0: args
iconst_2
istore 1                slot 1: 2 (int)
iconst_2
i2d
dstore 1                slot 1 et 2: 2.0 (double)
```

Variable locale

- 2 opérations possibles
  - load (iload, lload, fload, dload et aload)
  - store (istore, lstore, fstore, dstore et astore)

- Les opérations sont typés
  - Préfix:

    iload -> int, lload -> long, fload -> float,
    dload -> double et aload -> objet

Opération sur la pile

- Toutes les opérations s'effectue sur la pile
- Chaque case de la pile fait 32 bits
  - Les boolean, byte, short, int, float prennent 1 case
  - les long et double prennent 2 cases

- Exemple:

```
iconst_1      // empile 1
iconst_2      // empile 2
iadd          // dépile 1 et 2 et empile 3
istore 0      // dépile 3 et le stocke
              // dans la varialbe 0
```

Vérificateur

- Pour qu'une classe soit valide pour la machine virtuelle, il faut
  - Que l'on applique les opérations sur les bons types dans la pile et sur les variables locales
  - Que l'on indique le nombre maximal de variables locales, la taille maximal de la pile
- Sinon le vérificateur de bytecode de la machine virtuelle ne permettra pas le chargement de la classe
  - On verra plus tard les algos utilisés

Max stacks|max locals

- Pour calculer les tailles max, il faut simuler une execution abstraite

- void max(int);
  Code:
  0: iload_1
  1: **ifne 13** // (!=0)
  4: iload_1
  5: i2d
  6: dstore_3
  7: dload_3
  8: d2i
  9: istore_2
  10:**goto 15**
  13:iconst_2
  14:istore_2
  15:iload_2
  16:istore_3
  17:**return**

- Résultat: Stack=2, Locals=5

Les constantes

- Il y a plusieurs façon de charger une constante
  - iconst_0, iconst_1 ... iconst_5, iconst_m1
  - bipush [-128, +255]
  - sipush [-37728, +37727]
  - ldc/ldc2, pour toute les constantes pas seulement les integers
- Les booléens sont gérés comme des ints

Manipulation de la pile

- Il existe des instructions pour dupliquer le sommet de pile
- dup
  - duplique 1 valeur
- dup2
  - duplique 2 valeurs ou 1 seul long/double

- Il y a plein d'autres dup_...
- Les dups sont polymorphes

Opérations

- Opération numérique sont typés (préfix i,l,f,d)
  - iadd, isub, imul, idiv, irem
- Opération sur les ints
  - iand (&), ior (|), ixor(^), ineg(~), ishl(<<), ishr(>>), iushr(>>>)
- i++, ou i+=2 existe !
  - iinc 3 2 (ajoute 2 à la variable local 3)

Conversions

- Comme les opérations et les variables sont typés, il faut faire des conversions

- i2b, i2c, i2s, i2d, i2f, i2l,
  l2i, l2f, l2d,
  f2i, f2l, f2d,
  d2i, d2l, d2f

- On ne peut faire les conversions vers byte/char/short qu'à partir d'un int

Branchements

- Inconditionnel
  - goto index d'arrivé
- Conditionnel
  - ifeq, ifne, iflt, ifle, ifgt, ifge (compare à zéro)
  - if_acmp[eq|ne] (comparaison de refs)
  - if_icmp[eq|ne|lt|le|gt|ge] (comparaison d'int)
  - ifnonnull, ifnull (test à null)
- Sous-routine
  - jsr, ret (plus utilisé depuis là 1.5 en Java)
    - Le vérificateur aime pas !

Test long/float/double

- Le test se fait en deux fois, on test les long/float/double, cela renvoie un int puis on test avec ifeq/ifne...
- Test:
  - lcmp/fcmpg/dcmpg (comparaison, -1, 0, 1)
  - Existe aussi fcmpl/dcmpl (NaN pas traité pareil)
- Exemple:

```
0:   ldc2_w     #16; //double 2.0d
3:   dstore_2
4:   dload_2
5:   ldc2_w     #18; //double 4.0d
8:   dcmpg
9:   ifgt   17
```

Allocation d'un objet

- new Integer(2) est séparé en deux opérations
  - new
  - invokespecial <init> pour appeler les constructeurs
- Si on oublie d'appeler le constructeur le vérificateur plante !
- Exemple:

```
0:   new  #16; //class java/lang/Integer
3:   dup
4:   iconst_2
5:   invokespecial   #18; //Method java/lang/Integer."<init>":(I)V
8:   astore_1
```

Checkcast et instanceof

- Checkcast permet de faire un cast
  - Exemple

```
aload_1
checkcast#16; //class java/lang/Integer
```

- Instanceof fait le test instanceof
  - Exemple

```
aload_1
instanceof     #16; //class java/lang/Integer
```

Création de tableaux

- Tableau de type primitif
  - Newarray

```
iconst_5
newarray int
```

- Tableau d'objet
  - Anewarray

```
iconst_5
anewarray       #21; //class "[I"
```

Rappel les tableaux int[][] sont des tableaux d'objets

Accès aux valeurs des tableaux

- Charge l'élément d'un tableau
  - baload (byte et boolean), caload, iaload, lalod, faload, daload, aaload
  - bastore (byte et boolean), castore, iastore, lastore, fastore, dastore, aastore
- Exemple:

```
iconst_5
newarray int
astore_1
aload_1
iconst_3        // numéro de la case
iconst_5
iastore
aload_1
iconst_2        // numéro de la case
iaload
istore_2
```

Accès aux champs

- getfield/putfield pour les champs d'instance
- getstatic/putstatic pour les champs static

- Exemple:

```
public static void main(java.lang.String[] arg0);
   0  getstatic java.lang.System.out : java.io.PrintStream [14]
   3  ldc <String "HelloWorld"> [20]
   5  invokevirtual java.io.PrintStream.println(java.lang.String) : void [23]
   8  return
```

appels de méthodes

- Il existe 5 appels de méthodes:
  - invokespecial (constructeur et méthode privé de la classe courante)
  - invokesuper (super.)
  - invoke static (appel statique)
  - invokevirtual (appel polymorphe)
  - Invokedynamic (JSR 292)
- On doit avoir dans la pile le receiver (sauf pour invokestatic/invokedynamic) et les arguments

switch

- Il existe deux implantations des switchs
  - Tableswitch (table de branchement)
  - Lookup switch (dichotomie sur les branchements)

- Si les valeurs sont continues

  => tableswitch

lookupswitch

- Avec Jasmin:

```
lookupswitch
    3 : label1
    7 : label2
    default : defLabel

label1:
    ; case 3
    goto break

label2:
    ; case 7
    goto break;

defLabel:
    ; default

break:
```

tableswitch

- Avec Jasmin:

```
tableswitch 0
    label1
    label2
    default : defLabel

label1:
    ; case 0
    goto break

label2:
    ; case 1
    goto break;

defLabel:
    ; default

break:
```

Exceptions

- Les exceptions sont gérer dans une table stockée à la suite du bytecode
- On indique pour un type d'exception levée entre deux offsets d'instructions là où il faut sauter
- Le sommet de la pile du handler d'exception contient l'exception levée
- Avec Jasmin:
  .catch <classname> from <label1> to <label2> using <label3>

Exceptions

- public static void main(java.lang.String[]);
  Code:
  ```
     0: aload_0
     1: iconst_0
     2: aaload
     3: invokestatic  #16   // Method java/lang/Integer.parseInt:(Ljava/lang/String;)I
     6: pop
     7: goto 18
    10: astore_1
    11: getstatic     #22     // Field java/lang/System.out:Ljava/io/PrintStream;
    14: aload_1
    15: invokevirtual #28   // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
    18: return
  ```
- Exception table:
  ```
     from    to  target type
         0     7    10      Class java/lang/NumberFormatException
  ```

Finally

- public static void main(java.lang.String[]);
  Code:
  ```
     0: aload_0
     1: iconst_0
     2: aaload
     3: invokestatic  #16 // Method java/lang/Integer.parseInt:(Ljava/lang/String;)I
     6: pop
     7: goto 21
    10: astore_1
    11: getstatic     #22     // Field java/lang/System.out:Ljava/io/PrintStream;
    14: ldc           #28     // String finally
    16: invokevirtual #30 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    19: aload_1
    20: athrow
    21: getstatic     #22     // Field java/lang/System.out:Ljava/io/PrintStream;
    24: ldc           #28     // String finally
    26: invokevirtual #30  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    29: return
  ```
- Exception table:
  ```
     from    to  target type
         0    10    10   any
  ```