

# CS 239 Programming Languages

---

1. The Java Compiler Compiler	2
2. The Visitor Pattern	7
3. The Java Tree Builder	19
4. Scheme	27
5. Interpreters	70
6. Operational Semantics	116
7. Continuation Passing Style	137
8. Flow Analysis	205
9. Type-Safe Method Inlining	231

# Chapter 1: The Java Compiler Compiler

---

# The Java Compiler Compiler (JavaCC)

---

- Can be thought of as “Lex and Yacc for Java.”
- It is based on LL( $k$ ) rather than LALR(1).
- Grammars are written in EBNF.
- The Java Compiler Compiler transforms an EBNF grammar into an LL( $k$ ) parser.
- The JavaCC grammar can have embedded action code written in Java, just like a Yacc grammar can have embedded action code written in C.
- The lookahead can be changed by writing `LOOKAHEAD(...)`.
- The whole input is given in just one file (not two).

## The JavaCC input format

---

One file:

- header
- token specifications for lexical analysis
- grammar

# The JavaCC input format

---

Example of a token specification:

```
TOKEN :  
{  
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >  
}
```

Example of a production:

```
void StatementListReturn() :  
{  
{  
  ( Statement() )* "return" Expression() ";"  
}
```

## Generating a parser with JavaCC

---

```
javacc fortran.jj // generates a parser with a specified name
javac Main.java // Main.java contains a call of the parser
java Main < prog.f // parses the program prog.f
```

## Chapter 2: The Visitor Pattern

---

# The Visitor Pattern

---

For **object-oriented programming**,

the Visitor pattern **enables**

the definition of a **new operation**

on an **object structure**

**without changing the classes**

of the objects.

Gamma, Helm, Johnson, Vlissides:

**Design Patterns**, 1995.

## Sneak Preview

---

When using the **Visitor** pattern,

- the set of classes must be fixed in advance, and
- each class must have an accept method.

## First Approach: Instanceof and Type Casts

---

The running Java example: summing an integer list.

```
interface List {}

class Nil implements List {}

class Cons implements List {
    int head;
    List tail;
}
```

## First Approach: Instanceof and Type Casts

---

```
List l;      // The List-object
int sum = 0;
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil)
        proceed = false;
    else if (l instanceof Cons) {
        sum = sum + ((Cons) l).head;
        l = ((Cons) l).tail;
        // Notice the two type casts!
    }
}
```

**Advantage:** The code is written without touching the classes `Nil` and `Cons`.

**Drawback:** The code constantly uses type casts and `instanceof` to determine what class of object it is considering.

## Second Approach: Dedicated Methods

---

The first approach is **not** object-oriented!

To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects.

```
interface List {  
    int sum();  
}
```

We can now compute the sum of all components of a given `List`-object `l` by writing `l.sum()`.

## Second Approach: Dedicated Methods

---

```
class Nil implements List {
    public int sum() {
        return 0;
    }
}
```

```
class Cons implements List {
    int head;
    List tail;
    public int sum() {
        return head + tail.sum();
    }
}
```

**Advantage:** The type casts and instanceof operations have disappeared, and the code can be written in a systematic way.

**Disadvantage:** For each new operation on List-objects, new dedicated methods have to be written, and all classes must be recompiled.

## Third Approach: The Visitor Pattern

---

### The Idea:

- Divide the code into an object structure and a Visitor (akin to Functional Programming!)
- Insert an `accept` method in each class. Each `accept` method takes a Visitor as argument.
- A Visitor contains a `visit` method for each class (overloading!) A method for a class `C` takes an argument of type `C`.

```
interface List {  
    void accept(Visitor v);  
}
```

```
interface Visitor {  
    void visit(Nil x);  
    void visit(Cons x);  
}
```

## Third Approach: The Visitor Pattern

---

- The purpose of the `accept` methods is to invoke the `visit` method in the `Visitor` which can handle the current object.

```
class Nil implements List {
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

## Third Approach: The Visitor Pattern

---

- The control flow goes back and forth between the `visit` methods in the Visitor and the `accept` methods in the object structure.

```
class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil x) {}
    public void visit(Cons x) {
        sum = sum + x.head;
        x.tail.accept(this);
    }
}

.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

**Notice:** The `visit` methods describe both  
1) actions, and 2) access of subobjects.

# Comparison

---

The Visitor pattern combines the advantages of the two other approaches.

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

**The advantage of Visitors:** New methods without recompilation!

**Requirement for using Visitors:** All classes must have an accept method.

**Tools that use the Visitor pattern:**

- JJTree (from Sun Microsystems) and the Java Tree Builder (from Purdue University), both frontends for The Java Compiler Compiler from Sun Microsystems.

## Visitors: Summary

---

- **Visitor makes adding new operations easy.** Simply write a new visitor.
- **A visitor gathers related operations.** It also separates unrelated ones.
- **Adding new classes to the object structure is hard.** Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most likely to change the classes of objects that make up the structure.
- **Visitors can accumulate state.**
- **Visitor can break encapsulation.** Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

## Chapter 3: The Java Tree Builder

---

## The Java Tree Builder (JTB)

---

The Java Tree Builder (JTB) has been developed at Purdue in my group.

JTB is a frontend for The Java Compiler Compiler.

JTB supports the building of syntax trees which can be traversed using visitors.

JTB transforms a bare JavaCC grammar into three components:

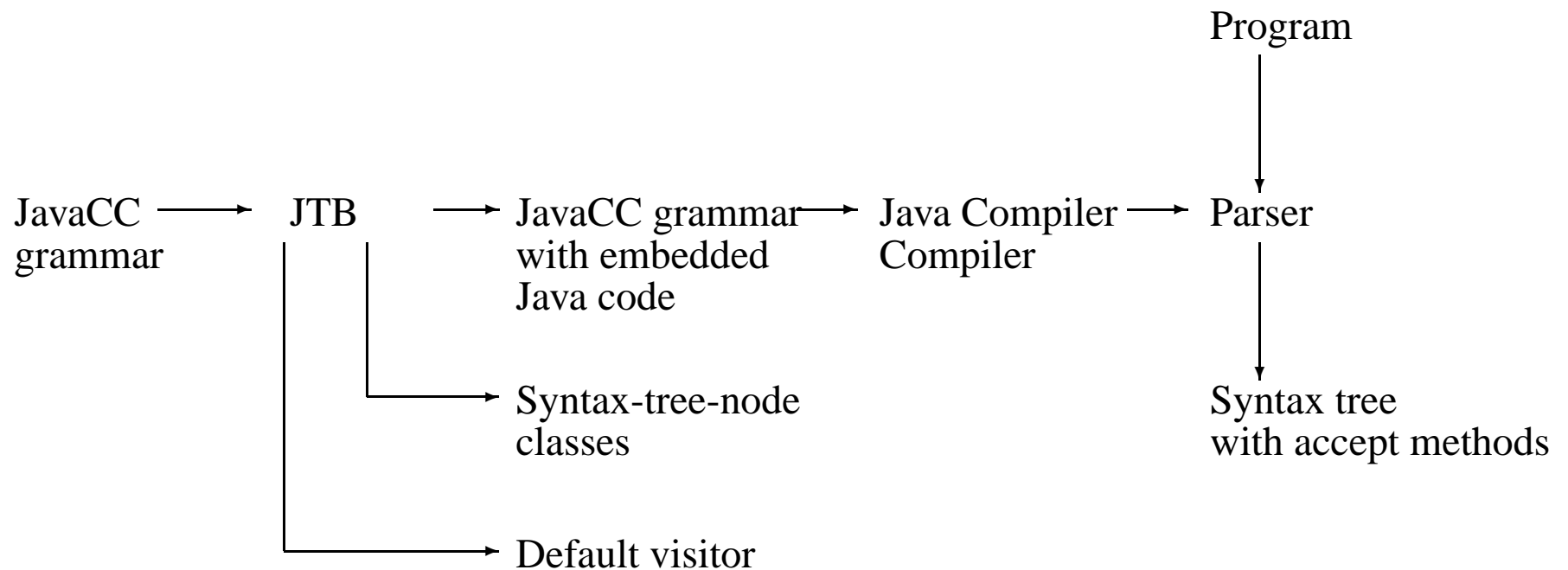
- a JavaCC grammar with embedded Java code for building a syntax tree;
- one class for every form of syntax tree node; and
- a default visitor which can do a depth-first traversal of a syntax tree.

## The Java Tree Builder

---

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.

The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



## Using JTB

---

```
jtb fortran.jj // generates jtb.out.jj
javacc jtb.out.jj // generates a parser with a specified name
javac Main.java // Main.java contains a call of the parser
                and calls to visitors
java Main < prog.f // builds a syntax tree for prog.f, and
                  executes the visitors
```

## Example (simplified)

---

For example, consider the Java 1.1 production

```
void Assignment() : {  
    { PrimaryExpression() AssignmentOperator()  
      Expression() }  
}
```

JTB produces:

```
Assignment Assignment () :  
{ PrimaryExpression n0;  
  AssignmentOperator n1;  
  Expression n2; {} }  
{ n0=PrimaryExpression()  
  n1=AssignmentOperator()  
  n2=Expression()  
  { return new Assignment(n0,n1,n2); }  
}
```

Notice that the production returns a syntax tree represented as an Assignment object.

## Example (simplified)

---

JTB produces a syntax-tree-node class for Assignment:

```
public class Assignment implements Node {
    PrimaryExpression f0; AssignmentOperator f1;
    Expression f2;

    public Assignment(PrimaryExpression n0,
                     AssignmentOperator n1,
                     Expression n2)
    { f0 = n0; f1 = n1; f2 = n2; }

    public void accept(visitor.Visitor v) {
        v.visit(this);
    }
}
```

Notice the `accept` method; it invokes the method `visit` for `Assignment` in the default visitor.

## Example (simplified)

---

The default visitor looks like this:

```
public class DepthFirstVisitor implements Visitor {
    ...
    //
    // f0 -> PrimaryExpression()
    // f1 -> AssignmentOperator()
    // f2 -> Expression()
    //
    public void visit(Assignment n) {
        n.f0.accept(this);
        n.f1.accept(this);
        n.f2.accept(this);
    }
}
```

Notice the body of the method which visits each of the three subtrees of the Assignment node.

## Example (simplified)

---

Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```
public class VprintAssignRHS extends DepthFirstVisitor {
    void visit(Assignment n) {
        VPrettyPrinter v = new VPrettyPrinter();
        n.f2.accept(v); v.out.println();
        n.f2.accept(this);
    }
}
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visit` in `VprintAssignRHS` is executed.

Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.

JTB is bootstrapped.

## Chapter 4: Scheme

---

# The Scheme Language

---

Interaction loop.

```
3
```

```
(+ 1 3)
```

```
(a b c)
```

```
'(a b c)
```

```
(define x 3)
```

```
x
```

```
(+ x 1)
```

```
(define l (a b c))
```

```
(define l '(a b c))
```

```
(define u (+ x 1))
```

# Procedures

---

Creating procedures with lambda:

```
(lambda (x) body)
(lambda (x) (+ x 1))
((lambda (x) (+ x 1)) 4)
(define mysucc (lambda (x) (+ x 1)))
(mysucc 4)
(define myplus (lambda (x y) (+ x y)))
(+ 3 4)
((lambda (x y) (+ x y)) 3 4)
(myplus 3 4)
```

Procedures can take other procedures as arguments:

```
((lambda (f x) (f x 3))
 myplus 5)
```

## Procedures

---

Procedures can return other procedures; this is called Currying:

```
(define twice  
  (lambda (f)  
    (lambda (x)  
      (f (f x))))))
```

```
(define add2 (twice (lambda (x) (+ x 1))))
```

# Kinds of Data

---

**Basic values** = Symbols  $\cup$  Numbers  $\cup$  Strings  $\cup$  Lists

**Symbols:** sequence of letters and digits starting with a letter.

The sequence can also include other symbols, such as -, \$, =, \*, /, ?, \_.

**Numbers:** integers, etc.

**Strings:** "this is a string"

**Lists:**

(1) the empty list is a list ()

(2) a sequence  $(s_0 \dots s_{n-1})$  where each  $s_i$  is a value (either a symbol, number, string, or list)

(3) nothing is a list unless it can be shown to be a list by rules (1) and (2).

This is an inductive definition, which will play an important part in our reasoning. We will often solve problems (e.g., write procedures on lists) by following this inductive definition. Examples of lists:

(turkey)

(atom)

((turkey) (xyz atom))

xyz -- a symbol, not a list

(xyz)

# List Processing

---

Putting list literals in programs: `'(a b c)` `'()`

Basic operations on lists:

`car`: if `l` is  $(s_0 \dots s_{n-1})$ , then `(car l)` is  $s_0$ . The `car` of the empty list is undefined.

e.g. `(define l '(a b c))` `(car l) => a`

`cdr`: if `l` is  $(s_0 s_1 \dots s_{n-1})$ , then `(cdr l)` is  $(s_1 \dots s_{n-1})$ . The `cdr` of the empty list is undefined.

`(cdr l) => (b c)`

Combining `car` and `cdr`:

`(car (cdr l))`

`(cdr (cdr l))`

etc.

## Building Lists

---

cons: if  $s$  is the value  $s$ , and  $l$  is the list  $(s_0 \dots s_{n-1})$ , then  $(\text{cons } s \ l)$  is the list  $(s \ s_0 \dots s_{n-1})$

Cons builds a list whose car is  $s$  and whose cdr is  $l$ .

```
(car (cons s l)) = s
```

```
(cdr (cons s l)) = l
```

```
cons : value * list -> list
```

```
car   : list -> value
```

```
cdr   : list -> list
```

# Booleans

---

Literals:

#t, #f

Predicates:

(number? s)

(symbol? s)

(string? s)

(null? s)

(pair? s)

(eq? s1 s2)           --works on symbols

(= n1 n2)            --works on numbers

(zero? n)

(> n1 n2)

etc

Conditional:

(if bool e1 e2)

# Recursive Procedures

---

Imagine we want to define a procedure to find powers, eg.

$$e(n, x) = x^n .$$

It is easy to define a bunch of procedures  $e_0(x) = x^0$ ,  $e_1(x) = x^1$ ,  $e_2(x) = x^2$ :

$$e_0(x) = 1$$

$$e_1(x) = x \times e_0(x)$$

$$e_2(x) = x \times e_1(x)$$

$$e_3(x) = x \times e_2(x)$$

and in general, we can define

$$e_n(x) = x \times e_{n-1}(x)$$

How did we do this? At each stage, we used the fact that we had already solved the problem for smaller  $n$ . This is the principle of mathematical induction.

## Recursive Procedures

---

How can we parameterize this construction to get a single procedure  $e$ ?

If  $n$  is 0, we know the answer:  $(e\ 0\ x) = 1$ .

If  $n$  is greater than 0, assume we can solve the problem for  $n - 1$ .

Then the answer is  $(e\ n\ x) = (*\ x\ (e\ (-\ n\ 1)\ x))$ .

# Recursive Procedures

---

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x))))))
```

Why does this work? Let's prove it works for any  $n$ , by induction on  $n$ :

(1) It surely works for  $n=0$ .

(2) Now assume (for the moment) that it works when  $n = k$ . Then it works when  $n = k + 1$ . Why? Because then  $(e\ n\ x) = (*\ x\ (e\ k\ x))$ , and we know  $e$  works when its first argument is  $k$ . So it gives the right answer when its first argument is  $k + 1$ .

# Structural Induction

---

- **The Moral:** If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem.
- Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg  $n=0$ , the basis step), and then it will terminate successfully.
- **Principle of structural induction:** If you always recur on smaller problems, then your procedure is sure to work.

$$\begin{aligned} & (e\ 3\ 2) \\ &= (*\ 2\ (e\ 2\ 2)) \\ &= (*\ 2\ (*\ 2\ (e\ 1\ 2))) \\ &= (*\ 2\ (*\ 2\ (*\ 2\ (e\ 0\ 2)))) \\ &= (*\ 2\ (*\ 2\ (*\ 2\ 1))) \\ &= 8 \end{aligned}$$

# Recursive Procedures

---

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

```
(fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

(1) This is the natural recursive definition of factorial; (2) each call of fact is made with a promise that the value returned will be multiplied by the value of  $n$  at the time of the call; and (3) thus fact is invoked in larger and larger control contexts as the calculation proceeds.

## Recursive Procedures

---

### Java

```
int fact(int n) {  
    int a=1;  
    while (n!=0) {  
        a=n*a;  
        n=n-1;  
    }  
    return a;  
}
```

### Scheme

```
(define fact-iter  
  (lambda (n)  
    (fact-iter-acc n 1)))  
  
(define fact-iter-acc  
  (lambda (n a)  
    (if (zero? n)  
        a  
        (fact-iter-acc (- n 1) (* n a)))))
```

# Recursive Procedures

---

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

`fact-iter-acc` is always invoked in the same context (in this case, no context at all).

When `fact-iter-acc` calls itself, it does so at the “tail end” of a call to `fact-iter-acc`.

That is, no promise is made to do anything with the returned value other than return it as the result of the call to `fact-iter-acc`.

Thus each step in the derivation above has the form

```
(fact-iter-acc n a).
```

## Recursive Procedures

---

**Procedure:** `nth-elt`

**Input:** a list  $l$  and an integer  $k$ .

**Output:** Returns the  $k$ -th element of list  $l$ , (counting from 0). If  $l$  does not contain sufficiently many elements, an error is signaled.

When  $k$  is 0, we know the answer: `(car l)`. If  $k$  is greater than 0, then the answer is the  $k - 1$ st element of `(cdr l)`. The only glitch is that we have to *guard* the `car` and `cdr` operations:

```
(define nth-elt
  (lambda (l k)
    (if (null? l)
        (error nth-elt "nth-elt:  ran off end")
        (if (zero? k)
            (car l)
            (nth-elt (cdr l) (- k 1))))))
```

## Example Run

---

```
(define nth-elt
  (lambda (l k)
    (if (null? l)
        (error nth-elt "nth-elt: ran off end")
        (if (zero? k)
            (car l)
            (nth-elt (cdr l) (- k 1)))))))
```

```
(nth-elt '(a b c d e) 3)
= (nth-elt '(b c d e) 2)
= (nth-elt '(c d e) 1)
= (nth-elt '(d e) 0)
= d
```

## Recursive Procedures

---

**Procedure:** `list-of-symbols?`

**Input:** a list *l*

**Output:** true if *l* consists entirely of symbols, false if it contains a non-symbol.

```
(define list-of-symbols?  
  (lambda (l)  
    (if (null? l)  
        #t  
        (if (symbol? (car l))  
            (list-of-symbols? (cdr l))  
            #f))))
```

## Recursive Procedures

---

**Procedure:** `member?`

**Input:** a symbol  $i$  and a list  $l$  of atoms.

**Output:** true if  $l$  contains the symbol  $i$  as one of its members, false otherwise.

```
(define member?  
  (lambda (i l)  
    (if (null? l) #f  
        (if (eqv? i (car l)) #t  
            (member? i (cdr l))))))
```

## Recursive Procedures

---

**Procedure:** `remove-first`

**Input:** a symbol  $i$  and a list  $l$  of atoms.

**Output:** A list the same as  $l$ , except that the first occurrence of  $i$  has been removed. If there is no occurrence of  $i$ , the list will be unchanged.

```
(define remove-first
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (cdr l)
            (cons (car l)
                  (remove-first i (cdr l)))))))
```

## Recursive Procedures

---

**Procedure:** `remove`

**Input:** a symbol  $i$  and a list  $l$  of atoms.

**Output:** A list the same as  $l$ , except that *all* occurrences of  $i$  have been removed. If there is no occurrence of  $i$ , the list will be unchanged.

```
(define remove
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (remove i (cdr l))
            (cons (car l)
                  (remove i (cdr l)))))))
```

# Recursive Procedures

---

**Procedure:** `remove-all`

**Input:** An symbol  $i$  and a list  $l$

**Output:** Like `remove`, except that all occurrences of  $i$  in  $l$  or any of its sublists (at any level) are to be removed.

```
(define remove-all
  (lambda (i l)
    (if (null? l) '()
        (if (atom? (car l))
            (if (eqv? i (car l))
                (remove-all i (cdr l))
                (cons (car l) (remove-all i (cdr l))))
            (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

## cond

---

In Scheme, a many-way branch is expressed using `cond`.

```
(cond
  (test1 exp1)
  (test2 exp2)
  ...
  (else exp_n))
```

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (if (eqv? i (car l))
           (remove-all i (cdr l))
           (cons (car l) (remove-all i (cdr l)))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

## let

---

When we need *local names*, we use the special form `let`:

```
(let ((var1 val1)
      (var2 val2)
      ...))
  exp)
```

# let

---

For example, we write

```
(let ((n 25))  
  (let ((x (sqrt n))  
        (y 3))  
    (+ x y)))
```

```
= (let ((x (sqrt 25))  
       (y 3))  
   (+ x y))
```

```
= (let ((x 5)  
       (y 3))  
   (+ x y))
```

```
= (+ 5 3)
```

```
= 8
```

# let

---

```
(let ((x 3))  
  (let ((y (+ x 4)))  
    (* x y)))
```

```
= (let ((y (+ 3 4)))  
    (* 3 y))
```

```
= (let ((y 7))  
    (* 3 y))
```

```
= (* 3 7) = 21
```

compare to:

```
(let ((x 3)  
      (y (+ x 4)))  
  (* x y))
```

Here the occurrence of `x` on the second line refers to whatever the enclosing value of `x` happens to be— maybe set by another `let`.

## let

---

So we can rewrite `remove-all` as

```
(define remove-all
  (lambda (i l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (let ((remember-cdr (remove-all i (cdr l))))
         (if (eqv? i (car l))
             remember-cdr
             (cons (car l) remember-cdr))))
      (else (cons (remove-all i (car l))
                  (remove-all i (cdr l)))))))
```

Let's can, of course, be nested.

# let

---

Using `let`, we can give names to local procedures and pass them around:

```
(let ((f (lambda (y z) (* y (+ y z))))
      (g (lambda (u) (+ u 5))))
  (f (g 3) 17))
```

```
(let ((x 5))
  (let ((f (lambda (y z) (* y (+ x z))))
        (g (lambda (u) (+ u x)))
        (x 38))
    (f (g 3) 17)))
```

```
(let ((f (lambda (y) (y (y 5))))
      (g (lambda (z) (+ z 8))))
  (f g))
```

```
(let ((f (lambda (y z) (y (y z))))
      (g (lambda (z) (+ z 8))))
  (f g 5))
```

## Local Recursive Procedures

---

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))
```

```
(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1)
                                    (* n a)))))
```

One'd like to write the program with a local recursive procedure:

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
          (lambda (n a)
            (if (zero? n)
                a
                (fact-iter-acc (- n 1) (* n a)))))
      (fact-iter-acc n 1))))
```

## Local Recursive Procedures

---

The scope of `fact-iter-acc` doesn't include its definition. Instead, we can use `letrec`:

```
(letrec
  ((name1 proc1)
   (name2 proc2)
   ...))
body)
```

`letrec` creates a set of mutually recursive procedures and makes their names available in the body.

## Local Recursive Procedures

---

So we can write:

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
              (lambda (n a)
                (if (zero? n) a
                    (fact-iter-acc (- n 1)
                                    (* n a))))))
      (fact-iter-acc n 1))))
```

## Local Recursive Procedures

---

```
(define remove
  (lambda (i l)
    (if (null? l) '()
        (if (eq? i (car l))
            (remove i (cdr l))
            (cons (car l) (remove i (cdr l)))))))
```

can be replaced by

```
(define remove
  (lambda (i l)
    (letrec
      ((loop
        ;; (loop l) = (remove i l)
        (lambda (l)
          (if (null? l) '()
              (if (eq? i (car l))
                  (loop (cdr l))
                  (cons (car l) (loop (cdr l)))))))
      (loop l))))
```

## Local Recursive Procedures

---

Another example:

```
(define linear-search
  (lambda (pred list fn)
    (letrec
      ((loop
        ;; (loop list) =
        ;; (linear-search pred list fn)
        (lambda (list)
          (cond
            ((null? list) #f)
            ((pred (car list)) (fn (car list)))
            (else (loop (cdr list)))))))
      (loop list))))
```

# Sequencing

---

Want to have some finer control over the order in which things are done in Scheme. We need this for worrying about (a) side-effects and (b) termination. In general, there is only one rule about sequencing in Scheme:

**Arguments are evaluated before procedure bodies.**

So in `((lambda (x y z) body) exp1 exp2 exp3)`

`exp1`, `exp2`, and `exp3` are guaranteed to be evaluated before `body`, but we don't know in what order `exp1`, `exp2`, and `exp3` are going to be evaluated, but they will all be evaluated before `body`.

This is **precisely** the same as

`(let ((x exp1) (y exp2) (z exp3)) body)`

In both cases, we evaluate `exp1`, `exp2`, and `exp3`, and then we evaluate `body` in an environment in which `x`, `y`, and `z` are bound to the values of `exp1`, `exp2`, and `exp3`.

# Data Types and their Representations in Scheme

---

1. Want to define new data types. This comes in two pieces: a *specification* and an *implementation*. The specification tells us *what* data (and what operations on that data) we are trying to represent. The implementation tells us *how* we do it.
2. We want to arrange things so that you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server).
3. Both the specification and implementation have to deal with two things: the data and the operations on the data.
4. Vital part of the implementation is the specification of how the data is represented. We will use the notation  $[v]$  for “the representation of data  $v$ ”.

# Numbers

---

**Data Specification:** the non-negative integers.

**Specification of Operations:**

$$\begin{aligned} \text{zero} &= [0] \\ (\text{is-zero? } [n] ) &= \begin{cases} \text{\#t} & n = 0 \\ \text{\#f} & n \neq 0 \end{cases} \\ (\text{succ } [n] ) &= [n + 1] \\ (\text{pred } [n + 1] ) &= [n] \end{aligned}$$

Now we can write procedures to do the other arithmetic operations; these will work no matter what representation of the natural numbers we use.

```
(define plus
  (lambda (x y)
    (if (is-zero? x) y
        (succ (plus (pred x) y)))))
```

will satisfy  $(\text{plus } [x] [y] ) = [x + y]$ , no matter what implementation of the integers we use.

# Unary Representation of Numbers

---

$$\begin{aligned} [0] &= () \\ [n+1] &= (\text{cons } \#t [n]) \end{aligned}$$

So the integer  $n$  is represented by a list of  $n$   $\#t$ 's.

It's easy to see that we can satisfy the specification by writing:

```
(define zero '())
```

```
(define is-zero? null?)
```

```
(define succ  
  (lambda (n) (cons #t n)))
```

```
(define pred cdr)
```

## Scheme Representation of Numbers

---

$[n]$  = the Scheme integer  $n$

```
(define zero 0)
```

```
(define is-zero? zero?)
```

```
(define succ (lambda (n) (+ n 1)))
```

```
(define prec (lambda (n) (- n 1)))
```

# Finite Functions

---

**Data Specification:** a function whose domain is a finite set of Scheme symbols, and whose range is unspecified:  $\{(s_1, v_1), \dots, (s_n, v_n)\}$ .

**Specification of Operations:**

```
;;; interface finite-function =  
;;; type ff  
;;; empty-ff : ff  
;;; apply-ff : ff * key -> value  
;;; extend-ff : key * val * ff -> ff
```

```
empty-ff                = [0]  
( apply-ff [f] s )     = f(s)  
( extend-ff s v [f] )  = [g],  
                        where  $g(s') = \begin{cases} v & s' = s \\ f(s') & \text{otherwise} \end{cases}$ 
```

This tells us what procedures are available for dealing with finite functions: it defines the *interface*. The interface gives the type of each procedure and a description of the intended behavior of each procedure.

# Procedural Representation

---

;;; uses ff = key -> value

$f = [\{(s_1, v_1), \dots, (s_n, v_n)\}]$  iff  $(f\ s_i) = v_i$ .

Implement the operations by:

```
(define apply-ff
  (lambda (ff z) (ff z)))
```

```
(define empty-ff
  (lambda (z)
    (error 'env-lookup
           (format "couldn't find ~s" z))))
```

```
(define extend-ff
  (lambda (key val ff)
    (lambda (z)
      (if (eq? z key)
          val
          (apply-ff ff z)))))
```

# Procedural Representation

---

;; Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
```

```
> (define ff-2 (extend-ff 'b 2 ff-1))
```

```
> (define ff-3 (extend-ff 'c 3 ff-2))
```

```
> (define ff-4 (extend-ff 'd 4 ff-3))
```

```
> (define ff-5 (extend-ff 'e 5 ff-4))
```

```
> ff-5
```

```
<Procedure>
```

```
> (apply-ff ff-5 'd)
```

```
4
```

```
> (apply-ff empty-ff 'c)
```

```
error in env-lookup: couldn't find c.
```

```
> (apply-ff ff-3 'd)
```

```
error in env-lookup: couldn't find d.
```

# Association-list Representation

---

$[\{(s_1, v_1), \dots, (s_n, v_n)\}] = ((s_1 \ . \ v_1) \ \dots \ (s_n \ . \ v_n))$

```
(define empty-ff '())
```

```
(define extend-ff  
  (lambda (key val ff)  
    (cons (cons key val) ff)))
```

```
(define apply-ff  
  (lambda (alist z)  
    (if (null? alist)  
        (error 'env-lookup  
              (format "couldn't find ~s" z))  
        (let ((key (caar alist))  
              (val (cdar alist))  
              (ff (cdr alist)))  
          (if (eq? z key) val (apply-ff ff z))))))
```

## Association-list Representation

---

;; Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
```

```
> (define ff-2 (extend-ff 'b 2 ff-1))
```

```
> (define ff-3 (extend-ff 'c 3 ff-2))
```

```
> (define ff-4 (extend-ff 'd 4 ff-3))
```

```
> ff-4
```

```
((d . 4) (c . 3) (b . 2) (a . 1))
```

## Chapter 5: Interpreters

---

# Interpreters

---

We will write interpreters using four ideas:

- Environments
- Cells
- Closures
- Recursive Environments

We will begin with a simple example that does not require any of these ideas: an interpreter for a stack machine.

We will then study interpreters for five subsets of Scheme, and we will conclude with an interpreter for Flyweight Java.

All of the interpreters will be written in Scheme.

# Stack Machine Code Grammar

---

Let us write an interpreter for a stack machine. The machine will have two components: an *action* and a *stack*.

The stack contains the data in the machine. We will represent the stack as a list of Scheme values, with the top of the stack at the front of the list.

The action represents the instruction stream being executed by the machine. Actions are defined by the grammar:

```
Action ::= halt
          | incr; Action
          | add; Action
          | push Integer ; Action
          | pop; Action
```

We will write an interpreter `eval-action` which takes an action and a stack and returns the value produced by the machine at the completion of the action. We will adopt the convention that the machine produces a value by leaving it on the top of the stack when it halts.

## Specification of Operations

---

We want to write a *specification* for `eval-action`. The specification should say what `(eval-action a s)` does for each possible value of  $a$ .

The specification is given by the following equations.

$$(\text{eval-action halt } s) = (\text{car } s)$$
$$(\text{eval-action incr; } a \text{ (v w ...)}) = \\ (\text{eval-action } a \text{ (v+1 w ...)})$$
$$(\text{eval-action add; } a \text{ (v w x ...)}) = \\ (\text{eval-action } a \text{ ((v+w) x ...)})$$
$$(\text{eval-action push v; } a \text{ (w ...)}) = \\ (\text{eval-action } a \text{ (v w ...)})$$
$$(\text{eval-action pop; } a \text{ (v w ...)}) = \\ (\text{eval-action } a \text{ (w ...)})$$

## Representation of Operations

---

To write Scheme code to implement the specification of `eval-action`, we need to specify a representation of the type of actions. Let us make a simple choice:

```
[halt]           = (halt)
[incr; a]        = ( incr . [a] )
[add; a]         = ( add  . [a] )
[push v; a]      = ( push v . [a] )
[pop; a]         = ( pop  . [a] )
```

so an action is represented as a list of instructions. This is like bytecode: a typical action is

```
(push 3 push 4 add halt)
```

# A Stack Machine Interpreter

---

```
(define eval-action
  (lambda (action stack)
    (let ((op-code (car action)))
      (case op-code
        ((halt)
         (car stack))
        ((incr)
         (eval-action (cdr action)
                       (cons (+ (car stack) 1) (cdr stack))))
        ((add)
         (eval-action (cdr action)
                       (cons (+ (car stack) (cadr stack)) (cddr stack))))
        ((push)
         (let ((v (cadr action)))
           (eval-action (cddr action) (cons v stack))))
        ((pop)
         (eval-action (cdr action) (cdr stack)))
        (else
         (error 'eval-action "unknown op-code:" op-code))))))
```

## Running the Interpreter

---

```
(define start
  (lambda (action)
    (eval-action action '())))

> (start '(push 3 push 4 add halt))
7
```

# MiniScheme Grammar

---

*Expression* ::= *IntegerLiteral* | *TrueLiteral* | *FalseLiteral*  
| *PlusExpression*  
| *IfExpression*  
| *LetExpression*  
| *Identifier*  
| *Assignment*  
| *ProcedureExp* | *Application*  
| *RecExpression*

*IntegerLiteral* ::= *INTEGER\_LITERAL*

*TrueLiteral* ::= #t

*FalseLiteral* ::= #f

*PlusExpression* ::= ( + *Expression Expression* )

*IfExpression* ::= ( if *Expression Expression Expression* )

*LetExpression* ::= ( let ( *Declaration*\* ) *Expression* )

*Assignment* ::= ( set! *Identifier Expression* )

*ProcedureExp* ::= ( lambda ( *Identifier*\* ) *Expression* )

*Application* ::= ( *Expression Expression*\* )

*RecExpression* ::= ( letrec ( *RecDeclaration*\* ) *Expression* )

*Declaration* ::= ( *Identifier Expression* )

*RecDeclaration* ::= ( *Identifier ProcedureExp* )

## Records

---

If we load the file `recscm.scm` from the CS 239 webpage, then we can define, create and inspect records:

```
> (define-record person (name height country))
> (define me (make-person "Jens" 180 "Denmark"))
> me
(person "Jens" 180 "Denmark")
> (record-case me
    (person (a b c)
            (display b)))
180
> (person->height me)
180
```

By executing

```
jtbtb -scheme minischeme.jj
```

we get the record definitions on the next page, in the file `records.scm`. We also get a `SchemeTreeBuilder` visitor.

## Record Definitions

---

```
(define-record IntegerLiteral (Token))
(define-record TrueLiteral (Token))
(define-record FalseLiteral (Token))
(define-record PlusExpression (Token1 Token2
  Expression1 Expression2 Token3))
(define-record IfExpression (Token1 Token2
  Expression1 Expression2 Expression3 Token3))
(define-record LetExpression (Token1 Token2
  Token3 List Token4 Expression Token5))
(define-record Identifier (Token))
(define-record Assignment (Token1 Token2 Identifier Expression
  Token3))
(define-record ProcedureExp (Token1 Token2
  Token3 List Token4 Expression Token5))
(define-record Application (Token1 Expression List Token2))
(define-record RecExpression (Token1 Token2
  Token3 List Token4 Expression Token5))
(define-record Declaration (Token1 Identifier Expression Token2))
(define-record RecDeclaration (Token1 Identifier ProcedureExp
  Token2))
```

# Syntax Trees

---

If we run the automatically generated SchemeTreeBuilder visitor on the MiniScheme program

```
(if #t ((lambda (x y) x) 5 6) #f)
```

then we get the syntax tree:

```
(define root '(Goal
  (IfExpression "(" "if" (TrueLiteral "#t" )
    (Application "("
      (ProcedureExp "(" "lambda" "("
        ( (Identifier "x" )
          (Identifier "y" ) ) )"
        (Identifier "x" ) )" )
      ( (IntegerLiteral "5" )
        (IntegerLiteral "6" ) ) )" )
    (FalseLiteral "#f" ) )" ) "" ))
```

Our MiniScheme interpreter will work on programs in this representation.

## A MiniScheme Interpreter

---

```
(load "recscm.scm")
(load "records")
(load "tree")

(define run
  (lambda ()
    (record-case root
      (Goal (Expression Token)
        (eval-Expression Expression))
      (else (error 'run "Goal not found")))))
```

# Simple Expressions

---

```
(define eval-Expression
  (lambda (Expression)
    (record-case Expression
      (IntegerLiteral (Token) (string->number Token))
      (TrueLiteral (Token) #t)
      (FalseLiteral (Token) #f)
      (PlusExpression (Token1 Token2 Expression1
                      Expression2 Token3)
        (+ (eval-Expression Expression1)
           (eval-Expression Expression2)))
      (IfExpression (Token1 Token2 Expression1
                    Expression2 Expression3 Token3)
        (if (eval-Expression Expression1)
            (eval-Expression Expression2)
            (eval-Expression Expression3)))
      (else (error 'eval-Expression "Expression not found")))))
```

# Environments

---

To handle names, let, etc., we will use environments.

An environment is a function whose domain is a finite set of identifiers:

$\{ (s_1, v_1), \dots, (s_n, v_n) \}$ .

```
;;; interface Environment {  
;;;   type Env  
;;;   empty-Env   : Env  
;;;   apply-Env   : Env * Id -> Value  
;;;   extend-Env  : Id * Value * Env -> Env  
;;; }
```

$(\text{empty-env}) = [\emptyset]$

$(\text{apply-env } [f] s) = f(s)$

$(\text{extend-env } s v [f]) = [g],$

where  $g(s') = \begin{cases} v & s' = s \\ f(s') & \text{otherwise} \end{cases}$

# Environments

---

```
(define empty-env  
  (lambda () '()))
```

```
(define apply-env  
  (lambda (env id)  
    (cadr (assoc id env))))
```

```
(define extend-env  
  (lambda (id value env)  
    (cons (list (Identifier->Token id) value) env)))
```

```
(define extend-env-list  
  (lambda (ids vals env)  
    (if (null? ids)  
        env  
        (extend-env-list  
          (cdr ids)  
          (cdr vals)  
          (extend-env (car ids) (car vals) env)))))
```

## Environments

---

```
(define run
  (lambda ()
    (record-case root
      (Goal (Expression Token)
        (eval-Expression Expression (empty-env)))
      (else (error 'run "Goal not found")))))
```

# Let Expressions

---

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (IfExpression (Token1 Token2 Expression1
                    Expression2 Expression3 Token3)
        (if (eval-Expression Expression1 env)
            (eval-Expression Expression2 env)
            (eval-Expression Expression3 env)))
      (LetExpression (Token1 Token2 Token3
                     List Token4 Expression Token5)
        (let* ((ids (map Declaration->Identifier List))
               (exps (map Declaration->Expression List))
               (vals (map (lambda (Expression)
                           (eval-Expression Expression env))
                          exps))
               (new-env (extend-env-list ids vals env)))
          (eval-Expression Expression new-env)))
      (Identifier (Token) (apply-env env Token))
      (else (error ...))))))
```

# Cells

---

In Scheme we can assign to all variables. To handle this, we will use cells.

```
(define make-cell
  (lambda (value)
    (cons '*cell value)))
```

```
(define deref-cell cdr)
```

```
(define set-cell! set-cdr!)
```

When we extend an environment, we will create a cell, store the initial value in the cell, and bind the identifier to the cell.

```
(define extend-env
  (lambda (id value env)
    (cons (list (Identifier->Token id)
                (make-cell value))
          env)))
```

## Assignments

---

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (Identifier (Token)
        (deref-cell (apply-env env Token)))
      (Assignment (Token1 Token2 Identifier
                    Expression Token3)
        (set-cell!
          (apply-env env
                     (Identifier->Token Identifier))
          (eval-Expression Expression env))
        'ok)
      (else (error ...))))))
```

## Closures

---

To represent user-defined procedures, we will use closures.

```
(define-record closure (formals body env))
```

# Closures

---

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (ProcedureExp (Token1 Token2 Token3
                    List Token4 Expression Token5)
        (make-closure List Expression env))
      (Application (Token1 Expression List Token2)
        (let*
          ((clos (eval-Expression Expression env))
           (ids (closure->formals clos))
           (vals (map (lambda (Exp)
                       (eval-Expression Exp env))
                      List)))
          (static-env (closure->env clos))
          (new-env
            (extend-env-list ids vals static-env)))
          (body (closure->body clos))
          (eval-Expression body new-env)))
        (else (error ...))))))
```

# Recursive Environments

---

To handle recursive definitions, we need two kinds of environment records. Normal environments contain cells. A recursive environment contains a `RecDeclarationList`. If one looks up a recursively-defined procedure, then it gets closed in the environment frame that contains it:

```
(define-record empty-env ())
(define-record normal-env (ids vals env))
(define-record rec-env (recdecl-list env))

(define empty-env make-empty-env)

(define extend-env-list
  (lambda (ids vals env)
    (make-normal-env
      (map cadr ids)
      (map (lambda (value) (make-cell value))
           vals)
      env)))
```

## Letrec

---

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (RecExpression (Token1 Token2 Token3
                     List Token4 Expression Token5)
        (eval-Expression
          Expression
          (make-rec-env List env)))
      (else (error ...))))))
```

## Recursive Environments

---

```
(define apply-env
  (lambda (env id)
    (record-case env
      (empty-env () (error 'apply-env "Unbound identifier"))
      (normal-env (id-list vals old-env)
        (if (member id id-list)
            (ribassoc id id-list vals)
            (apply-env old-env id)))
      ...
      (else (error 'apply-env "unknown environment frame")))))
```

# Recursive Environments

---

```
(define apply-env
  (lambda (env id)
    (record-case env
      ...
      (rec-env (recdecl-list old-env)
        (let ((id-list (map (lambda (RecDeclaration)
                              (cadr (RecDeclaration->Identifier
                                      RecDeclaration)))
                            recdecl-list)))
          (if (member id id-list)
              (let* ((RecDeclaration
                     (ribassoc id id-list recdecl-list))
                    (ProcedureExp (RecDeclaration->ProcedureExp
                                    RecDeclaration)))
                (make-cell (make-closure ;; must return a cell!
                            (ProcedureExp->List ProcedureExp)
                            (ProcedureExp->Expression ProcedureExp)
                            env)))
                  (apply-env old-env id))))))
```

## Recursive Environments

---

```
(define ribassoc
  (lambda (id id-list vals)
    (cond
      ((null? id-list)
       (error 'ribassoc "id not found"))
      ((equal? id (car id-list))
       (car vals))
      (else
       (ribassoc id (cdr id-list) (cdr vals))))))
```

# Flyweight Java Grammar

---

```
Goal ::= MainClass
      TypeDeclaration*
MainClass ::= class Identifier {
            public static void main
              ( String [ ] Identifier ) {
                MethodBody
              }
            }
TypeDeclaration ::= InterfaceDeclaration
                 | InterfaceExtends
                 | ClassDeclaration
InterfaceDeclaration ::= interface Identifier InterfaceBody
InterfaceExtends ::= interface Identifier
                  extends Identifier InterfaceBody
InterfaceBody ::= { InterfaceMember* }
InterfaceMember ::= ResultType Identifier ( Type Identifier ) ;
```

# Flyweight Java Grammar

---

*ClassDeclaration* ::= class *Identifier* implements *Identifier* {  
    *ClassBodyDeclaration*\* }

*ClassBodyDeclaration* ::= *FieldDeclaration*  
    | *MethodDeclaration*

*FieldDeclaration* ::= *Type Identifier* ;

*MethodDeclaration* ::= public *ResultType Identifier*  
    ( *Type Identifier* ) { *MethodBody* }

*MethodBody* ::= *StatementListReturn*  
    | *StatementList*

*Type* ::= *BooleanType*  
    | *IntegerType*  
    | *Identifier*

*BooleanType* ::= boolean

*IntegerType* ::= int

*ResultType* ::= *VoidType*  
    | *Type*

*VoidType* ::= void

## Flyweight Java Grammar

---

*Block* ::= { *StatementList* }  
*StatementListReturn* ::= *StatementList* return *Expression* ;  
*StatementList* ::= *Statement*\*  
*Statement* ::= *Block*  
                  | *AssignmentStatement*  
                  | *IfStatement*  
                  | *MessageSendStatement*  
                  | *PrintStatement*  
*AssignmentStatement* ::= *Identifier* = *Expression* ;  
*IfStatement* ::= if ( *Expression* ) *Statement*  
                  else *Statement*  
*MessageSendStatement* ::= *MessageSend* ;  
*PrintStatement* ::= System.out.println( *Expression* ) ;

## Flyweight Java Grammar

---

*Expression* ::= *CompareExpression*  
                  | *PlusExpression*  
                  | *MinusExpression*  
                  | *MessageSend*  
                  | *PrimaryExpression*

*CompareExpression* ::= *PrimaryExpression* < *PrimaryExpression*  
*PlusExpression* ::= *PrimaryExpression* + *PrimaryExpression*  
*MinusExpression* ::= *PrimaryExpression* - *PrimaryExpression*  
*MessageSend* ::= *PrimaryExpression* . *Identifier* ( *Expression* )

## Flyweight Java Grammar

---

*PrimaryExpression* ::= *IntegerLiteral*  
| *TrueLiteral*  
| *FalseLiteral*  
| *Identifier*  
| *ThisExpression*  
| *AllocationExpression*  
| *BracketExpression*

*IntegerLiteral* ::= *INTEGER\_LITERAL*

*TrueLiteral* ::= true

*FalseLiteral* ::= false

*Identifier* ::= *IDENTIFIER*

*ThisExpression* ::= this

*AllocationExpression* ::= new *Identifier* ( )

*BracketExpression* ::= ( *Expression* )

## Record Definitions

---

```
(define-record Goal (MainClass List Token))
(define-record MainClass (Token1 Identifier1 Token2
  Token3 Token4 Token5 Token6
  Token7 Token8 Token9 Token10
  Identifier2 Token11 Token12 MethodBody
  Token13 Token14))
(define-record InterfaceDeclaration (Token Identifier
  InterfaceBody))
(define-record InterfaceExtends (Token1 Identifier1 Token2
  Identifier2 InterfaceBody))
(define-record InterfaceBody (Token1 List Token2))
(define-record InterfaceMember (ResultType Identifier1 Token1
  Type Identifier2 Token2 Token3))
(define-record ClassDeclaration (Token1 Identifier1 Token2
  Identifier2 Token3 List Token4))
(define-record FieldDeclaration (Type Identifier Token))
```

# Record Definitions

---

```
(define-record MethodDeclaration (Token1 ResultType
  Identifier1 Token2 Type Identifier2 Token3
  Token4 MethodBody Token5))
(define-record BooleanType (Token))
(define-record IntegerType (Token))
(define-record VoidType (Token))
(define-record Block (Token1 StatementList Token2))
(define-record StatementListReturn (StatementList Token1
  Expression Token2))
(define-record StatementList (List))
(define-record AssignmentStatement (Identifier Token1
  Expression Token2))
(define-record IfStatement (Token1 Token2 Expression
  Token3 Statement1 Token4 Statement2))
(define-record MessageSendStatement (MessageSend Token))
(define-record PrintStatement (Token1 Token2 Expression
  Token3 Token4))
```

## Record Definitions

---

```
(define-record CompareExpression (PrimaryExpression1 Token
    PrimaryExpression2))
(define-record PlusExpression (PrimaryExpression1 Token
    PrimaryExpression2))
(define-record MinusExpression (PrimaryExpression1 Token
    PrimaryExpression2))
(define-record MessageSend (PrimaryExpression Token1
    Identifier Token2 Expression Token3))
(define-record IntegerLiteral (Token))
(define-record TrueLiteral (Token))
(define-record FalseLiteral (Token))
(define-record Identifier (Token))
(define-record ThisExpression (Token))
(define-record AllocationExpression (Token1 Identifier Token2
    Token3))
(define-record BracketExpression (Token1 Expression Token2))
```

To represent objects we will use:

```
(define-record object (class-id env))
```

## Environments

---

```
(define empty-env
  (lambda ()
    '()))
```

```
(define apply-env
  (lambda (env id)
    (display env) (newline) (display id) (newline)
    (cadr (assoc id env))))
```

```
(define extend-env
  (lambda (id value env)
    (cons (list (Identifier->Token id) (make-cell value))
          env)))
```

## Cells

---

```
(define make-cell  
  (lambda (value)  
    (cons '*cell value)))
```

```
(define deref-cell cdr)
```

```
(define set-cell! set-cdr!)
```

## A Flyweight Java Interpreter

---

```
(load "recscm.scm")
(load "records")
(load "tree")

(define run
  (lambda ()
    (record-case root
      (Goal (MainClass List Token)
        (run-MainClass MainClass List))
      (else (error 'run "Goal not found")))))
```

## run-MainClass

---

```
(define run-MainClass
  (lambda (MainClass class-env)
    (record-case MainClass
      (MainClass (Token1 Identifier1 Token2
                  Token3 Token4 Token5 Token6
                  Token7 Token8 Token9 Token10
                  Identifier2 Token11 Token12 MethodBody
                  Token13 Token14)
                (eval-MethodBody MethodBody class-env (empty-env)
                                  'no-this))
      (else (error 'run-MainClass "MainClass not found")))))
```

## eval-MethodBody

---

```
(define eval-MethodBody
  (lambda (MethodBody class-env env this)
    (record-case MethodBody
      (StatementListReturn (StatementList Token1 Expression
                                           Token2)
        (eval-StatementList StatementList class-env env this)
        (eval-Expression Expression class-env env this))
      (StatementList (List)
        (eval-StatementList MethodBody class-env env this))
      (else (error 'eval-MethodBody "MethodBody not found")))))
```

```
(define eval-StatementList
  (lambda (StatementList class-env env this)
    (record-case statementlist
      (StatementList (List)
        (for-each
          (lambda (Statement)
            (eval-Statement Statement class-env env this))
          List))))))
```

## eval-Statement

---

```
(define eval-Statement
  (lambda (Statement class-env env this)
    (record-case Statement
      (Block (Token1 StatementList Token2)
        (eval-StatementList StatementList class-env env this))
      (AssignmentStatement (Identifier Token1 Expression Token2)
        (set-cell! (apply-env env (Identifier->Token Identifier))
          (eval-Expression Expression class-env env this)))
      (IfStatement (Token1 Token2 Expression Token3 Statement1
                    Token4 Statement2)
        (if (eval-Expression Expression class-env env this)
            (eval-Statement Statement1 class-env env this)
            (eval-Statement Statement2 class-env env this)))
      (MessageSendStatement (MessageSend Token)
        (eval-Expression MessageSend class-env env this))
      (PrintStatement (Token1 Token2 Expression Token3 Token4)
        (display (eval-Expression Expression class-env env this))
        (newline))
      (else (error 'eval-Statement "Statement not found")))))
```

## eval-Expression

---

```
(define eval-Expression
  (lambda (Expression class-env env this)
    (record-case Expression
      (CompareExpression (PrimaryExpression1 Token
                          PrimaryExpression2)
        (< (eval-Expression PrimaryExpression1
                            class-env env this)
           (eval-Expression PrimaryExpression2
                            class-env env this))))
      (PlusExpression (PrimaryExpression1 Token
                      PrimaryExpression2)
        (+ (eval-Expression PrimaryExpression1
                            class-env env this)
           (eval-Expression PrimaryExpression2
                            class-env env this))))
      (MinusExpression ... ))))
```

## eval-Expression

---

```
(MessageSend (PrimaryExpression Token1 Identifier Token2
              Expression Token3)
  (let* ((receiver (eval-Expression
                    PrimaryExpression class-env env this))
         (argument (eval-Expression Expression
                                    class-env env this))
         (class-id (object->class-id receiver))
         (CDDL (apply-class-env class-env class-id))
         (MethodDeclaration
              (find-method CDDL Identifier)))
        (record-case MethodDeclaration
          (MethodDeclaration (Token1 ResultType Identifier1
                             Token2 Type Identifier2 Token3
                             Token4 MethodBody Token5)
            (eval-MethodBody
              MethodBody class-env
              (extend-env Identifier2
                          argument (object->env receiver))
              receiver))))))
```

## eval-Expression

---

```
(IntegerLiteral (Token)
  (string->number Token))
(TrueLiteral ()
  #t)
(FalseLiteral ()
  #f)
(Identifier (Token)
  (deref-cell (apply-env env Token)))
(ThisExpression (Token)
  this)
(AllocationExpression (Token1 Identifier Token2 Token3)
  (make-object Identifier
    (allocate (apply-class-env class-env
      Identifier))))
(BracketExpression (Token1 Expression Token2)
  (eval-Expression Expression class-env env this))
(else (error 'eval-Expression "Expression not found"))))
```

## Finding Classes

---

;;; search procedures

```
(define apply-class-env
      ;; returns a ClassBodyDeclarationList
  (lambda (class-env class-id)
      ;; class-env is a TypeDeclarationList
    (if (null? class-env)
        (error 'apply-class-env "Class not found")
        (record-case (car class-env)
                     (ClassDeclaration (Token1 Identifier1 Token2
                                         Identifier2 Token3 List Token4)
                                       (if (equal? Identifier1 class-id)
                                           List
                                           (apply-class-env (cdr class-env) class-id)))
                     (else (apply-class-env (cdr class-env) class-id))))))
```

## Finding Methods

---

```
(define find-method          ;; returns a MethodDeclaration
  (lambda (ClassBodyDeclarationList method-id)
    (if (null? ClassBodyDeclarationList)
        (error 'find-method "Method not found")
        (record-case (car ClassBodyDeclarationList)
                     (FieldDeclaration (Type Identifier Token)
                                       (find-method (cdr ClassBodyDeclarationList)
                                                  method-id))
                     (MethodDeclaration (Token1 ResultType Identifier1
                                         Token2 Type Identifier2 Token3
                                         Token4 MethodBody Token5)
                                       (if (equal? Identifier1 method-id)
                                           (car ClassBodyDeclarationList)
                                           (find-method (cdr ClassBodyDeclarationList)
                                                  method-id))))))))))
```

# Allocating Objects

---

```
(define allocate          ;; returns an env
  (lambda (ClassBodyDeclarationList)
    (allocate-CBDL ClassBodyDeclarationList (empty-env))))
```

```
(define allocate-CBDL    ;; returns an env
  (lambda (ClassBodyDeclarationList env)
    (if (null? ClassBodyDeclarationList)
        env
        (record-case (car ClassBodyDeclarationList)
                    (FieldDeclaration (Type Identifier Token)
                                       (allocate-CBDL (cdr ClassBodyDeclarationList)
                                                       (extend-env Identifier
                                                                    '*undefined env)))
                    (MethodDeclaration (Token1 ResultType Identifier1
                                         Token2 Type Identifier2 Token3
                                         Token4 MethodBody Token5)
                                       (allocate-CBDL (cdr ClassBodyDeclarationList)
                                                       env))))))
```

## Chapter 6: Operational Semantics

---

# Operational Semantics

---

An interpreter can be viewed as an executable description of the semantics of a programming language.

We will now present more formal ways of describing the operational semantics of programming languages. We will cover two of the main approaches:

- **Big-step semantics** with environments.
- **Small-step semantics** with syntactic substitution.

Big-step semantics with environments is close in spirit to the interpreters we have seen earlier.

Small-step semantics with syntactic substitution formalizes the inlining of a procedure call as an approach to computation.

# Lambda-calculus

---

Here is a tiny subset of Scheme, called the lambda-calculus:

$$\begin{aligned} \textit{Expression} & ::= \textit{Identifier} \mid \textit{ProcExp} \mid \textit{AppExp} \\ \textit{ProcExp} & ::= (\textit{lambda} (\textit{Identifier}) \textit{Expression}) \\ \textit{AppExp} & ::= (\textit{Expression} \textit{Expression}) \end{aligned}$$

The only values that can be computed are *procedures*, and the only computation that can take place is *procedure application*.

The traditional syntax for procedures in the lambda-calculus uses the Greek letter  $\lambda$  (lambda), and the grammar for the lambda-calculus can be written:

$$\begin{aligned} e & \in \textit{Expression} \\ e & ::= x \mid \lambda x.e \mid e_1e_2 \\ x & \in \textit{Identifier} \quad (\text{infinite set of variables}) \end{aligned}$$

In the lambda-calculus, brackets are only used for grouping of expressions. There is a standard convention for saving brackets that says that the body of a  $\lambda$ -abstraction extends “as far as possible.” For example,  $\lambda x.xy$  is short for  $\lambda x.(xy)$  and not  $(\lambda x.x)y$ . Moreover,  $e_1e_2e_3$  is short for  $(e_1e_2)e_3$  and not  $e_1(e_2e_3)$ .

## Extension of the Lambda-calculus

---

We will give the semantics for the following extension of the lambda-calculus:

$e \in \textit{Expression}$

$e ::= x \mid \lambda x.e \mid e_1e_2 \mid c \mid \textit{succ } e$

$x \in \textit{Identifier}$  (infinite set of variables)

$c \in \textit{IntegerConstant}$

# Big-Step Semantics

---

Here is a big-step semantics with environments for the lambda-calculus.

$$\begin{aligned}w, v &\in \textit{Value} \\ v &::= c \mid \langle \lambda x. e, \rho \rangle \\ \rho &\in \textit{Environment} \\ \rho &::= x_1 \mapsto v_1, \dots, x_n \mapsto v_n\end{aligned}$$

The semantics is given by five rules:

$$\rho \vdash x \triangleright v \quad (\rho(x) = v) \tag{1}$$

$$\rho \vdash \lambda x. e \triangleright \langle \lambda x. e, \rho \rangle \tag{2}$$

$$\frac{\rho \vdash e_1 \triangleright \langle \lambda x. e, \rho' \rangle \quad \rho \vdash e_2 \triangleright v \quad \rho', x \mapsto v \vdash e \triangleright w}{\rho \vdash e_1 e_2 \triangleright w} \tag{3}$$

$$\rho \vdash c \triangleright c \tag{4}$$

$$\frac{\rho \vdash e \triangleright c_1}{\rho \vdash \textit{succ } e \triangleright c_2} \quad ([c_2] = [c_1] + 1) \tag{5}$$

# Small-Step Semantics

---

In small-step semantics, one step of computation =  
either one **primitive** operation,  
or **inline** one procedure call.

We can do steps of computation in different orders:

```
> (define foo  
    (lambda (x y) (+ (* x 3) y)))  
> (foo (+ 4 1) 7)  
22
```

Let us calculate:

```
(foo  
  (+ 4 1) 7)  
=> ((lambda (x y) (+ (* x 3) y))  
    (+ 4 1) 7)  
=> (+ (* (+ 4 1) 3) 7)  
=> 22
```

## Small-Step Semantics

---

We can also calculate like this:

```
(foo  
  (+ 4 1) 7)
```

```
=> (foo  
    5 7)
```

```
=> ((lambda (x y) (+ (* x 3) y))  
    5 7)
```

```
=> (+ (* 5 3) 7)
```

```
=> 22
```

# Free Variables

---

A variable  $x$  occurs free in an expression  $E$  if and only if

- $E$  is a variable reference and  $E$  is the same as  $x$ ; or
- $E$  is of the form  $(E_1E_2)$  and  $x$  occurs free in  $E_1$  or  $E_2$ ; or
- $E$  is of the form  $(\text{lambda } (y) E')$ , where  $y$  is different from  $x$  and  $x$  occurs free in  $E'$ .

Example: no variables occur free in the expression

```
(lambda (y) ((lambda (x) x) y))
```

Another example: the variable  $y$  occurs free in the expression

```
((lambda (x) x) y)
```

Standard terminology: an expression is *closed* if it does not contain free variables.

## Procedure Application

---

Now consider an application:

( *rator rand* )

To actually perform a procedure application, we have to evaluate the operator (*rator*) to a procedure. This gives:

( (lambda (*var*) *body* ) *rand* )

The basic Scheme rule for this situation is:

**The operand (*rand*) is evaluated to a value *before* the procedure call takes place.**

## Example of Call-by-Value

---

```
((lambda (x) x)
  ((lambda (y) (+ y 9)) 5))
```

```
=> ((lambda (x) x) (+ 5 9))
```

```
=> ((lambda (x) x) 14)
```

```
=> 14
```

This is called call-by-value reduction.

The slogan is: “always evaluate the arguments first”.

This is the evaluation principle for Scheme and ML.

## An Alternative Rule

---

Consider again the situation:

( (lambda (var) *body* ) *rand* )

Let us now change the Semantics completely!

Here is a new evaluation principle:

**The procedure call takes place without evaluating the operand *rand* at all.**

## Example of Call-by-Name

---

```
((lambda (x) x)
  ((lambda (y) (+ y 9) 5))
```

```
=> ((lambda (y) (+ y 9)) 5)
```

```
=> (+ 5 9)
```

```
=> 14
```

This is called call-by-name reduction.

The slogan is: “don’t work if you can avoid it; be lazy!”.

This is the evaluation principle for Miranda and Haskell.

## Does it make a Difference?

---

Now we have one language with two different evaluation principles. That is: two semantics!

What if we take a program and first run it with call-by-value reduction (just run it in Scheme) and then run it again but this time with call-by-name reduction (have to build a new system for this case ...).

Is there at least one program where the two runs will give different results?

Answer: the following is true of all programs:

- If the run with call-by-value reduction terminates, then the run with call-by-name reduction terminates. (But the converse is in general false).
- If both runs terminate, then they give the same result.

## Call-by-value can be too Eager

---

Sometimes call-by-value reduction fails to terminate, even though call-by-name reduction terminates.

Notice that:

```
(define delta (lambda (x) (x x)))
```

```
(delta delta)
```

```
=> (delta delta)
```

```
=> (delta delta)
```

```
=> ...
```

## Call-by-value can be too Eager

---

Consider the program:

```
((lambda (y) 3) (delta delta))
```

Call-by-value reduction fails to terminate because it continues forever with trying to evaluate the operand.

Call-by-name reduction terminates, simply because it throws away the argument:

```
((lambda (y) 3) (delta delta))
```

=> 3

## Call-by-value is more Efficient

---

Consider this call-by-name reduction:

```
((lambda (x) (x (x '((2 3))))))
  ((lambda (w) w) car))
=> (((lambda (w) w) car)
     (((lambda (w) w) car) '((2 3))))
=> (car (((lambda (w) w) car) '((2 3))))
=> (car (car '((2 3)))) => (car '(2 3)) => 2
```

And now, with call-by-value reduction (one step shorter!):

```
((lambda (x) (x (x '((2 3))))))
  ((lambda (w) w) car))
=> ((lambda (x) (x (x '((2 3))))))
  car)
=> (car (car '((2 3)))) => (car '(2 3)) => 2
```

## Beta-Redex

---

A procedure call which is ready to be inlined is of the form:

$$( (\text{lambda } (var) \textit{body} ) \textit{rand} )$$

Such a pattern is called a *beta-redex*.

We can generalize call-by-value reduction and call-by-name reduction by allowing the next redex chosen to be *any* beta-redex, also one in the body of a lambda.

The process of inlining a beta-redex for some redex is called *beta-reduction*.

$$( (\text{lambda } (var) \textit{body} ) \textit{rand} ) \Rightarrow \textit{body}[var:=rand]$$

Here,  $\textit{body}[var:=rand]$  intuitively means “the result is like *body* except that all occurrences of *var* is replaced by *rand*”.

It is when we allow arbitrary beta-reduction to take place, that the tiny subset of Scheme is truly the *lambda-calculus*.

## Name Clashes

---

Care must be taken to avoid name clashes. Example:

```
((lambda (x)
  (lambda (y) (y x)))
 (y 5))
```

should *not* be transformed into

```
(lambda (y) (y (y 5)))
```

The reference to  $y$  in  $(y\ 5)$  should remain free!

The solution is to change the name of the inner variable name  $y$  to some name, say  $z$ , that does not occur free in the argument  $y\ 5$ .

```
((lambda (x)
  (lambda (z) (z x)))
 (y 5))
=> (lambda (z) (z (y x)))    ;; the y is still there!
```

# Substitution

---

The notation  $e[x := M]$  denotes  $e$  with  $M$  substituted for every free occurrence of  $x$  in such a way that name clashes are avoided. We will define  $e[x := M]$  inductively on  $e$ .

$$\begin{aligned}x[x := M] &\equiv M \\y[x := M] &\equiv y \quad (x \neq y) \\(\lambda x.e_1)[x := M] &\equiv \lambda x.e_1 \\(\lambda y.e_1)[x := M] &\equiv \lambda z.((e_1[y := z])[x := M]) \\&\quad (\text{where } x \neq y \text{ and} \\&\quad z \text{ does not occur free in } e_1 \text{ or } M) \\(e_1 e_2)[x := M] &\equiv (e_1[x := M]) (e_2[x := M]) \\c[x := M] &\equiv c \\(\text{succ } e_1)[x := M] &\equiv \text{succ } (e_1[x := M])\end{aligned}$$

The subcase for application always creates a new variable. We can save work by observing that if  $x$  does not occur free in  $e$ , then there is nothing to do. Moreover, if  $y$  does not occur free in  $M$ , then we need not rename  $y$  to  $z$ .

## Alpha-Conversion

---

The renaming of a bound variable by

$$\lambda x.e \equiv \lambda z.(e[z := x])$$

( $z$  does not occur free in  $e$ ), is known as *alpha-conversion*.

# Small-Step Semantics

---

Here is a small-step semantics with syntactic substitution for the  $\lambda$ -calculus.

$$\begin{aligned}v &\in \textit{Value} \\v &::= c \mid \lambda x.e\end{aligned}$$

The semantics is given by the reflexive, transitive closure of the relation  $\rightarrow_V$ :

$$\rightarrow_V \subseteq \textit{Expression} \times \textit{Expression}$$

$$(\lambda x.e)v \rightarrow_V e[x := v] \tag{6}$$

$$\frac{e_1 \rightarrow_V e'_1}{e_1 e_2 \rightarrow_V e'_1 e_2} \tag{7}$$

$$\frac{e_2 \rightarrow_V e'_2}{v e_2 \rightarrow_V v e'_2} \tag{8}$$

$$\textit{succ } c_1 \rightarrow_V c_2 \quad (\lceil c_2 \rceil = \lceil c_1 \rceil + 1) \tag{9}$$

$$\frac{e_1 \rightarrow_V e_2}{\textit{succ } e_1 \rightarrow_V \textit{succ } e_2} \tag{10}$$

## Chapter 7: Continuation Passing Style

---

# Overview

---

This note shows how to translate a Scheme program into efficient low-level code. The translation is in a number of steps.

1. From a Scheme program to a Scheme program in *tail form*. The idea is to transform the program such that functions never return. This is done by introducing *continuations*.
2. From a Scheme program in tail-form to a Scheme program in *first-order form*. The idea is to transform the program such that all functions are defined at the top level. This is done by representing the continuations as first-order data structures.
3. From a Scheme program in first-order form to a Scheme program in *imperative form*. The idea is to transform the program such that functions take no arguments. This is done by passing the arguments in a fixed number of global variables. It is possible to avoid using a stack because the program is in tail form.
4. From a Scheme program in imperative form to C, machine code, etc. A Scheme program in imperative form is so close to machine code that the key task is to replace each function call with a jump.

## Game Plan

---

1. From Scheme to Tail Form
2. From Tail Form to First-Order Form
3. From First-Order Form to Imperative Form
4. From Imperative Form to Machine Code

These notes covers the first three steps. The reader will have no difficulty working out the details of the fourth step.

## Recursive Style

---

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

## Iterative Style

---

### Java

```
int fact(int n) {  
    int a=1;  
    while (n!=0) {  
        a=n*a;  
        n=n-1;  
    }  
    return a;  
}
```

### Scheme

```
(define fact-iter  
  (lambda (n)  
    (fact-iter-acc n 1)))  
  
(define fact-iter-acc  
  (lambda (n a)  
    (if (zero? n)  
        a  
        (fact-iter-acc (- n 1) (* n a)))))
```

## Recursive versus Iterative Behavior

---

```
(fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= 24
```

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

## Example

---

```
(define even-length?  
  (lambda (l)  
    (if (null? l) #t (odd-length? (cdr l)))))
```

```
(define odd-length?  
  (lambda (l)  
    (if (null? l) #f (even-length? (cdr l)))))
```

```
even-length?: if (null? l) then return #t  
               else begin l := (cdr l);  
                        goto odd-length?  
               end  
odd-length?:   if (null? l) then return #f  
               else begin l := (cdr l);  
                        goto even-length?  
               end
```

## A Grammar for CPS

---

Below is a grammar which describes programs that look like the ones above. We need two non-terminals: *SimpleExp*, for expressions that can go on the right-hand sides of assignments, etc., and *TailFormExp*, for the programs themselves.

```
TailFormExp ::= SimpleExp
                | ( SimpleExp SimpleExp1 ... SimpleExpn )
                | (if SimpleExp TailFormExp TailFormExp )
                | (begin SimpleExp1 ... SimpleExpn TailFormExp )
SimpleExp ::= Identifier
                | Constant
                | ( PrimitiveOperation SimpleExp1 ... SimpleExpn )
                | (set! Identifier SimpleExp )
                | (lambda ( Identifier1 ... Identifiern ) TailFormExp )
```

Tail form positions are those where the subexpression, when evaluated, gives the value of the whole expression (no promises or wrapping). All other positions must be simple.

## A Grammar for CPS

---

A more complete grammar:

$$\begin{aligned} \textit{TailFormExp} & ::= (\textit{cond} (\textit{SimpleExp} \textit{TailFormExp}) \dots) \\ & \quad | (\textit{let} (\textit{SimpleDeclList}) \textit{TailFormExp}) \\ & \quad | (\textit{letrec} (\textit{SimpleDeclList}) \textit{TailFormExp}) \\ \textit{SimpleDeclList} & ::= \textit{SimpleDecl}_1 \dots \textit{SimpleDecl}_n \\ \textit{SimpleDecl} & ::= (\textit{Identifier} \textit{SimpleExp}) \end{aligned}$$

Note: our *SimpleExp* corresponds to what EOPL calls “simple *and* tail-form”.

## Examples

---

```
(car x)                tail form (also simple)
(car (cdr x))          tail form (also simple)
(car (f x))            not tail form
(f (car x))           tail form
(lambda (v) (k (+ v 1))) tail form
(lambda (v) (+ (k v) 1)) not tail form
(if (zero? x) (f (+ x 1)) (g (- x 1))) tail form
(if (zero? x) (+ (f x) 1) (g (- x 1))) not tail form
(if (p x) (f (+ x 1)) (g (- x 1))) not tail form
(if (p (car x)) (f (+ x 1)) (g (- x 1))) not tail form
```

```
(lambda (x)                ; not tail form
  (if (zero? x) 0 (+ (f (- x 1)) 1)))
```

```
(lambda (x) ; tail form
  (if (zero? x) 0 (f (- x 1) (lambda (v) (k (+ v 1))))))
```

```
(lambda (n a) ; tail form
  (if (zero? n) a (fact-iter-acc (- n 1) (* n a))))
```

# Converting to Tail Form

---

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

Typical state:

```
(* 6 (* 5 (* 4 (fact 3))))
```

Let's write another version of `fact`, called `(fact-cps n g)`, which computes `(g (fact n))`.

```
(define fact-cps
  (lambda (n g)
    (if (zero? n)
        (g 1)
        (fact-cps (- n 1)
                  (lambda (v) (g (* n v)))))))
```

# From Scheme to Tail Form: The Transformation Rules

---

```
(define foo  
  (lambda (x y) -----))
```

==>

```
(define foo-cps  
  (lambda (x y k) (k -----)))
```

```
(k (foo a (- n 1)))
```

==>

```
(foo-cps a (- n 1) k)
```

```
(k (----- (foo a (- n 1)) -----))
```

==>

```
(foo-cps a (- n 1) (lambda (v) (k (----- v -----))))
```

## Example

---

```
(define remove-all
  (lambda (a lsym)
    (cond
      ((null? lsym) '())
      ((eq? a (car lsym))
       (remove-all a (cdr lsym)))
      (else (cons (car lsym) (remove-all a (cdr lsym)))))))
```

==>

```
(define remove-all-cps
  (lambda (a lsym k)
    (k (cond
        ((null? lsym) '())
        ((eq? a (car lsym))
         (remove-all a (cdr lsym)))
        (else (cons (car lsym) (remove-all a (cdr lsym)))))))
```

## Example

---

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond
      ((null? lsym) (k '()))
      ((eq? a (car lsym))
       (k (remove-all a (cdr lsym))))
      (else
       (k (cons (car lsym) (remove-all a (cdr lsym))))))))
```

==>

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym) (k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
           (remove-all-cps a (cdr lsym)
                             (lambda (v) (k (cons (car lsym) v))))))))
```

## Example

---

```
(define subst
  (lambda (old new s)
    (if (pair? s)
        (cons (subst old new (car s))
              (subst old new (cdr s)))
        (if (eq? s old) new s))))
```

==>

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (k (cons (subst old new (car s))
                 (subst old new (cdr s))))
        (if (eq? s old) (k new) (k s)))))
```

## Example

---

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (cdr s)
                    (lambda (cdr-val)
                      (k (cons (subst old new (car s)) cdr-val))))
        (if (eq? s old) (k new) (k s)))))
```

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (cdr s)
                    (lambda (cdr-val)
                      (subst-cps old new (car s)
                                  (lambda (car-val)
                                    (k (cons car-val cdr-val))))))
        (if (eq? s old) (k new) (k s)))))
```

## Example

---

Alternatively, we can compute the car first:

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (car s)
                    (lambda (car-val)
                      (subst-cps old new (cdr s)
                                  (lambda (cdr-val)
                                    (k (cons car-val cdr-val))))))
        (if (eq? s old)
            (k new)
            (k s))))))
```

## Example

---

```
(define subst-with-letrec
  (lambda (old new s)
    (letrec
      ((loop
        ;; (loop s) = (subst old new s)
        (lambda (s)
          (if (pair? s)
              (cons (loop (car s))
                    (loop (cdr s)))
              (if (eq? s old) new s))))))
      (loop s))))
```

## Example

---

```
(define subst-with-letrec-cps
  (lambda (old new s k)
    (letrec
      ((cps-loop
        (lambda (s k)
          (if (pair? s)
              (cps-loop (car s)
                        (lambda (car-val)
                          (cps-loop (cdr s)
                                    (lambda (cdr-val)
                                      (k (cons car-val cdr-val)))))))
              (if (eq? s old) (k new) (k s))))))
      (cps-loop s k))))
```

## Example

---

```
(define subst-with-letrec
  (lambda (old new s)
    (letrec
      ((cps-loop
        (lambda (s k)
          (if (pair? s)
              (cps-loop (car s)
                        (lambda (car-val)
                          (cps-loop (cdr s)
                                    (lambda (cdr-val)
                                      (k (cons car-val cdr-val)))))))
              (if (eq? s old) (k new) (k s))))))
      (cps-loop s (lambda (x) x))))))
```

## CPSing if and let

---

These follow our rules, too!

```
(k (if (foo x) ... ...)) = (foo-cps x
                           (lambda (v)
                             (k (if v ... ...))))
= (foo-cps x
   (lambda (v)
     (if v (k ...) (k ...))))
```

```
(k (let ((y (foo x))) ...)) = (foo-cps x
                              (lambda (v)
                                (k (let ((y v)) ...))))
= (foo-cps x
   (lambda (v)
     (let ((y v)) (k ...))))
= (foo-cps x
   (lambda (y)
     (k ...)))
```

# Typed CPS Transformation

---

CPS transformation of simply-typed  $\lambda$ -terms with call-by-value semantics.

$$\begin{array}{l} \text{Types:} \quad t ::= t \rightarrow t \mid \text{Int} \\ \text{Expressions:} \quad e ::= x \mid \lambda x.e \mid e_1 e_2 \end{array}$$

Here is a CPS transformation which is defined by a function  $[[\cdot]]$ :

$$\begin{array}{l} [[x]] = \lambda k.kx \\ [[\lambda x.e]] = \lambda k.k(\lambda x. [[e]]) \\ [[e_1 e_2]] = \lambda k. [[e_1]] (\lambda v_1. [[e_2]] (\lambda v_2. (v_1 v_2) k)) \end{array}$$

Let  $o$  be a type of answers. Define  $t^*$  inductively:

$$\begin{array}{l} \text{Int}^* = \text{Int} \\ (s \rightarrow t)^* = s^* \rightarrow (t^* \rightarrow o) \rightarrow o \end{array}$$

Define  $A^*(x) = t^*$  if  $A(x) = t$ .

**Theorem** If  $A \vdash e : t$ , then  $A^* \vdash [[e]] : (t^* \rightarrow o) \rightarrow o$ .

The proof is by induction on the structure of the derivation of  $A \vdash e : t$ .

## From Tail Form to First-Order Form

---

We've converted to tail form by using continuation-passing style. Now we want to get rid of all those higher-order functions.

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym)
           (k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
           (remove-all-cps a (cdr lsym)
                             (lambda (v) (k (cons (car lsym) v))))))))))
```

```
(define remove-all
  (lambda (a lsym)
    (remove-all-cps a lsym (lambda (v) v))))
```

## Continuations as a Datatype

---

Grammar for continuations

*Continuation* ::= (lambda (v) v)  
                  ::= (lambda (v) (*Continuation* (cons (car lsym) v)))

Interface Specification. (Remember  $\lceil k \rceil$  means a representation of  $k$ ).

```
(make-identity) =  $\lceil$ (lambda (v) v) $\rceil$   
(make-rem1 lsym  $\lceil k \rceil$ ) =  $\lceil$ (lambda (v) (k (cons (car lsym) v))) $\rceil$   
(apply-continuation  $\lceil k \rceil$  v) = (k v)
```

## Representation-Independent Code

---

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym)
           (apply-continuation k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
           (remove-all-cps a (cdr lsym)
                             (make-rem1 lsym k))))))

(define remove-all
  (lambda (a lsym)
    (remove-all-cps a lsym (make-identity))))
```

## Procedural Representation of Continuations

---

```
(define make-identity
  (lambda ()
    (lambda (v) v)))
```

```
(define make-rem1
  (lambda (lsym k)
    (lambda (v)
      (apply-continuation k (cons (car lsym) v)))))
```

```
(define apply-continuation
  (lambda (k v)
    (k v)))
```

# Abstract Syntax Tree Representation

---

```
(make-identity) = '(identity-record)
(make-rem1 v k) = '(rem1-record v k)
```

Here the right-hand sides are patterns for list structures; the value of `(make-rem1 v k)` is a list whose first element is the symbol `rem1`, whose second element is the value of `v`, and whose third element is the value of `k`.

```
(define-record identity-record ())
(define-record rem1-record (lsym k))

(define apply-continuation
  (lambda (k v)
    (record-case k
      (identity-record () v)      ; this was (make-identity)
      (rem1-record (lsym k)      ; this was (make-rem1 lsym k)
        (apply-continuation k (cons (car lsym) v)))
      (else (error "bad continuation"))))
```

# Nifty Tree Representation

---

Since there's only one possible tag (`rem1`), we don't need to include it.

```
(make-identity) = '()  
(make-rem1 lsym k) = (lsym . k)
```

```
(define make-identity  
  (lambda () '()))
```

```
(define make-rem1  
  (lambda (lsym k) (cons lsym k)))
```

```
(define apply-continuation  
  (lambda (k v)  
    (if (null? k)  
        v  
        (let ((lsym (car k))  
              (k1 (cdr k)))  
          (apply-continuation k1 (cons (car lsym) v))))))
```

## Even Niftier Tree Representation

---

Like the previous, but store only `(car lsym)`, not `lsym`.

```
(make-identity) = '()
```

```
(make-rem1 lsym k) = '(,(car lsym) . ,k)
```

Here `'` introduces a pattern; `,` marks a pattern expression, so `'(,(car lsym) . ,k)` denotes a cons-cell whose `car` is the value of the `car` of `lsym` and whose `cdr` is the value of `k`. This is backquote notation; see Dybvig.

## Even Niftier Tree Representation

---

```
(define make-identity (lambda () '()))
```

```
(define make-rem1  
  (lambda (lsym k)  
    (cons (car lsym) k)))
```

```
(define apply-continuation  
  (lambda (k v)  
    (if (null? k)  
        v  
        (let ((car-of-lsym (car k))  
              (k1 (cdr k)))  
          (apply-continuation k1  
                               (cons car-of-lsym v)))))))
```

## Another Case Study: subst

---

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (car s)
                    (lambda (v1)
                      (subst-cps old new (cdr s)
                                  (lambda (v2) (k (cons v1 v2))))))
        (if (eq? s old) (k new) (k s))))))
```

```
(define subst
  (lambda (old new s)
    (subst-cps old new s (lambda (x) x))))
```

```
Continuation ::= (lambda (x) x)
               ::= (lambda (v1)
                     (subst-cps old new (cdr s)
                                   (lambda (v2) (Continuation (cons v1 v2))))))
               ::= (lambda (v2) (Continuation (cons v1 v2)))
```

## Datatype of subst-Continuations

---

```
(make-identity) = [(lambda (x) x)]
```

```
(make-subst1 old new s [k]) = [(lambda (v1)
  (subst-cps old new (cdr s)
    (lambda (v2) (k (cons v1 v2)))))]
```

```
(make-subst2 v1 [k]) = [(lambda (v2) (k (cons v1 v2)))]
```

```
(apply-continuation [k] v) = (k v)
```

## Representation-Independent Code

---

```
(define subst-cps
  (lambda (old new s k)                ; k is a representation
    (if (pair? s)
        (subst-cps old new (car s)
                    (make-subst1 old new s k))
        (if (eq? s old)
            (apply-continuation k new)
            (apply-continuation k s))))))

(define subst
  (lambda (old new s)
    (subst-cps old new s (make-identity))))
```

# Procedural Representation

---

```
(define make-identity
  (lambda ()
    (lambda (x) x)))
```

```
(define make-subst1
  (lambda (old new s k)
    (lambda (v1)
      (subst-cps old new (cdr s) (make-subst2 v1 k)))))
```

```
(define make-subst2
  (lambda (v1 k)
    (lambda (v2) (apply-continuation k (cons v1 v2)))))
```

```
(define apply-continuation
  (lambda (k v)
    (k v)))
```

# Abstract Syntax Tree Representation

---

```
(define-record identity-record ())
(define-record subst1-record (old new s k))
(define-record subst2-record (v1 k))

(define apply-continuation
  (lambda (k x)
    (record-case k
      (identity-record () x)
      (subst1-record (old new s k)
        (let ((v1 x))
          (subst-cps old new (cdr s)
            (make-subst2 v1 k))))
      (subst2-record (v1 k)
        (let ((v2 x))
          (apply-continuation k (cons v1 v2))))
      (else (error "bad continuation")))))
```

## List of Frames Representation of Continuations

---

```
(make-identity) = '((lastframe))
```

```
(make-subst1 old new s k) = '((frame1 ,old ,new ,s) . ,k)
```

```
(make-subst2 v1 k) = '((frame2 ,v1) . ,k)
```

Typical continuation looks like a list of frames:

```
((frame1 <o> <n> <s_1>)  
 (frame2 <v_1>)  
 (frame2 <v_2>)  
 ...  
 (lastframe))
```

## List of Frames Representation of Continuations

---

```
(define-record lastframe ())  
(define-record frame1 (old new s))  
(define-record frame2 (v1))  
  
(define make-identity  
  (lambda ()  
    (cons (make-lastframe) '())))  
  
(define make-subst1  
  (lambda (old new s k)  
    (cons (make-frame1 old new s) k)))  
  
(define make-subst2  
  (lambda (v1 k)  
    (cons (make-frame2 v1) k)))
```

## List of Frames Representation of Continuations

---

```
(define apply-continuation
  (lambda (k x)
    (let ((frame (car k))(k (cdr k)))
      (record-case frame
        (frame1 (old new s)
          (let ((v1 x))
            (subst-cps old new (cdr s) (make-subst2 v1 k))))
        (frame2 (v1)
          (let ((v2 x))
            (apply-continuation k (cons v1 v2))))
        (lastframe () x)
        (else (error "bad continuation"))))))))
```

## List of Frames without Endmarker

---

Same as list-of-frames, except

```
(make-identity) = '()
(define make-identity (lambda () '()))

(define apply-continuation
  (lambda (k x)
    (if (null? k) ; annoying to have to test this first!
        x
        (let ((frame (car k))(k (cdr k)))
          (record-case frame
            (frame1 (old new s)
              (let ((v1 x))
                (subst-cps old new (cdr s)
                           (make-subst2 v1 k))))
            (frame2 (v1)
              (let ((v2 x))
                (apply-continuation k
                                     (cons v1 v2))))
            (else (error "bad continuation"))))))))
```

## Unframed List

---

Like list-of-frames, but without interior sublists

```
(make-identity) = '(lastframe)
```

```
(make-subst1 old new s k) = '(frame1 ,old ,new ,s . ,k)
```

```
(make-subst2 v1 k) = '(frame2 ,v1 . ,k)
```

Typical continuation looks like a list of items

```
(frame1 <o> <n> <s_1> frame2 <v_1> frame2 <v_2> ... lastframe)
```

## Unframed List

---

```
(define make-identity
  (lambda ()
    '(lastframe)))
```

```
(define make-subst1
  (lambda (old new s k)
    (cons 'frame1 (cons old (cons new (cons s k))))))
```

```
(define make-subst2
  (lambda (v1 k)
    (cons 'frame2 (cons v1 k))))
```

## Unframed List

---

```
(define apply-continuation
  (lambda (k x)
    (case (car k)
      ((frame1) (let ((old (cadr k))
                     (new (caddr k))
                     (s (caddr k))
                     (k (cddddr k)))
                 (let ((v1 x))
                   (subst-cps old new (cdr s)
                             (make-subst2 v1 k))))
                ((frame2) (let ((v1 (cadr k))
                               (k (caddr k)))
                          (let ((v2 x))
                            (apply-continuation k (cons v1 v2))))))
      ((lastframe) x)
      (else (error "bad continuation"))))))
```

## Stack Representation of Continuations

---

Will start with unframed-stack representation, and represent the continuation as a real stack: an array and a pointer.

Will represent the stack as a Scheme vector (equivalent to an array: a randomly-accessible mutable data structure).

# Stack Representation of Continuations

---

```
(define make-stack
  (lambda (size)
    (list '*stack -1 (make-vector size))))
```

```
(define stack->top cadr)
(define stack->data caddr)
```

```
(define stack-ref
  (lambda (stack ptr)
    (vector-ref (stack->data stack) ptr)))
```

```
(define stack-set-val!      ;; works by side-effect; convenient!
  (lambda (stack ptr val) ;; returns a pointer to the new stack
    (vector-set! (stack->data stack) ptr val)))
```

```
(define stack-set-top!     ;; works by side-effect; convenient!
  (lambda (stack ptr)      ;; returns a pointer to the new stack
    (set-car! (cdr stack) ptr)))
```

## Stack Representation of Continuations

---

```
(define stack-push
  (lambda (val stack)
    (let ((ptr (stack->top stack)))
      (stack-set-top! stack (+ ptr 1))
      (stack-set-val! stack (+ ptr 1) val)
      stack)))
```

```
(define stack-pop
  (lambda (stack n)
    (stack-set-top! stack (- (stack->top stack) n))
    stack))
```

# Debugging

---

Now that we are doing pointer arithmetic, some decent debugging tools are a must:

```
(define debug-print
  (lambda (stack)
    (if (eq? (car stack) '*stack)
        (begin
          (printf "sp = ~s~%" (stack->top stack))
          (let loop ((sp (stack->top stack)) (n 10))
            (if (or (< sp 0) (zero? n))
                (printf "~%")
                (begin
                  (printf "stack[~s] = ~s~%" sp
                          (stack-ref stack sp))
                  (loop (- sp 1) (- n 1))))))
          (printf "Warning: bad stack!~%stack = ~s~%" stack))))
```

## Continuations as Arrays

---

Now we can alter the unframed-stack representation to use this array representation. Note that we are always using the *same* array, but we are modifying it destructively. This works just so long as we have only one continuation active at any time. If not, this will fail dramatically.

```
(define make-identity
  (lambda ()
    (let ((stack (make-stack 200)))      ; use some suitable size
      (stack-push 'lastframe stack))))
```

## Continuations as Arrays

---

```
(define make-subst1
  (lambda (old new s k)
    (stack-push 'frame1           ; just change cons to stack-push
      (stack-push old
        (stack-push new
          (stack-push s k)))))))
```

```
(define make-subst2
  (lambda (v1 k)
    (stack-push 'frame2
      (stack-push v1 k))))
```

## Continuations as Arrays

---

```
(define apply-continuation
  (lambda (k x)                                ; k is a stack
    (debug-print k)
    (let ((sp (stack->top k)))
      (case (stack-ref k sp)
        ((frame1) (let ((old (stack-ref k (- sp 1)))
                        (new (stack-ref k (- sp 2)))
                        (s (stack-ref k (- sp 3)))
                        (k (stack-pop k 4)))
                    (let ((v1 x))
                      (subst-cps old new (cdr s)
                                (make-subst2 v1 k))))))
        ((frame2) (let ((v1 (stack-ref k (- sp 1)))
                        (k (stack-pop k 2)))
                    (let ((v2 x))
                      (apply-continuation k
                                (cons v1 v2))))))
        ((lastframe) x)
        (else (error "bad continuation"))))))
```

## Testing

---

Here is a run, with a snapshot at every invocation of `apply-continuation`.

```
> (subst 'foo 'bar '((a foo) foo))
```

stack[10]		frame1	
stack[9]		foo	
stack[8]	frame1	bar	frame2
stack[7]	foo	(foo)	bar
stack[6]	bar	frame2	frame2
stack[5]	(a foo)	a	a
stack[4]	frame1	frame1	frame1
stack[3]	foo	foo	foo
stack[2]	bar	bar	bar
stack[1]	((a foo) foo)	((a foo) foo)	((a foo) foo)
stack[0]	lastframe	lastframe	lastframe

# Testing

---

stack[6]	frame2		frame1
stack[5]	a		foo
stack[4]	frame1	frame1	bar
stack[3]	foo	foo	(foo)
stack[2]	bar	bar	frame2
stack[1]	((a foo) foo)	((a foo) foo)	(a bar)
stack[0]	lastframe	lastframe	lastframe

stack[4]	frame2		
stack[3]	bar		
stack[2]	frame2	frame2	
stack[1]	(a bar)	(a bar)	
stack[0]	lastframe	lastframe	lastframe

((a bar) bar)

# Clever Representation of Continuations

---

Original:

```
(define fact
  (lambda (n)
    (if (zero? n) 1
        (* n (fact (- n 1))))))
```

CPS version:

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        (k 1)
        (fact-cps (- n 1)
                  (lambda (v) (k (* n v)))))))
```

```
(define fact
  (lambda (n)
    (fact-cps (lambda (v) v))))
```

## Continuations for fact

---

*Continuation* ::= (lambda (v) v)  
                  ::= (lambda (v) (*Continuation* (\* n v)))

(make-identity) = [(lambda (v) v)]  
(make-fact1 n [k]) = [(lambda (v) (k (\* n v)))]

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        (apply-cont k 1)
        (fact-cps (- n 1) (make-fact1 n k)))))
```

```
(define fact
  (lambda (n)
    (fact-cps n (make-identity))))
```

## Key Observation

---

Claim: Every fact continuation is of the form

```
(lambda (v) (* p v))
```

for some p.

```
(lambda (v) v) = (lambda (v) (* 1 v))
```

If  $k = (\text{lambda } (v) (* p v))$

then

```
(make-fact1 n k) = (lambda (v) (k (* n v)))  
                  = (lambda (v) ((lambda (w) (* p w))  
                                   (* n v)))  
                  = (lambda (v) (* p (* n v)))  
                  = (lambda (v) (* (* p n) v))
```

## Using the Key Observation

---

So we can represent  $(\text{lambda } (v) (* p v))$  by just the number  $p$ . (Neat!)

```
(define make-identity
  (lambda () 1))
```

```
(define make-fact1
  (lambda (n p)      ; p      repr (lambda (v) (* p v))
    (* n p)))      ; (* n p) repr (lambda (v) (* (* n p) v))
```

```
(define apply-cont
  (lambda (p v)
    (* p v)))
```

## Using the Key Observation

---

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        (apply-cont k 1)
        (fact-cps (- n 1) (make-fact1 n k)))))
(define fact
  (lambda (n) (fact-cps n (make-identity))))
```

Inlining these definitions in `fact-cps` we get:

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        k ; = (* k 1) = (apply-cont k 1)
        (fact-cps (- n 1) (* n k)))))
(define fact ;; Wow! It's fact-iter
  (lambda (n) (fact-cps n 1)))
```

## From First-Order Form to Imperative Form

---

Now we have tail-form, first-order (no first-class functions, just data structures) program. Want to convert it to flowchart. Will use idea of lambda-variables as registers: Replace

```
(define foo
  (lambda (x y) ..body..))
...
(foo E1 E2)
```

by

```
foo: ..body..    % take input from registers x and y
```

```
x := E1;        % be careful about overwriting registers here!
y := E2;
goto foo;
```

Will do this using Scheme instead of pseudocode.

# From First-Order Form to Imperative Form

---

```
;; cps version; uses abstract syntax tree representation
```

```
(define-record identity ())
```

```
(define-record rem1 (lsym k))
```

```
(define remove-all
```

```
  (lambda (a lsym)
```

```
    (remove-all-cps a lsym (make-identity))))
```

```
(define remove-all-cps
```

```
  (lambda (a lsym k)
```

```
    (cond ((null? lsym)
```

```
          (apply-continuation k '()))
```

```
          ((eq? a (car lsym))
```

```
            (remove-all-cps a (cdr lsym) k))
```

```
          (else
```

```
            (remove-all-cps a (cdr lsym)
```

```
              (make-rem1 lsym k))))))
```

## From First-Order Form to Imperative Form

---

```
(define apply-continuation
  (lambda (k v)
    (record-case k
      (identity () v)
      (rem1 (lsym k1)
            (apply-continuation k1 (cons (car lsym) v))))))
```

# Register Machine Version

---

```
(define remove-all
  (lambda (a lsym)
    (let ((a a) (lsym lsym)
          (k (make-identity))
          (v '*unbound*))
      (letrec
        ((remove-all-cps
          (lambda ()
            (cond ((null? lsym)
                   (set! v '()))
                  (apply-continuation))
              ((eq? a (car lsym))
               (set! lsym (cdr lsym))
               (remove-all-cps))
              (else
               (set! k (make-rem1 lsym k))
               (set! lsym (cdr lsym))
               (remove-all-cps))))))
```

## Register Machine Version

---

```
(define remove-all
  ...
  (apply-continuation
    (lambda ()
      (record-case k
        (identity () v)
        (rem1 (lsym k1)
              (set! k k1)
              (set! v (cons (car lsym) v))
              (apply-continuation))))))
  (remove-all-cps))))
```

## subst

---

Now let's do it for subst. Let's take the array representation:

```
(define subst
  (lambda (old new s)
    (subst-cps old new s (make-identity))))
```

```
(define subst-cps
  (lambda (old new s k)      ; k is a representation
    (if (pair? s)
        (subst-cps old new (car s)
                    (make-subst1 old new s k))
        (if (eq? s old)
            (apply-continuation k new)
            (apply-continuation k s))))))
```

## subst

---

```
(define make-identity
  (lambda ()
    (let ((stack (make-stack 200))) ; use some suitable size
      (stack-push 'lastframe stack))))

(define make-subst1
  (lambda (old new s k)
    (stack-push 'frame1 ; just change cons to stack-push
      (stack-push old
        (stack-push new
          (stack-push s k)))))))

(define make-subst2
  (lambda (v1 k)
    (stack-push 'frame2 (stack-push v1 k))))
```

## subst

---

```
(define apply-continuation
  (lambda (k x)                                ; k is a stack
    (let ((sp (stack->top k))
          (ins (stack-ref k sp)))
      (case ins
        ((frame1) (let ((old (stack-ref k (- sp 1)))
                        (new (stack-ref k (- sp 2)))
                        (s (stack-ref k (- sp 3)))
                        (k (stack-pop! k 4)))
                    (let ((v1 x))
                      (subst-cps old new (cdr s)
                                (make-subst2 v1 k))))))
        ((frame2) (let ((v1 (stack-ref k (- sp 1)))
                        (k (stack-pop! k 2)))
                    (let ((v2 x))
                      (apply-continuation k
                                (cons v1 v2))))))
        ((lastframe) x)
        (else (error "bad continuation"))))))
```

## subst

---

```
(define subst
  (lambda (old-arg new-arg s-arg)
    ;; (subst-cps old new (make-identity))
    (set! old old-arg)
    (set! new new-arg)
    (set! k (make-identity))
    (subst-cps)))
```

## subst

---

```
(define subst-cps
  (lambda ()
    ; uses old, new, s, k
    (if (pair? s)
        ;; (subst-cps old new (car s))
        ;; (make-subst1 old new s k))
        (begin
          (set! k (make-subst1 old new s k))
          (set! s (car s))
          (subst-cps))
        (if (eq? s old)
            (begin
              ;; (apply-continuation k new)
              (set! x new)
              (apply-continuation))
            (begin
              ;; (apply-continuation k s)
              (set! x s)
              (apply-continuation))))))
```

## subst

---

```
(define apply-continuation
  (lambda () ; uses k, x
    (let ((sp (stack->top k))
          (ins (stack-ref k sp)))
      (case ins
        ((frame1) (let ;; need different local names
                     ((old1 (stack-ref k (- sp 1)))
                      (new1 (stack-ref k (- sp 2)))
                      (s1 (stack-ref k (- sp 3)))
                      (k1 (stack-pop! k 4)))
                     (let ((v1 x))
                         ;;(subst-cps old new (cdr s))
                         ;; (make-subst2 v1 k))
                         (set! old old1)
                         (set! new new1)
                         (set! s (cdr s1))
                         (set! k (make-subst2 v1 k1))
                         (subst-cps))
```

## subst

---

```
(define apply-continuation
  ...
  ((frame2) (let
              ((v1 (stack-ref k (- sp 1)))
               (k1 (stack-pop! k 2)))
              (let ((v2 x))
                  ;; (apply-continuation k (cons v1 v2))
                  (set! x (cons v1 v2))
                  (set! k k1)
                  (apply-continuation)))
              ((lastframe) x)
              (else (error "bad continuation"))))))
```

## Chapter 8: Flow Analysis

---

## Example of Method Inlining (1/3)

---

Method inlining is used by all Java virtual machines, by bytecode compaction tools such as JAX, etc.

```
A x = new A();
B y = new B();

x.m( new Q() );
y.m( new S() );

class A {
    void m(Q arg) {
        arg.p();
    }
}
class B extends A {
    void m(Q arg) {...}
}

class Q {
    void p() {...}
}
class S extends Q {
    void p() {...}
}
```

## Example of Method Inlining (2/3)

---

```
A x = new A();  
B y = new B();
```

```
x.m( new Q() );
```

```
y.m( new S() );
```

```
class A {  
    void m(Q arg) {  
        arg.p();  
    }  
}  
class B extends A {  
    void m(Q arg) {...}  
}
```

```
class Q {  
    void p() {...}  
}  
class S extends Q {  
    void p() {...}  
}
```

## Example of Method Inlining (3/3)

---

```
A x = new A();  
B y = new B();  
  
new Q().p();  
y.m( new S() );
```

```
class A {  
    void m(Q arg) {  
        arg.p();  
    }  
}  
  
class B extends A {  
    void m(Q arg) {...}  
}
```

```
class Q {  
    void p() {...}  
}  
  
class S extends Q {  
    void p() {...}  
}
```

## Benchmark Characteristics

---

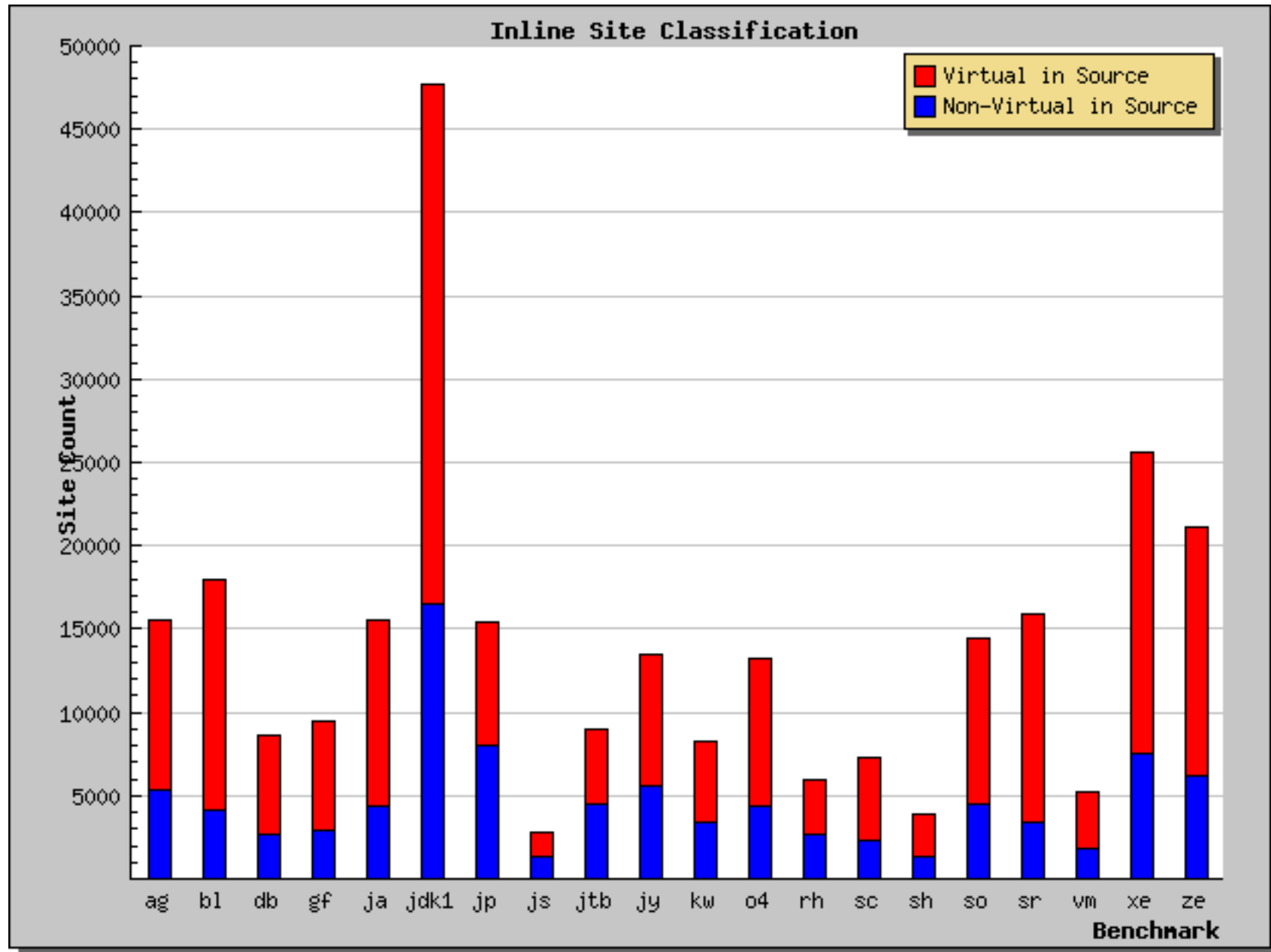
The Purdue Java benchmark suite:

100,000 class files corresponding to  
10,000,000 lines of Java source code.

In this note, I will concentrate on:

27,000 class files corresponding to  
2,300,000 lines of Java source code.

# How many calls can be inlined?



## Enabling technology: flow analysis

---

Goal: find call sites with unique callee.

Set-based analysis: the flow set for an expression is a set of class names.

Idea: flow set for  $e = \{ A, B, C \}$  means

$e$  will evaluate to either an A-object, a B-object, or a C-object

**CHA** → **TSMI** → **0-CFA**



**cost and accuracy**

Any flow analysis must be approximative. Too large sets are sound and conservative, but very large sets are useless. How precise information can we get? How fast can we compute it?

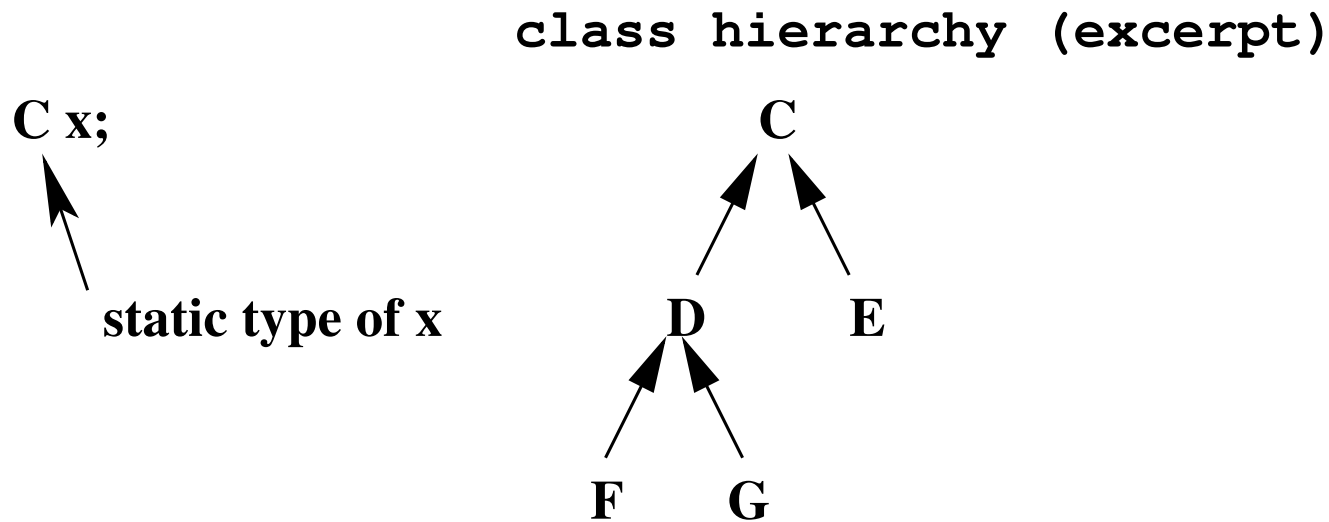
# Class Hierarchy Analysis (CHA)

---

CHA relies on the type system; it is a type-based analysis.

[Dean, Grove, Chambers ECOOP 1995]

Flow set = all subtypes of the static type



**Flow set for x = { C, D, E, F, G }**

## Example of Class Hierarchy Analysis (1/2)

---

```
A x = new A();
B y = new B();

x.m( new Q() );
y.m( new S() );

class A {
    void m(Q arg) {
        arg.p();
    }
}
class B extends A {
    void m(Q arg) {...}
}

class Q {
    void p() {...}
}
class S extends Q {
    void p() {...}
}
```

## Example of Class Hierarchy Analysis (2/2)

---

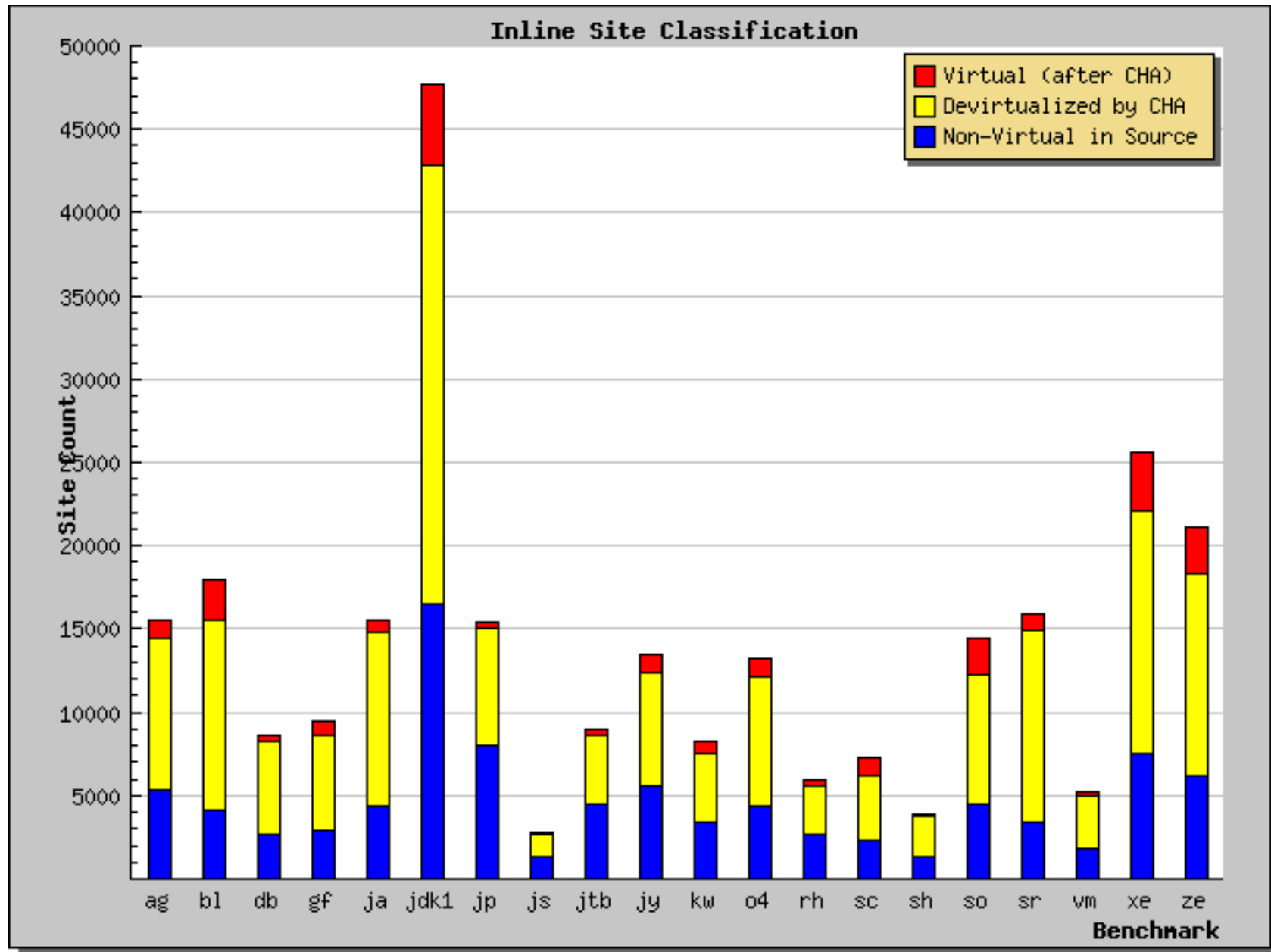
```
A x = new A();
B y = new B();

x.m( new Q() );
y.m( new S() );
```

```
class A {
    void m(Q arg) {
        arg.p();
    }
}
class B extends A {
    void m(Q arg) {...}
}

class Q {
    void p() {...}
}
class S extends Q {
    void p() {...}
}
```

# How good is CHA?



## 0-CFA

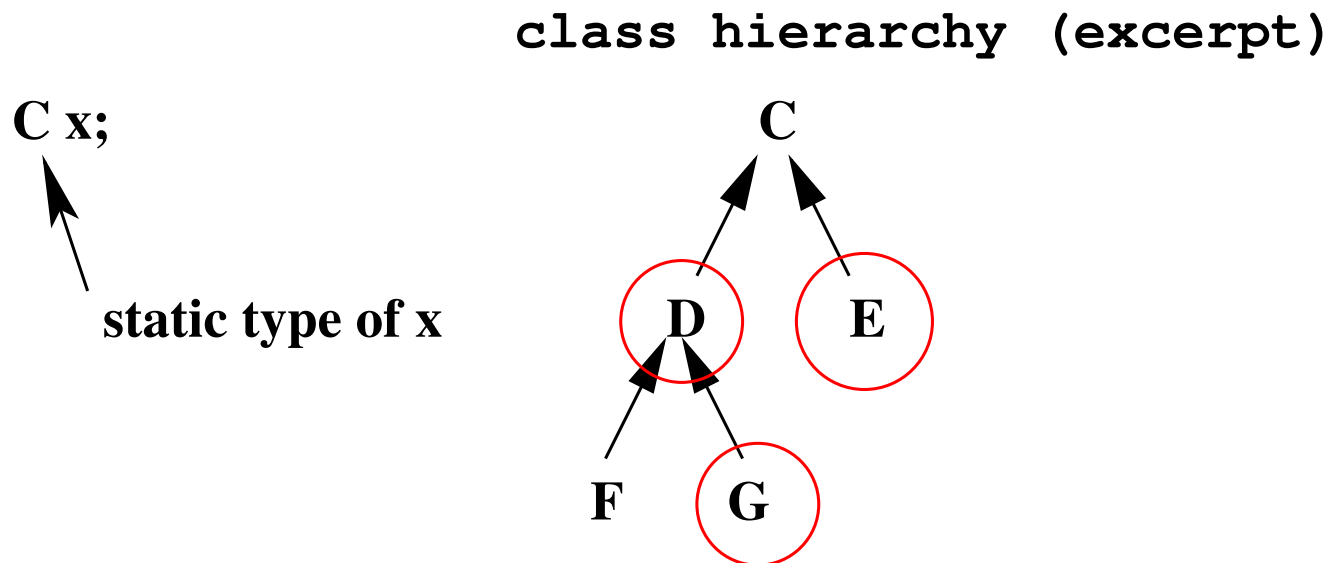
---

Flow insensitive, context insensitive flow analysis.

[Palsberg et al. OOPSLA 1991, ECOOP 1992]

Called 0-CFA by Shivers in 1992. Can be done in  $O(n^3)$  time.

Note: 0-CFA does not rely on the type system, but is type respecting.



**Flow set for x = { D, E, G }**

## 0-CFA Constraints

---

Approach: (1) generate constraints, (2) solve constraints.

For each expression  $e$ , there is a flow variable  $[e]$ .

### Program

### Constraints

`new C()`

$C \in [\text{new C}()]$

`x = e;`

$[x] \supseteq [e]$

`e1.m(e2)`

$C \in [e_1] \Rightarrow [e_2] \subseteq [a]$

and

$C \in [e_1] \Rightarrow [e] \subseteq [e_1.m(e_2)]$

`class C {`

...

`B m(A a) {`

...

`return e;`

`}`

`}`

## Example of 0-CFA (1/2)

---

```
A x = new A();
B y = new B();

x.m( new Q() );
y.m( new S() );

class A {
    void m(Q arg) {
        arg.p();
    }
}
class B extends A {
    void m(Q arg) {...}
}

class Q {
    void p() {...}
}
class S extends Q {
    void p() {...}
}
```

$A \in \llbracket \text{new } A() \rrbracket$

$B \in \llbracket \text{new } B() \rrbracket$

$Q \in \llbracket \text{new } Q() \rrbracket$

$S \in \llbracket \text{new } S() \rrbracket$

$\llbracket \text{new } A() \rrbracket \subseteq \llbracket x \rrbracket$

$\llbracket \text{new } B() \rrbracket \subseteq \llbracket y \rrbracket$

$A \in \llbracket x \rrbracket \Rightarrow \llbracket \text{new } Q() \rrbracket \subseteq \llbracket A.\text{arg} \rrbracket$

$B \in \llbracket x \rrbracket \Rightarrow \llbracket \text{new } Q() \rrbracket \subseteq \llbracket B.\text{arg} \rrbracket$

$A \in \llbracket y \rrbracket \Rightarrow \llbracket \text{new } S() \rrbracket \subseteq \llbracket A.\text{arg} \rrbracket$

$B \in \llbracket y \rrbracket \Rightarrow \llbracket \text{new } S() \rrbracket \subseteq \llbracket B.\text{arg} \rrbracket$

## Example of 0-CFA (2/2)

---

```
A x = new A();  
B y = new B();
```

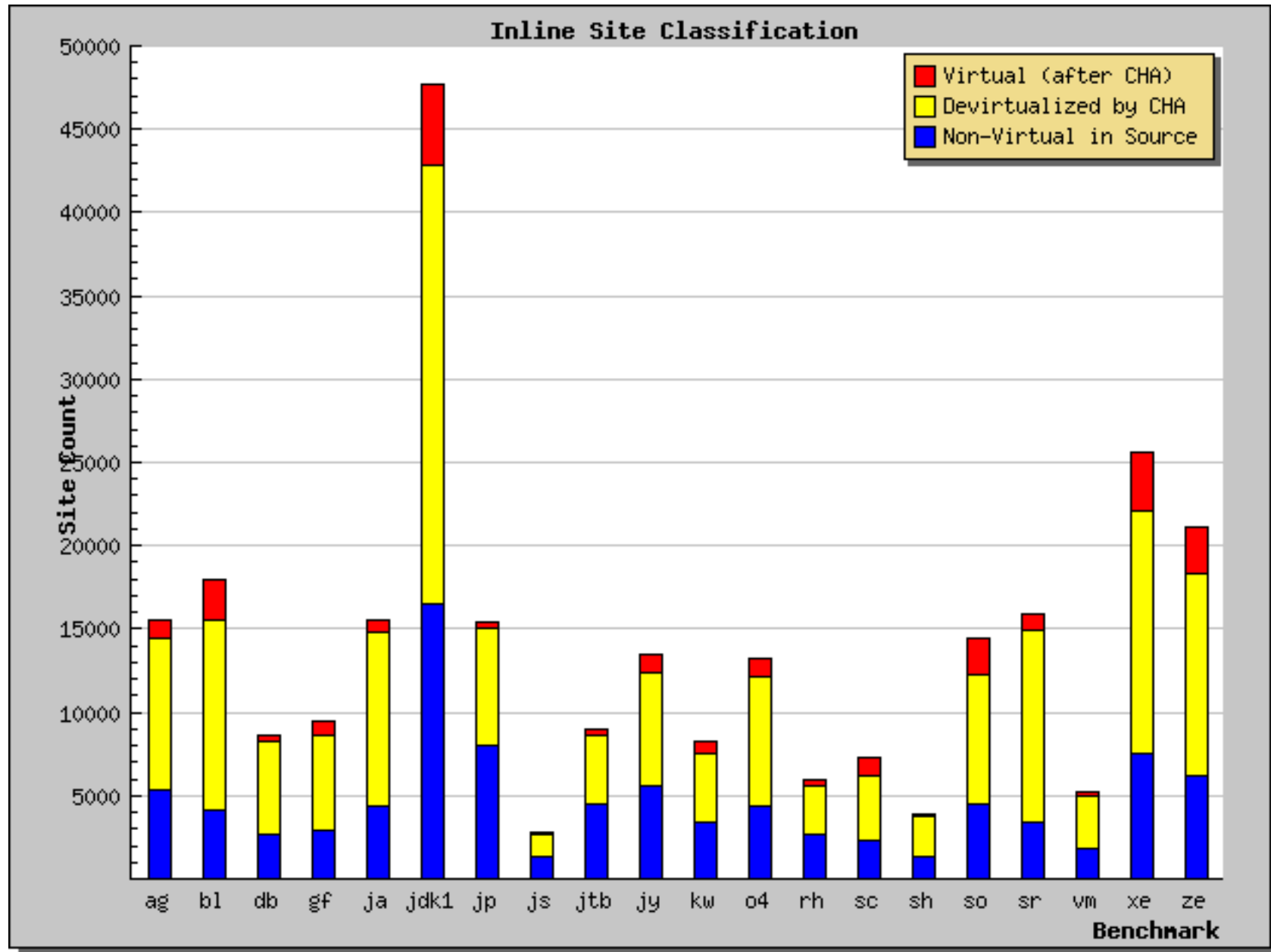
```
x.m( new Q() );
```

```
y.m( new S() );
```

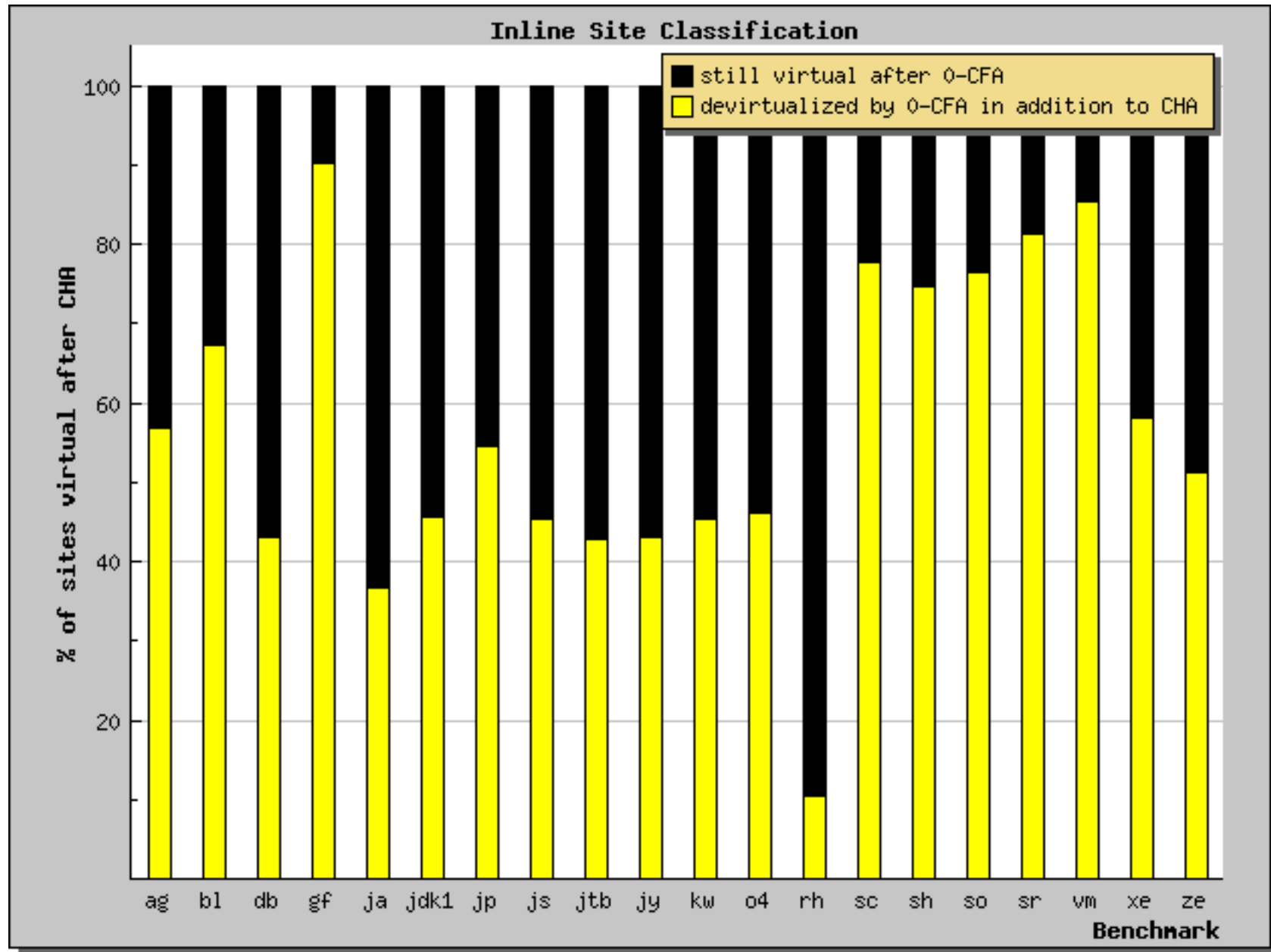
```
class A {  
    void m(Q arg) {  
        arg.p();  
    }  
}  
class B extends A {  
    void m(Q arg) {...}  
}
```

```
class Q {  
    void p() {...}  
}  
class S extends Q {  
    void p() {...}  
}
```

# How good is CHA? (revisited)

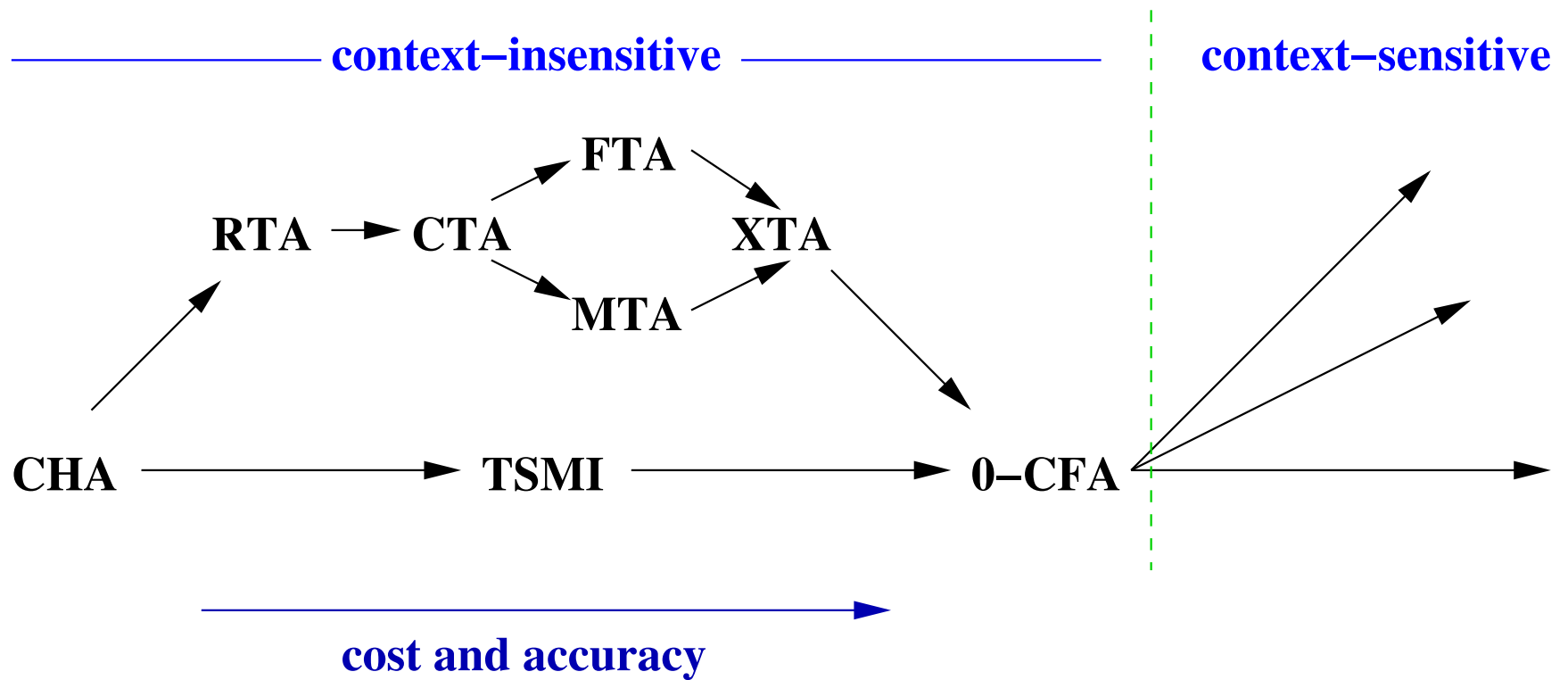


# How good is 0-CFA?



# Is there a middle ground?

---



[Tip and Palsberg, OOPSLA 2000]

See also [Hendren et al, OOPSLA 2000]

## World-Assumptions

---

- The *closed-world* assumption:

All parts of the program are known at compile-time. A compile-time analysis is for the whole program. Example: flow analysis like the ones to be presented in this note.

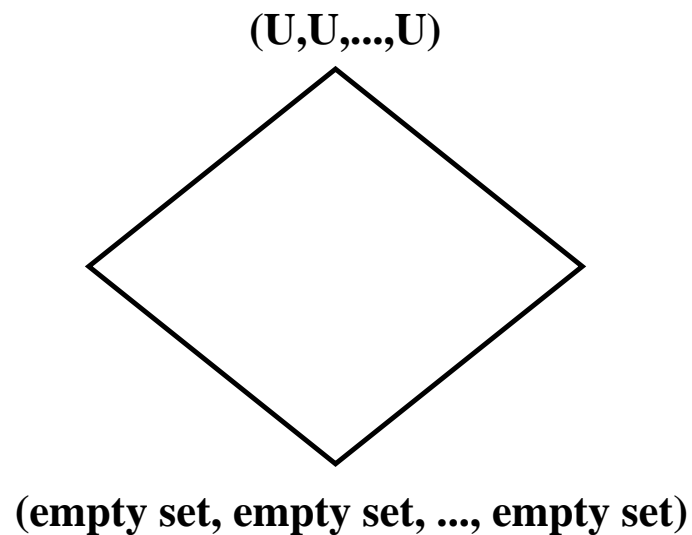
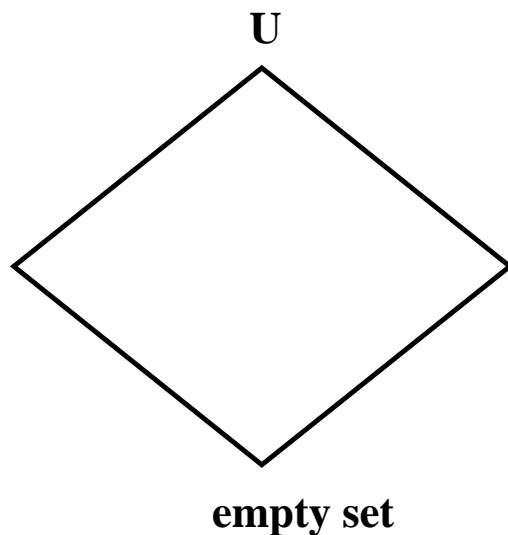
- The *open-world* assumption:

Some parts of the program are unknown at compile-time. A compile-time analysis is local for each fragment of the program. Example: inference of principal types.

# The Space of Flow Sets

---

- Each flow variable ranges over sets of classes.
- Because of the closed-world assumption, the maximal set of classes  $\mathbf{U}$  is finite.
- So, the space of flow sets is a powerset of a finite set of classes.
- A powerset is a lattice; lattices have good mathematical properties.
- The top of the lattice corresponds to trivial flow information.



## Flow Constraints

---

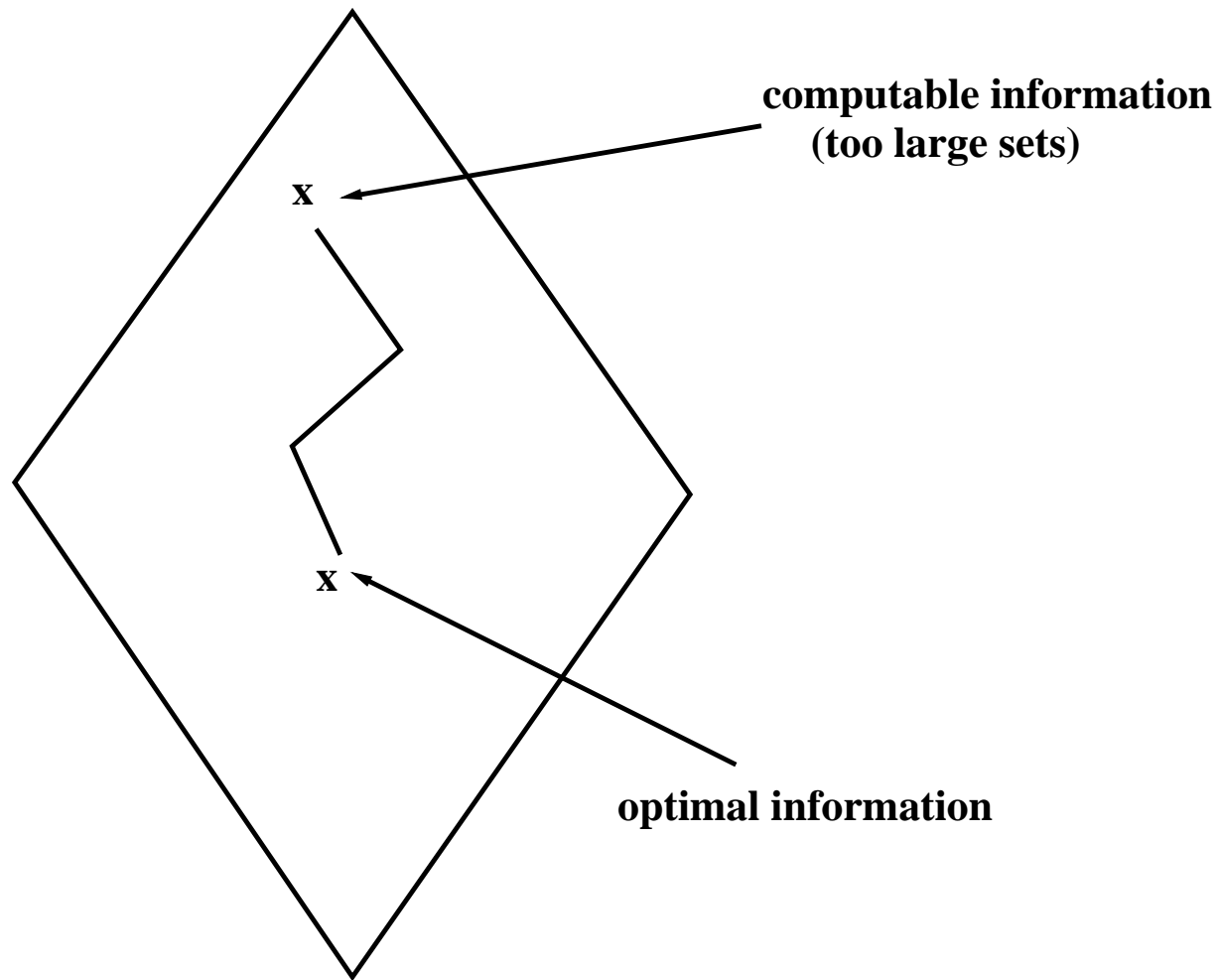
- We will work with flow constraints of three forms:

$$\begin{array}{lll} c \in X & \textit{start constraints} \\ X \subseteq Y & \textit{propagation constraints} \\ (c \in X) \Rightarrow (Y \subseteq Z) & \textit{conditional constraints} \end{array}$$

- There is always a unique minimal solution.
- The minimal solution can be computed in worst-case cubic time.

# The Minimal Solution is above the Optimal Information

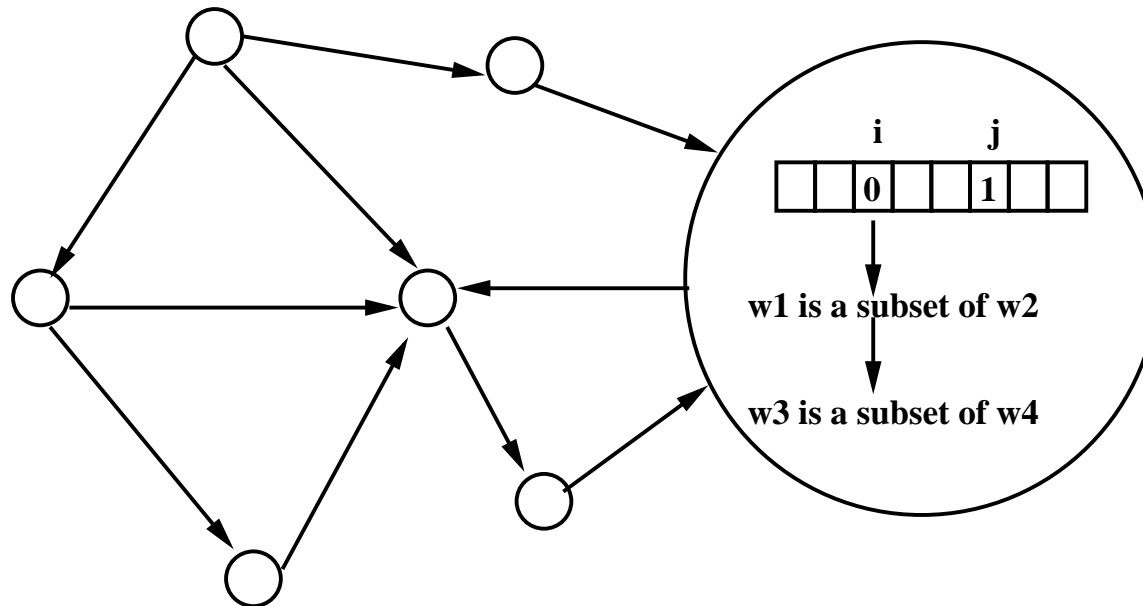
---



# The Constraint Solver

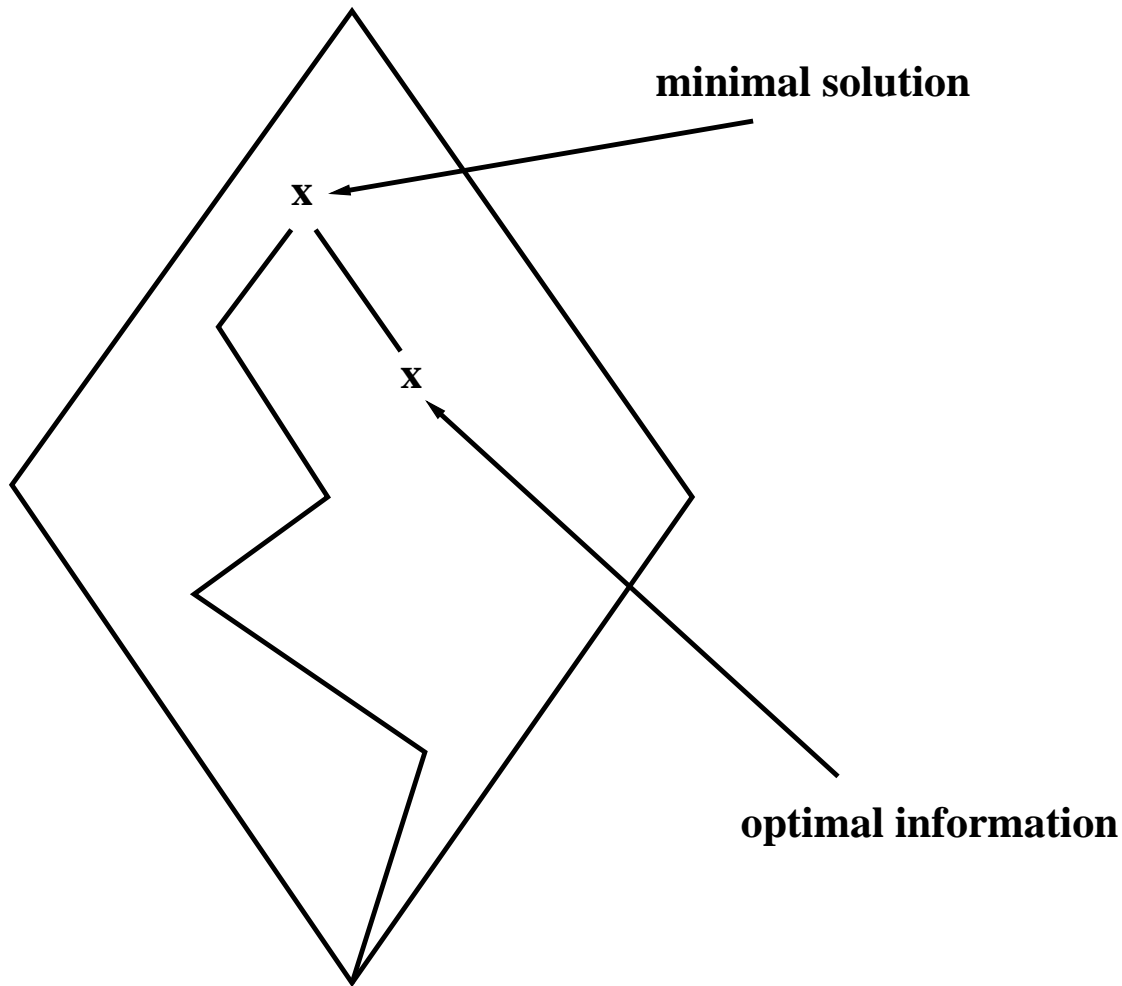
---

- Accepts constraints one at the time; maintains the minimal solution.
- The internal data structure is a graph; one node per flow variable.
- An edge  $v \rightarrow w$  implies  $v \subseteq w$ .
- The value of a flow variable  $X$  is recorded in a bit vector  $B(X)$ .
- Conditional constraints are tied to the relevant bit.
- $K(X, i) = \text{list of pending constraints for bit } i \text{ in node } X$



# Iterating from the Bottom towards the Minimal Solution

---



# Solver Operations

---

INSERT( $i \in X$ ) =

    PROPAGATE( $X, i$ )

INSERT( $X \subseteq Y$ ) =

    create an edge  $X \rightarrow Y$ ;

**for**  $i \in B(X)$  **do** PROPAGATE( $Y, i$ ) **end**

INSERT( $c \in X \Rightarrow Y \subseteq Z$ ) =

**if**  $B(X, c)$

**then** INSERT( $Y \subseteq Z$ )

**else**  $K(X, c) = K(X, c) \cup (Y \subseteq Z)$

**end**

PROPAGATE( $v, i$ ) =

**if**  $\neg B(v, i)$  **then**

$B(v, i) = \text{true}$ ;

**for**  $v \rightarrow w$  **do** PROPAGATE( $w, i$ ) **end**;

**for**  $k \in K(v, i)$  **do** INSERT( $k$ ) **end**;

$K(v, i) = ()$

**end**

# An Amortized Analysis

---

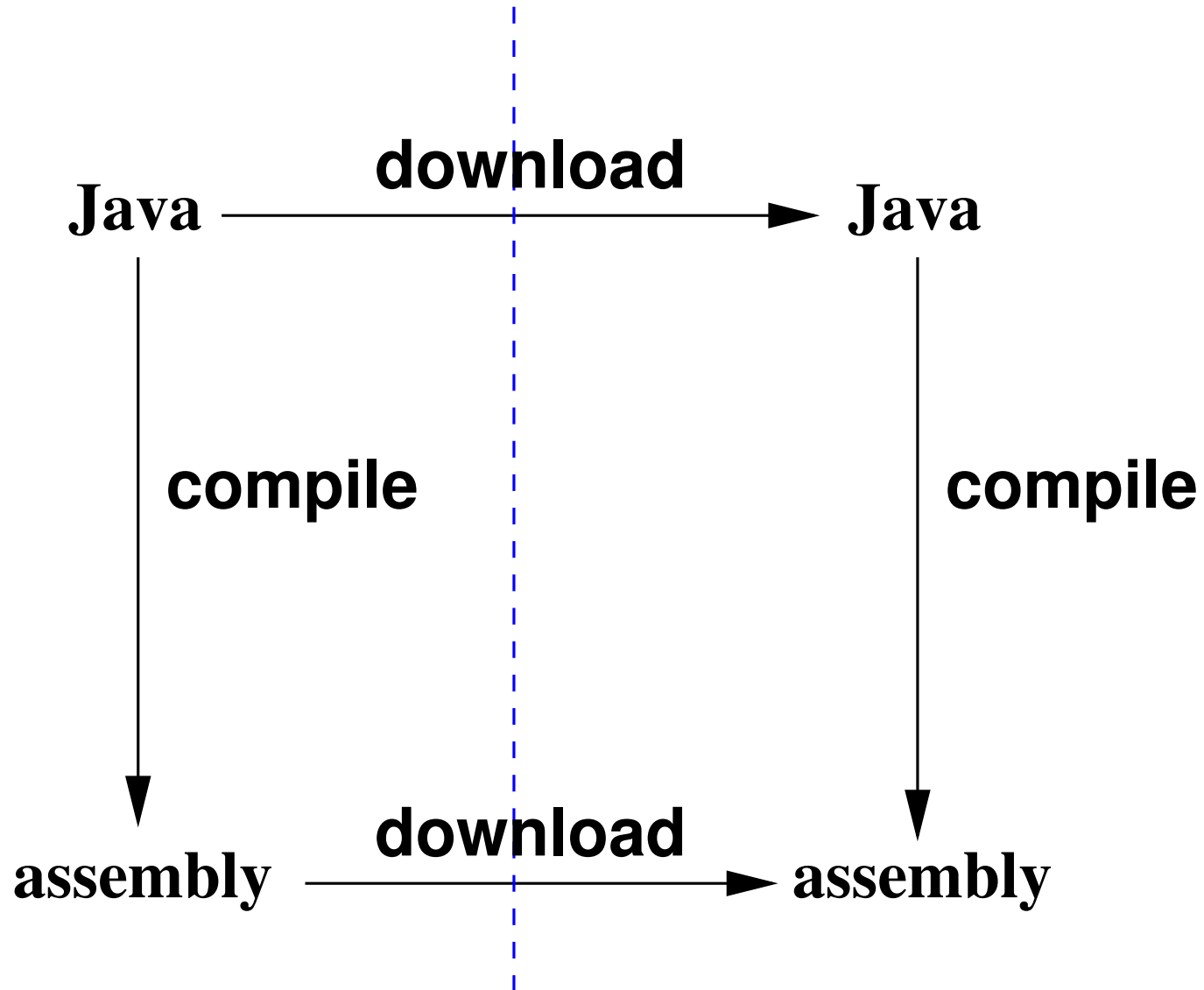
- The size of the program:  $n$ .
- Number of classes:  $O(n)$ .
- Number of nodes:  $O(n)$ .
- Number of edges:  $O(n^2)$ .
- Number of constraints:  $O(n) \times O(n) = O(n^2)$ .
- Each activity “uses up” some part of the graph.
- Each bit is propagated along a specific edge at most once.
- Since there are  $O(n)$  classes and  $O(n^2)$  edges, this totals  $O(n^3)$ .
- Each of the  $O(n^2)$  constraints may: (1) be inserted into and deleted from a list once and (2) cause the creation of a single edge,
- Which is  $O(1)$  for each constraint.
- The total cost is thus  $O(n^3)$ .
- In practice, locality in programs gives more like linear behavior.

## Chapter 9: Type-Safe Method Inlining

---

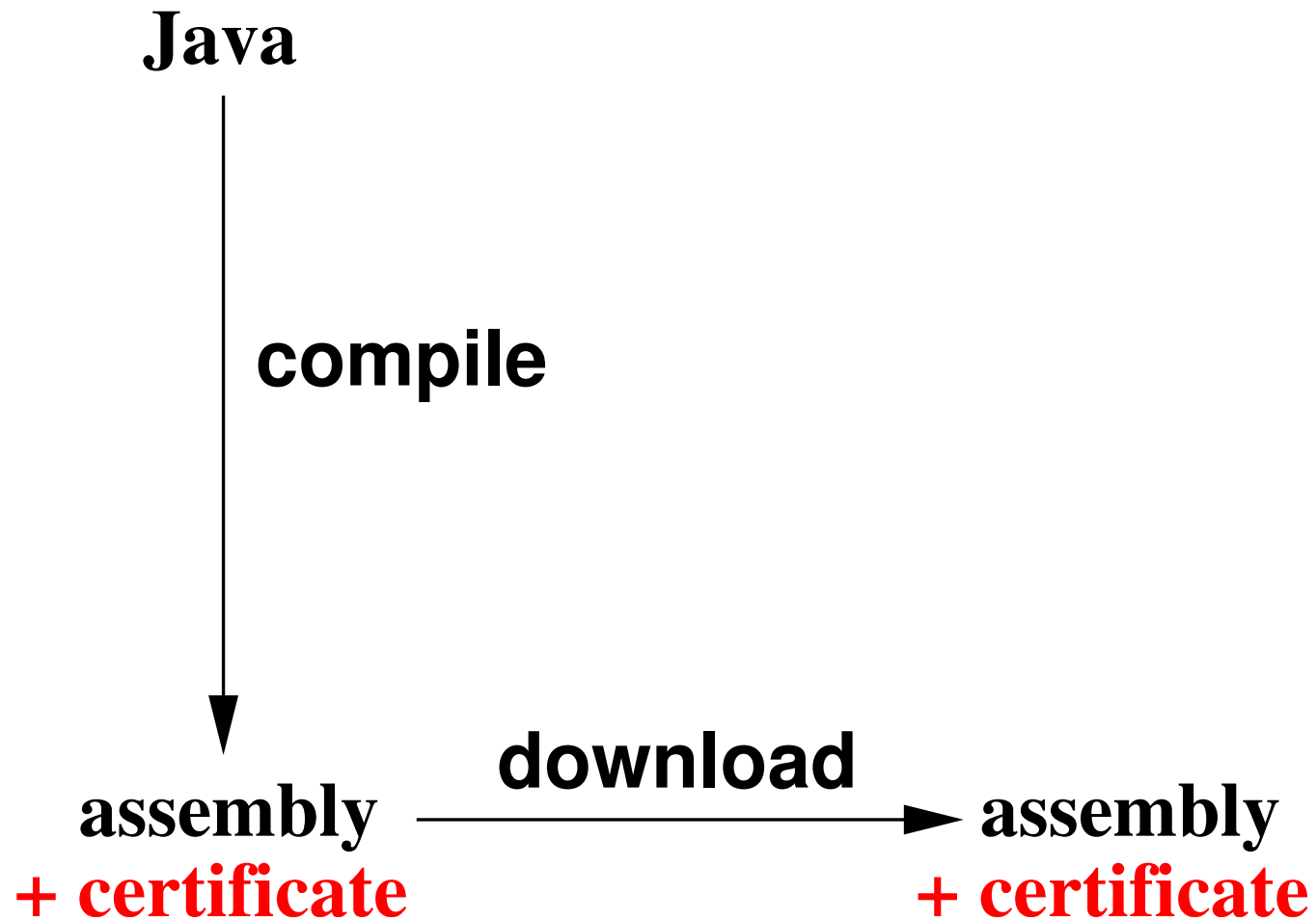
# Compilers

---



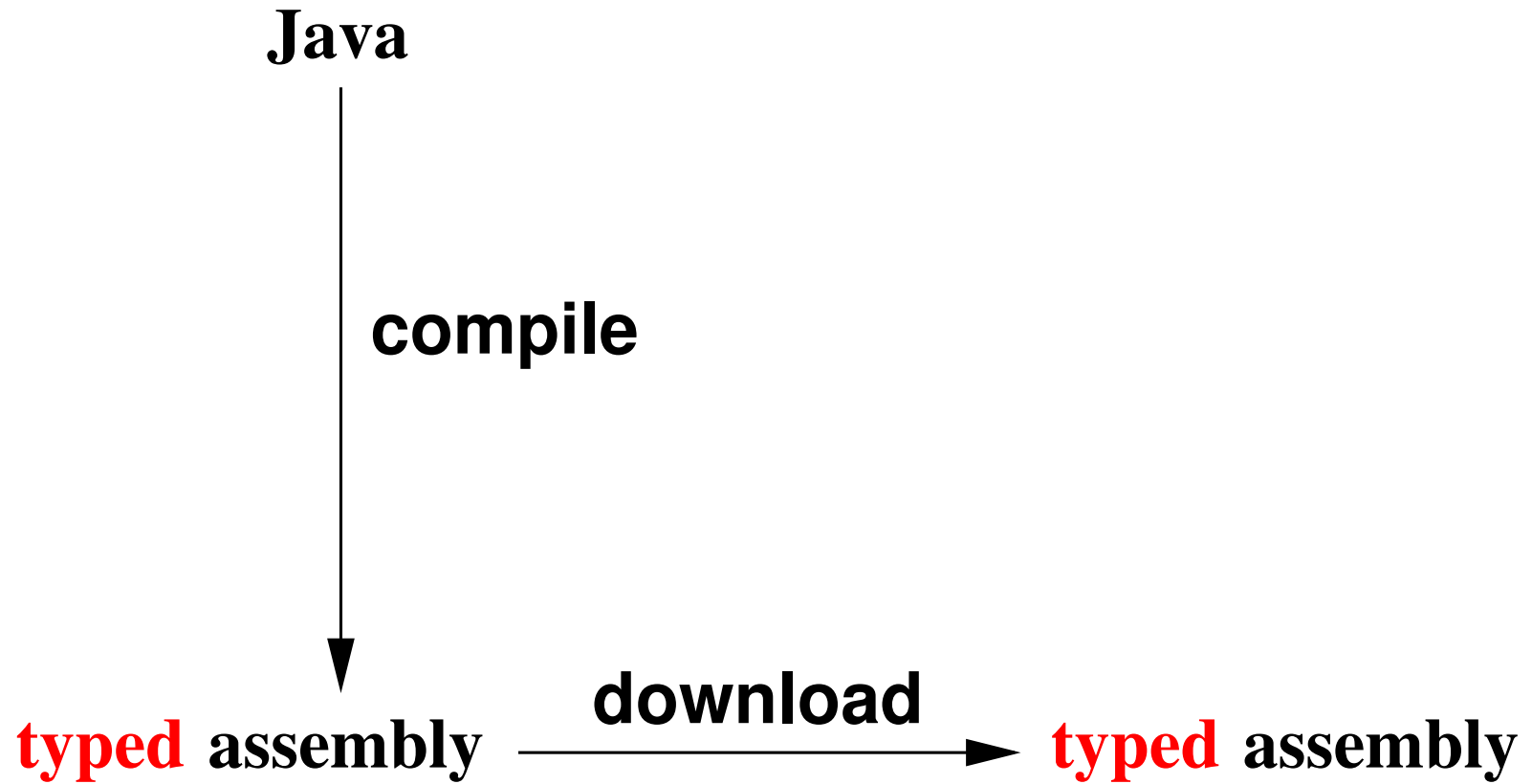
# Certifying Compilers

---



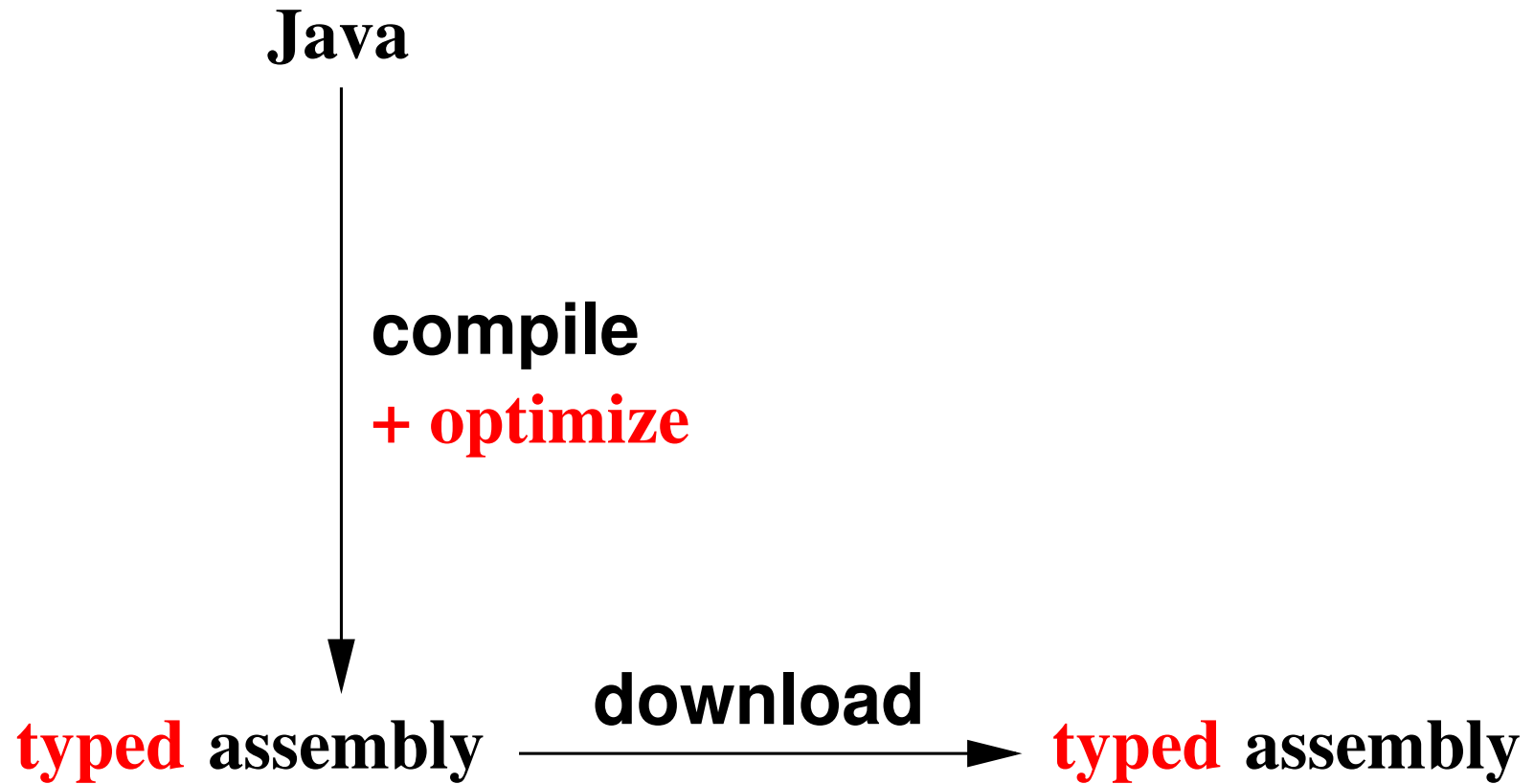
# Type-Preserving Compilers

---



# Certifying Optimizing Compilers

---



# Typed Intermediate Languages

[Modern Compiler Implementation in Java,  
by Appel and Palsberg, 2002]

[Palsberg, tutorial at PLDI 2002]

Types help debugging the compiler.

Project in my undergraduate compiler course:



**MiniJava**



**Piglet**



**Spiglet**



**Kanga**



**MIPS**

# Our Results

---

New approach to method inlining:

- New flow analysis: TSML.
- Key idea 1: align the flow analysis with the Java type system.
- Key idea 2: change the type annotations.

Result: method inlining is type safe.

Theorems

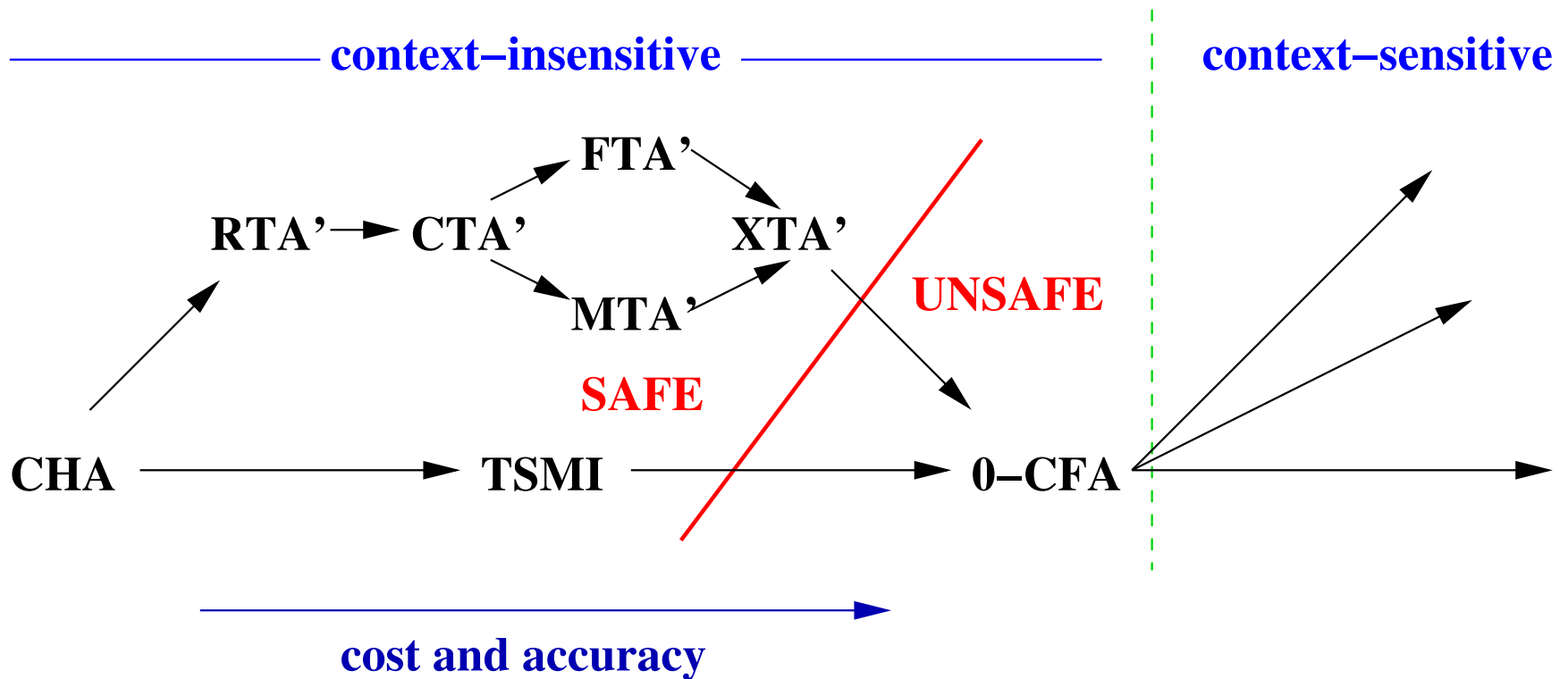
Experiments

Conclusion: our approach is correct, efficient, and scalable, and inlines many method calls.



# Optimization versus Typability

A more optimistic view:



## Inlining does not preserve typability (1/2)

---

```
B x = new C();  
x.m();
```

```
class B {  
    void m() {...}  
}
```

```
class C extends B {  
    C f;  
    void m() {  
        this.f = this;  
    }  
}
```

## Inlining does not preserve typability (2/2)

---

```
B x = new C();
```

```
x.f = x; // does not type check!
```

```
class B {  
    void m() {...}  
}
```

```
class C extends B {  
    C f;  
    void m() {  
        this.f = this;  
    }  
}
```

## Expensive approaches

---

Type casts: [Wright, Jagannathan et al. TIC'98]

Type casts may hurt performance.

```
B x = new C();
```

```
((C)x).f = (C)x; // two type casts!
```

```
class B {  
    void m() {...}  
}
```

```
class C extends B {  
    C f;  
    void m() {  
        this.f = this;  
    }  
}
```

Limited type inference, else type casts: [Hendren et al. SAS'00]

Type inference for objects:

[Palsberg et al. I&C 1995, NJC 1997, FOOL'02, LICS'02]

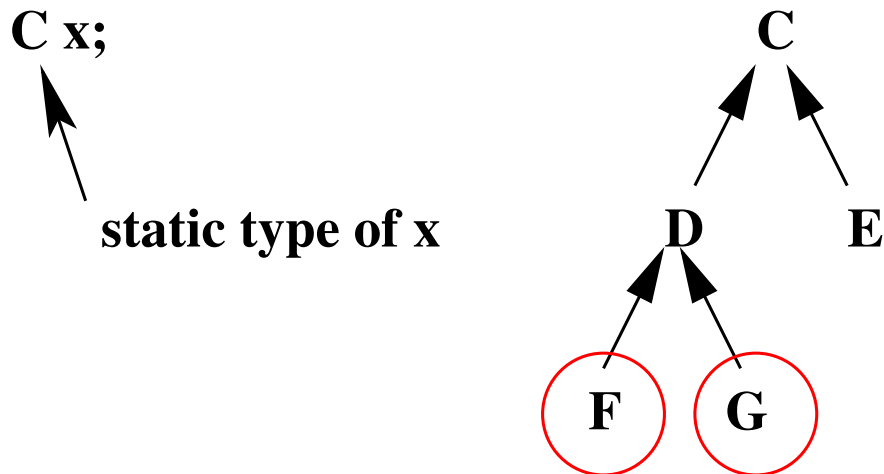
## Our approach: TSMI

---

Our approach = flow analysis + change types + inline.

Paradigm: type = least-upper-bound(flow-set)

class hierarchy (excerpt)



Flow set for **x** = { **F**, **G** }

**New type for x = l.u.b. { F, G } = D**

# TSMI

---

Key insight: need to tune the flow analysis to match the type system.

[Palsberg et al. IPL 1992, I&C 1995, TOPLAS 1995 + 1998, JFP 2001]

TSMI is more conservative than 0-CFA.

TSMI imposes more flow constraints than 0-CFA, for example:

- For each class  $C$ , the constraint:  $C \in [\text{this}]$
- For each method  $m$  that overrides another method  $m$ , the constraints:
  - the flow sets for corresponding arguments must be equal
  - the flow sets for the return values must be equal
- No flow set can be empty.

## Example of TSMI (1/2)

---

```
A x = new A();
B y = new B();

x.m( new Q() );
y.m( new S() );

class A {
    void m(Q arg) {
        arg.p();
    }
}
class B extends A {
    void m(Q arg) {...}
}

class Q {
    void p() {...}
}
class S extends Q {
    void p() {...}
}
```

## Example of TSMI (2/2)

---

```
A x = new A();  
B y = new B();
```

```
x.m( new Q() );
```

```
y.m( new S() );
```

```
class A {  
    void m(Q arg) {  
        arg.p();  
    }  
}  
class B extends A {  
    void m(Q arg) {...}  
}
```

```
class Q {  
    void p() {...}  
}  
class S extends Q {  
    void p() {...}  
}
```

## TSMI handles the problematic case correctly (1/2)

---

```
B x = new C();  
x.m();
```

```
class B {  
    void m() {...}  
}
```

```
class C extends B {  
    C f;  
    void m() {  
        this.f = this;  
    }  
}
```

## TSMI handles the problematic case correctly (2/2)

---

`C` `x = new C();`

`x.f = x;`

```
class B {  
    void m() {...}  
}
```

```
class C extends B {  
    C f;  
    void m() {  
        this.f = this;  
    }  
}
```

## TSMI: Lessons learned

---

Student experience: **very** difficult to align the flow analysis with the Java type system.

Our experience: we debugged TSMI over and over while doing the correctness proof.

Moral: first theory, then implementation.

# Correctness

---

Based on Featherweight Java [Igarashi, Pierce, Wadler, OOPSLA 1999]

Subset of the Java grammar, small-step operational semantics, type system, TSMI flow analysis, flow-directed inlining.

$\phi$  = flow information for the program  $P$  computed by TSMI.

$\llbracket P \rrbracket_\phi$  = program with transformed types, before inlining

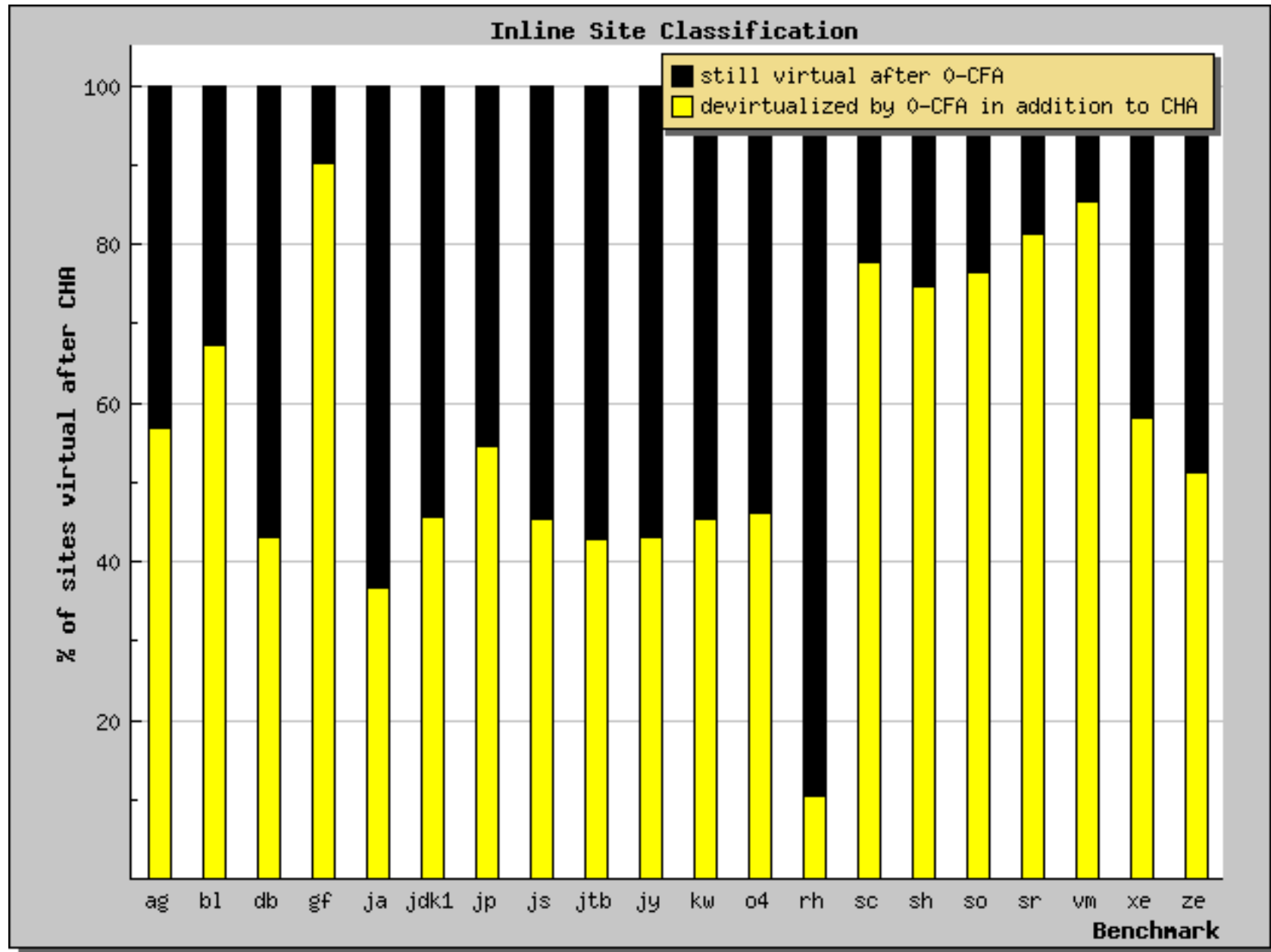
**Theorem 1 (Typability Preservation)** *If  $P$  type checks, then  $\llbracket P \rrbracket_\phi$  type checks.*

**Theorem 2 (Operational Correctness)**  $P \mapsto P'$  *if and only if*  $\llbracket P \rrbracket_\phi \mapsto \llbracket P' \rrbracket_\phi$ .

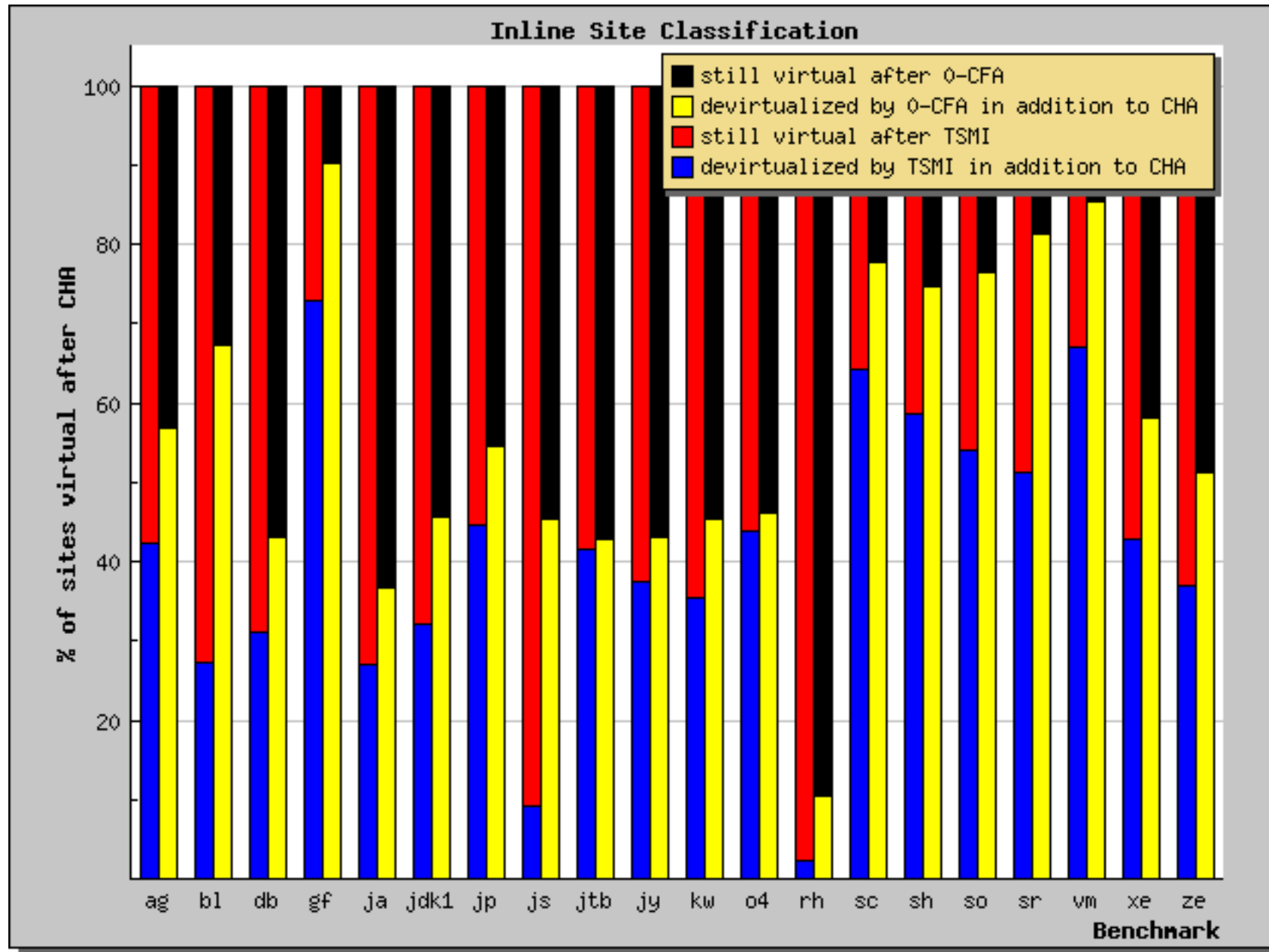
**Theorem 3 (Analysis Preservation)**  $\phi$  *is a TSMI flow analysis of*  $\llbracket P \rrbracket_\phi$ .

**Theorem 4 (Idempotence)**  $\llbracket \llbracket P \rrbracket_\phi \rrbracket_\phi = \llbracket P \rrbracket_\phi$ .

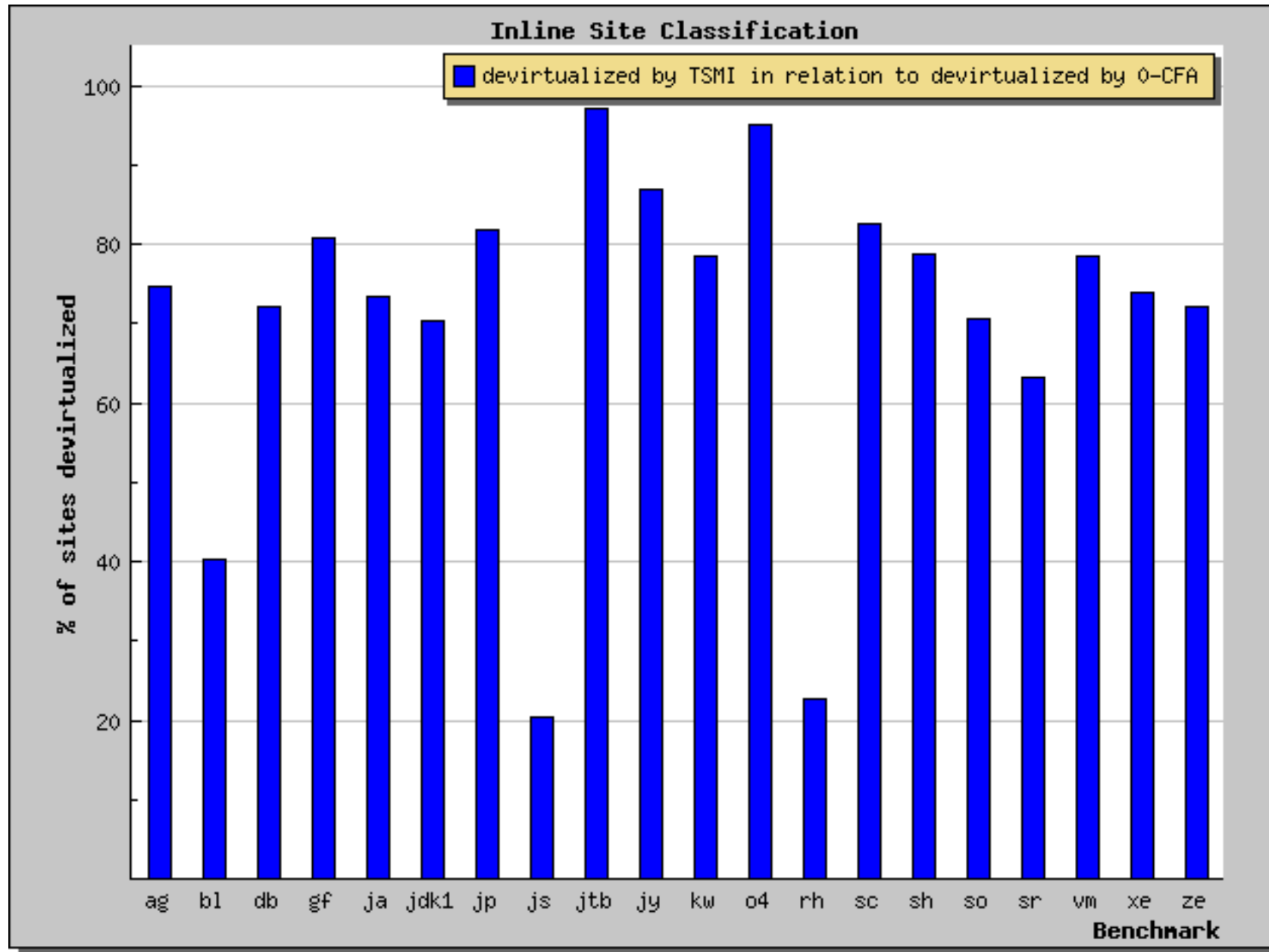
# How good is 0-CFA? (revisited)



# How good is TSMI?



# How good is TSMI relative to 0-CFA?



## Conclusion

---

Type-safe method inlining, based on a new flow analysis.

Almost as good as O-CFA and potentially much faster and more lightweight.