Compilation Methods

Prof. Dr. Uwe Kastens

Summer 2013

1 Introduction

Objectives

The students are going to learn

- what the main tasks of the **synthesis part of optimizing compilers** are,
- how **data structures and algorithms** solve these tasks systematically,
- what can be achieved by **program analysis and optimizing transformations**,

Prerequisites

- Constructs and properties of programming languages
- What does a compiler know about a program?
- How is that information represented?
- Algorithms and data structures of the analysis parts of compilers (frontends)

Main aspects of the lecture ***Programming Languages and Compilers*** (PLaC, BSc program)
http://ag-kastens.upb.de/lehre/material/plac

Syllabus

Week	**Chapter**	**Topic**
1	1 Introduction	Compiler structure
	2 Optimization	Overview: Data structures, program transformations
2		Control-flow analysis
3		Loop optimization
4, 5		Data-flow analysis
6		Object oriented program analysis
7	3 Code generation	Storage mapping
		Run-time stack, calling sequence
8		Translation of control structures
9		Code selection by tree pattern matching
10, 11	4 Register allocation	Expression trees (Sethi/Ullman)
		Basic blocks (Belady)
		Control flow graphs (graph coloring)
12	5 Code Parallelization	Data dependence graph
13		Instruction Scheduling
14		Loop parallelization
15	Summary	

References

Course material:

Compilation Methods: http://ag-kastens.upb.de/lehre/material/compii
Programming Languages and Compilers: http://ag-kastens.upb.de/lehre/material/plac

Books:

U. Kastens: **Übersetzerbau**, Handbuch der Informatik 3.3, Oldenbourg, 1990; (sold out)

K. Cooper, L. Torczon: Engineering A Compiler, Morgan Kaufmann, 2003

S. S. Muchnick: **Advanced Compiler Design & Implementation**,
Morgan Kaufmann Publishers, 1997

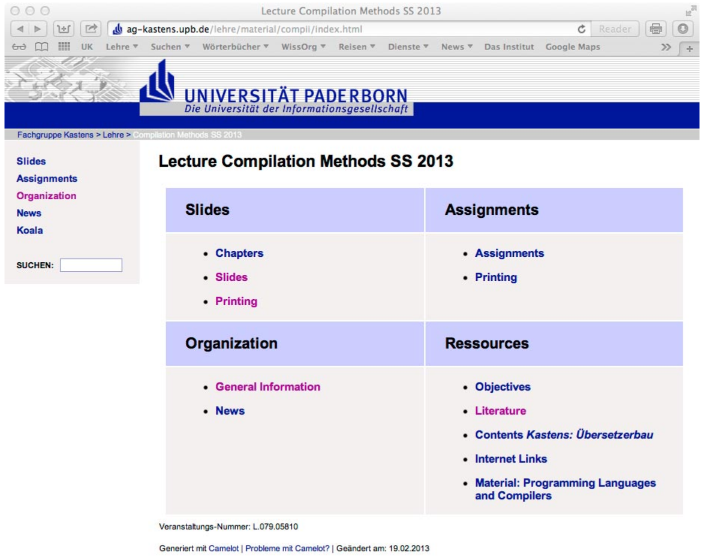
A. W. Appel: **Modern Compiler Implementation in C**, 2nd Edition
Cambridge University Press, 1997, (in Java and in ML, too)

W. M. Waite, L. R. Carter: **An Introduction to Compiler Construction,**
Harper Collins, New York, 1993

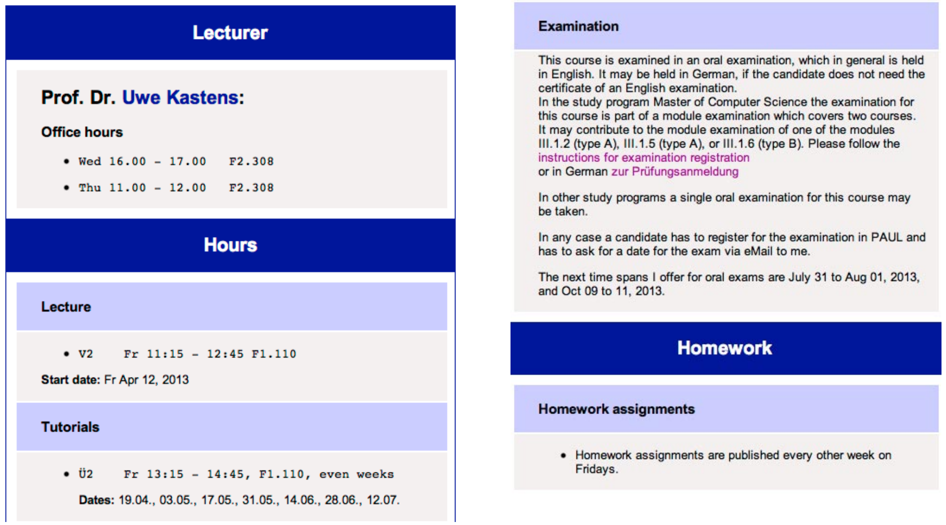
M. Wolfe: **High Performance Compilers for Parallel Computing**, Addison-Wesley, 1996

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: **Compilers - Principles, Techniques, & Tools**,
2nd Ed, Pearson International Edition (Paperback), and Addison-Wesley, 2007

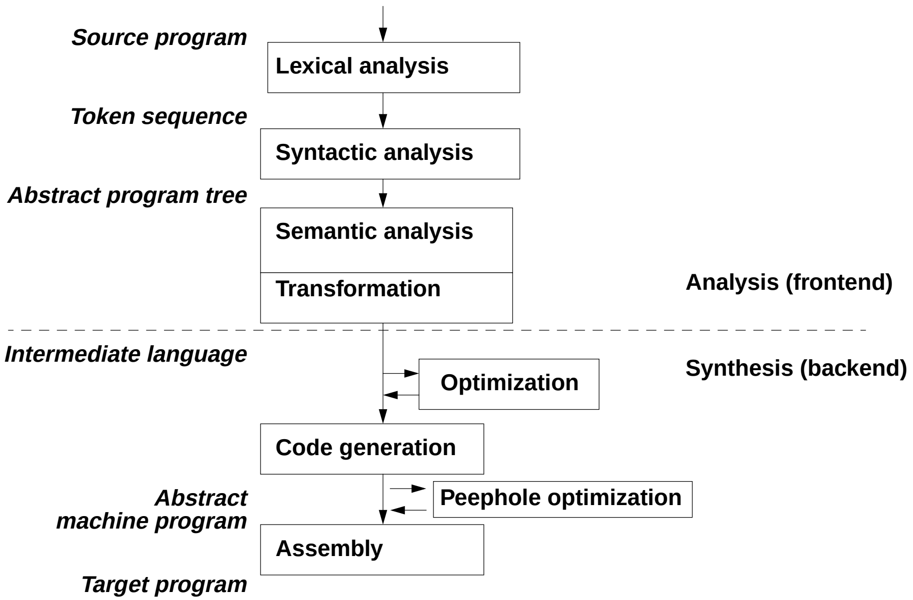
Course Material in the Web: HomePage



Course Material in the Web: Organization



Compiler Structure and Interfaces



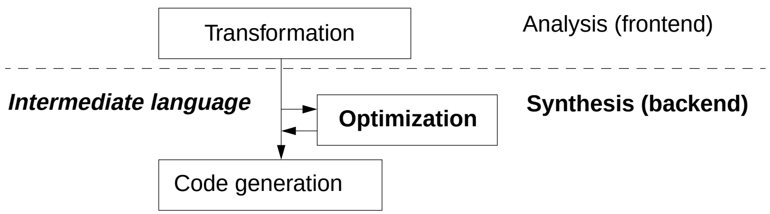
2 Optimization

Objective:
Reduce run-time and / or code size of the program,
without changing its observable effects.
Eliminate redundant computations, simplify computations.

Input: Program in intermediate language

Task: find redundancies (**analysis**)
improve the code (**optimizing transformations**)

Output: Improved program in intermediate language



Overview on Optimizing Transformations

Name of transformation: **Example for its application:**

1. **Algebraic simplification** of expressions
   `2*3.14 => 6.28   x+0 => x   x*2 =>` shift left `x**2 => x*x`
2. **Constant propagation** (dt. Konstantenweitergabe)
   constant values of variables propagated to uses: `x = 2; ... y = x * 5;`
3. **Common subexpressions** (gemeinsame Teilausdrücke)
   avoid re-evaluation, if values are unchanged `x = a*(b+c);...y = (b+c)/2;`
4. **Dead variables** (überflüssige Zuweisungen)
   eliminate redundant assignments `x = a + b; ... x = 5;`
5. **Copy propagation** (überflüssige Kopieranweisungen)
   substitute use of x by y `x = y; ... ; z = x;`
6. **Dead code** (nicht erreichbarer Code)
   eliminate code, that is never executed `b = true;...if (b) x = 5; else y = 7;`

Overview on Optimizing Transformations (continued)

Name of transformation: **Example for its application:**

7. **Code motion** (Code-Verschiebung)
   move computations to cheaper places `if (c) x = (a+b)*2; else x = (a+b)/2;`
8. **Function inlining** (Einsetzen von Aufrufen)
   substitute call of small function by a computation over the arguments `int Sqr (int i) { return i * i; }` `x = Sqr (b*3)`
9. **Loop invariant code**
   move invariant code before the loop `while (b) {... x = 5; ...}`
10. **Induction variables in loops**
   transform multiplication into incrementation `i = 1; while (b) { k = i*3; f(k); i = i+1;}`

Program Analysis for Optimization

Static analysis:
static properties of program structure and of **every execution**;
safe, pessimistic assumptions
where input and dynamic execution paths are not known

Context of analysis - the larger the more information:

Expression	local optimization
Basic block	local optimization
procedure (control flow graph)	global intra-procedural optimization
program module (call graph) separate compilation	global inter-procedural optimization
complete program	optimization at link-time or at run-time

Analysis and Transformation:
Analysis provides preconditions for **applicability of transformations**

Transformation may change analysed properties,
may **inhibit or enable** other transformations

Order of analyses and transformations **is relevant**

Program Analysis in General

Program text is systematically analyzed to exhibit
structures of the program,
properties of program entities,
relations between program entities.

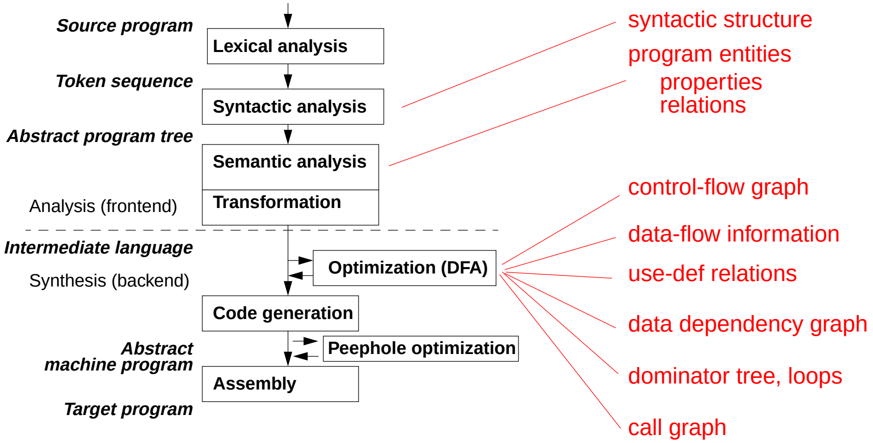
Objectives:

Compiler:
- Code improvement
- automatic parallelization
- automatic allocation of threads

Software engineering tools:
- program understanding
- software maintenance
- evaluation of software qualities
- reengineering, refactoring

Methods for program analysis stem from **compiler construction**

Overview on Program Analysis in Compilers

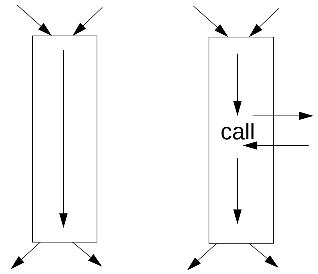


Basic Blocks

Basic Block (dt. Grundblock):
Maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.

Begin of a basic block:

- procedure entry
- target of a branch
- instruction after a branch or return (must have a label)



Function calls
are usually not considered as a branch, but as operations that have effects

Local optimization
considers the context of one single basic block (or part of it) at a time.

Global optimization:
Basic blocks are the nodes of control-flow graphs.

Example for Basic Blocks

A C function that computes Fibonacci numbers:

```
int fib (int m)
{  int f0 = 0, f1 = 1, f2, i;
   if (m <= 1)
      return m;
   else
   {  for(i=2; i<=m; i++)
      {  f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
}  }
```

Intermediate code with basic blocks:
[Muchnick, p. 170]

```
    receive m             B1
    f0 <- 0
    f1 <- 1
    if m <= 1 goto L3

    i <- 2                B3

L1: if i <= m goto L2     B4

    return f2             B5

L2: f2 <- f0 + f1         B6
    f0 <- f1
    f1 <- f2
    i <- i + 1
    goto L1

L3: return m              B2
```

if-condition belongs to the preceding basic block

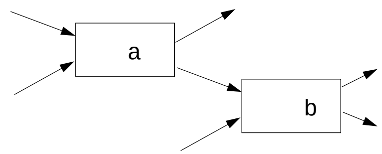
while-condition does not belong to the preceding basic block

Control-Flow Graph (CFG)

A **control-flow graph, CFG** (dt. Ablaufgraph)
represents the control structure of a function

Nodes: **basic blocks** and 2 unique nodes **`entry`** and **`exit`**.

Edge a -> b: **control may flow** from the end of **a** to the begin of **b**



Fundamental data structure for

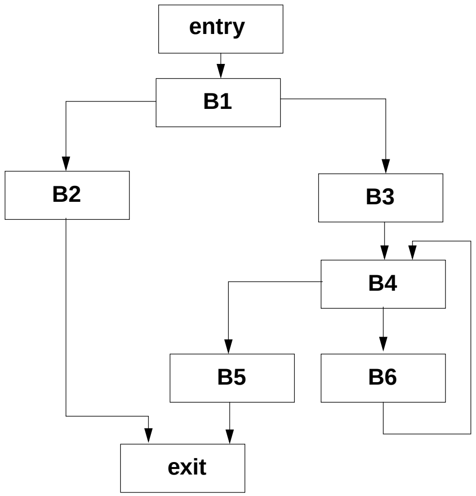
- control flow analysis
- structural transformations
- code motion
- data-flow analysis (DFA)

Example for a Control-flow Graph

Intermediate code with basic blocks:

```
    receive m                   B1
    f0 <- 0
    f1 <- 1
    if m <= 1 goto L3
    i <- 2                      B3
L1: if i <= m goto L2           B4
    return f2                   B5
L2: f2 <- f0 + f1               B6
    f0 <- f1
    f1 <- f2
    i <- i + 1
    goto L1
L3: return m                    B2
```

Control-flow graph:
[Muchnick, p. 172]



Control-Flow Analysis

Compute **properties on the control-flow** based on the CFG:

- **dominator relations**:
  properties of paths through the CFG
- **loop recognition**:
  recognize loops - independent of the source language construct
- **hierarchical reduction of the CFG**:
  a region with a unique entry node on the one level is a node of the next level graph

Apply **transformations** based on control-flow information:

- **dead code elimination**:
  eliminate unreachable subgraphs of the CFG
- **code motion**:
  move instructions to better suitable places
- **loop optimization**:
  loop invariant code, strength reduction, induction variables

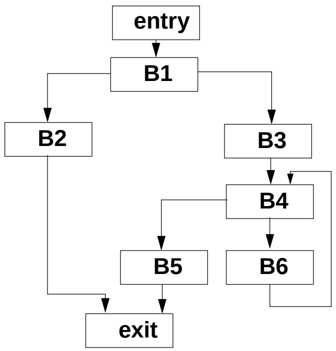
Dominator Relation on CFG

Relation over nodes of a CFG, characterizes paths through CFG,
used for loop recognition, code motion

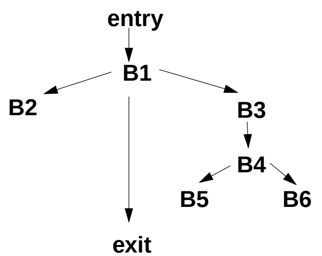
a dominates b (**a dom b**):
a is on every path from the entry node to b (reflexive, transitive, antisymmetric)

a is immediate dominator of b (a idom b):
a dom b and a ≠ b, and there is no c such that c ≠ a, c ≠ b, a dom c, c dom b.

control-flow graph



Tree of (immediate) dominators
(dom is transitive closure of the tree)



Immediate Dominator Relation is a Tree

Every node has a unique immediate dominator.

The dominators of a node are linearly ordered by the idom relation.

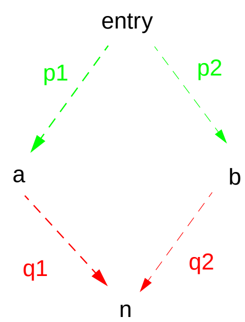
Proof by contradiction:
Assume:
a ≠ b, a dom n, b dom n and
not (a dom b) and not (b dom a)

Then there are pathes in the CFG

- p1: from entry to a not touching b, since not (b dom a)
- p2: from entry to b not touching a, since not (a dom b)
- q1: from a to n not touching b, since a dom n and not (a dom b)
- q2: from b to n not touching a, since b dom n and not (b dom a)

Hence, there is a path p1-q1 from
entry via a to n not touching b.
That is a contradiction to the assumption b dom n.
Hence, n has a unique immediate dominator, either a or b.

CFG



Dominator Computation

Algorithm computes the sets of dominators
Domin(n) for all nodes n∈N of a CFG:

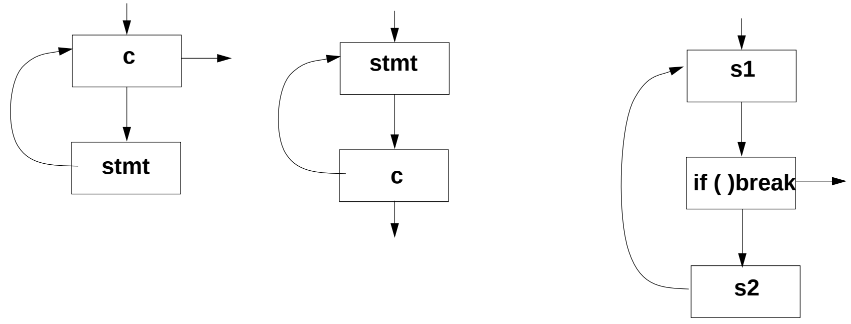
```
for each n∈N do Domin(n) = N;
Domin(entry) = {entry};

repeat
   for each n∈N-{entry} do
      T = N;
      for each p∈pred(n) do
         T = T ∩ Domin(p);
      Domin(n) = {n} ∪ T;
until Domin is unchanged
```

Symmetric relation for backward analysis:

a postdominates b (a pdom b):
a is on every path from b to the exit node (reflexive, transitive, antisymmetric)

Loop Recognition: Structured Loops

while (c) stmt; **do stmt; while (c);** **do s1; if ( )break; s2; while (true);**



Loop Recognition: Natural Loops

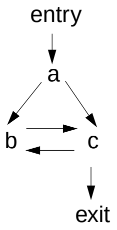
Back edge t->h in a CFG: head h dominates tail t (h dom t).

Natural loop of a back edge t->h:
set S of nodes such that S contains h, t and
all nodes from which t can be reached without passing through h.
h is the **loop header**.

Iterative computation of the natural loop for t->h:
add predecessors of nodes in S according to the formula:

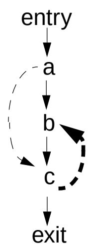
$$S = \{h, t\} \cup \{ p \mid \exists a \, (a \in S \setminus \{h\} \wedge p \in pred(a)) \}$$

This definition of **back edges** is stronger than that of **DFS back edges**:



no back edge

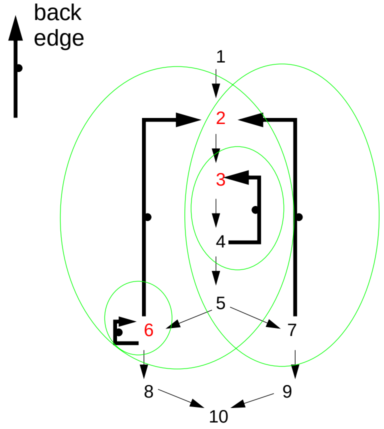
a DFS classification of edges:
tree, forward, back



DFS back edge

Example for Loop Recognition

back edge:	natural loop:
4 -> 3	$S_1 = \{3,4\}$
6 -> 2	$S_2 = \{2, 3, 4, 5, 6\}$
7 -> 2	$S_3 = \{2, 3, 4, 5, 7\}$
6 -> 6	$S_4 = \{6\}$

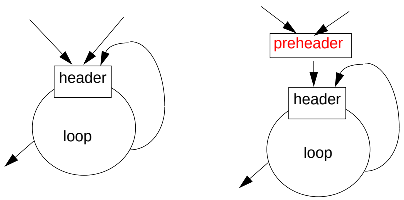


loops are

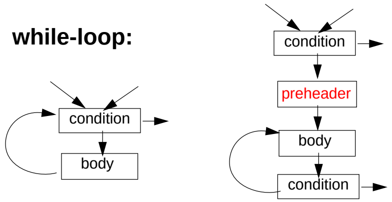
- **disjoint** $S_1 \cap S_4 = \emptyset$
- **nested** $S_1 \subset S_2$
- **non-nested,** S_2, S_3
  but have the same loop header,
  are comprised into one loop

Loop Optimization

- Introduce a **preheader** for a loop, as a place for loop invariant computations:
  a new, empty basic block that lies on every path to the loop header, but is not iterated:



while-loop:



- move **loop invariant computations** to the preheader:
  check use-def-chains: if an expression E contains no variables that are defined in the loop, then replace E by a temporary variable t, and compute t = E; in the preheader.
- eliminate **redundant bounds-checks**:
  propagate value intervals using the same technique as for constant propagation (see DFA)
  Example in Pascal:

```
var   a: array [1..10] of integer;
      i: integer;

for i := 1 to 10 do a[i] := i;
```

- **induction variables**, **strength reduction**: see next slide

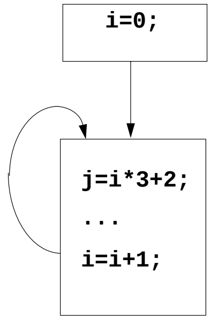
Loop Induction Variables

Induction variables may occur in any loop - not only in **`for`** loops.

Induction variable i:
i is incremented (decremented) by a constant value c on every iteration.

Basic induction variable i:
There is exactly one definition **`i = i + c;`** or **`i = i - c;`**
that is executed on every path through the loop.

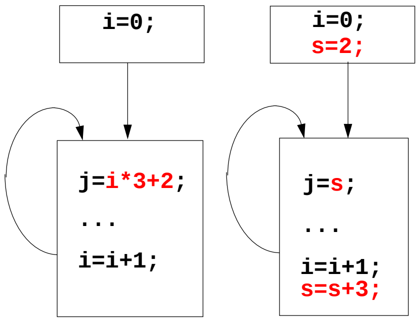
Dependent induction variable j:
j depends on induction variable i by a linear function **`i * a + b`**
represented by (i, a, b).



Transformation of Induction Variables

Transformation of dependent induction variables:

1. For each (i, a, b) create a temporary variable s.   **`j: (i, 3, 2)`**
2. Initialize **`s = i * a + b;`** in the preheader.
3. Replace **`i * a + b`** in the loop by s.
4. Add **`s = s + c*a;`** behind the increment of i



Strength reduction:
Replace a costly operation (multiplication) by a cheaper one (addition).

Linear increment of array address computation (next slide)

Examples for Transformations of Induction Variable

```
do
   k = i*3+1;
   f (5*k);
   /* x = a[i]; compiled: */
   x = cont(start+i*elsize);
   i = i + 2;
while (Ek)
```

basic induction variable:

```
   i:   c = 2
```

dependent induction variables:

```
   k:   (i, 3, 1)
   arg: (k, 5, 0)
   ind: (i, elsize, start)
```

```
sk = i*3+1;
sarg = sk*5;
sind = start + i*elsize;
do
   k = sk;
   f (sarg);
   x = cont (sind);
   i = i + 2;
   sk = sk + 6;
   sarg = sarg + 30;
   sind = sind + 2*elsize;
while (Ek)
```

Data-Flow Analysis

Data-flow analysis (DFA) provides information about
how the **execution of a program may manipulate its data**.

Many different problems can be formulated as **data-flow problems**, for example:

- Which assignments to variable **v** may influence a use of **v** at a certain program position?
- Is a variable **v** used on any path from a program position **p** to the exit node?
- The values of which expressions are available at program position **p**?

Data-flow problems are stated in terms of

- **paths through the control-flow graph** and
- **properties of basic blocks**.

Data-flow analysis provides information for **global optimization.**

Data-flow analysis does not know

- which input values are provided at run-time,
- which branches are taken at run-time.

Its results are to be interpreted **pessimistic**

Data-Flow Equations

A data-flow problem is stated as a **system of equations** for a control-flow graph.

System of Equations for **forward problems** (propagate information along control-flow edges):

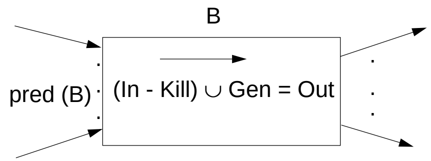
Example **Reaching definitions:**
A definiton **d** of a variable **v** reaches the begin of a block **B** if **there is a path** from **d** to **B** on which **v** is not assigned again.

In, Out, Gen, Kill represent analysis information:
sets of statements,
sets of variables,
sets of expressions
depending on the analysis problem

2 equations for each basic block:

$$Out(B) = f_B(In(B)) = Gen(B) \cup (In(B) - Kill(B))$$

$$In(B) = \underset{h \in pred(B)}{\Theta} Out(h)$$



In, Out **variables** of the system of equations for each block

Gen, Kill a pair of **constant sets** that characterize a block w.r.t. the DFA problem

Θ meet operator; e. g. $\Theta = \cup$ for „reaching definitions“, $\Theta = \cap$ for „available expressions“

Specification of a DFA Problem

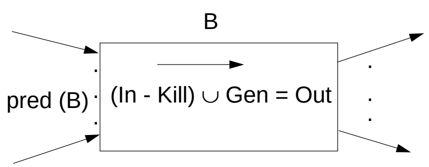
Specification of reaching definitions:

1. **Description**:
   A definiton **d** of a variable **v** reaches the begin of a block **B**
   if **there is a path** from **d** to **B** on which **v** is not assigned again.
2. It is a **forward problem**.
3. The **meet operator** is union.
4. The **analysis information** in the sets are assignments at certain program positions.
5. **Gen (B)**:
   contains all definitions `d: v = e;` in **B**,
   such that **v** is not defined after **d** in **B**.
6. **Kill (B)**:
   if **v** is assigned in **B**, then **Kill(B)**
   contains all definitions `d: v = e;`
   of blocks different from **B**.

2 equations for each basic block:

$$Out(B) = f_B(In(B)) = Gen(B) \cup (In(B) - Kill(B))$$

$$In(B) = \underset{h \in pred(B)}{\Theta} Out(h)$$



Variants of DFA Problems

- **forward** problem:
  DFA information flows **along the control flow**
  In(B) is determined by Out(h) of the predecessor blocks

  backward problem (see C-2.23):
  DFA information flows **against the control flow**
  Out(B) is determined by In(h) of the successor blocks
- **union** problem:
  problem description: „there is a path“;
  meet operator is $\Theta = \cup$
  solution: minimal sets that solve the equations

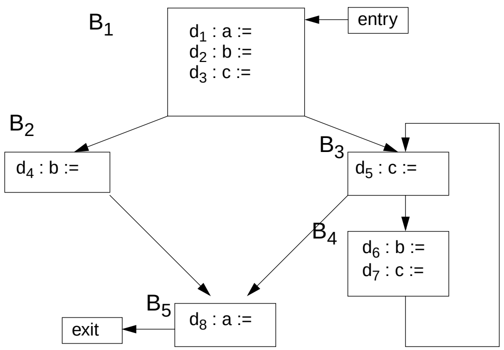
  intersect problem:
  problem description: „for all paths“
  meet operator is $\Theta = \cap$
  solution: maximal sets that solve the equations
- **optimization information**: **sets of** certain statements, of variables, of expressions.

Further classes of DFA problems over general lattices instead of sets are not considered here.

Example Reaching Definitions

Gen (B):
contains all definitions **d: v = e;** in **B**, such that **v** is not defined after **d** in **B**.

Kill (B):
contains all definitions **d: v = e;** in blocks different from **B**, such that B has a definition of **v**.



	Description of DFA-Problem: Gen	Kill	DFA-Solution: In	Out
B_1	d_1, d_2, d_3	d_4, d_5, d_6, d_7, d_8	$\varnothing$	d_1, d_2, d_3
B_2	d_4	d_2, d_6	d_1, d_2, d_3	d_1, d_3, d_4
B_3	d_5	d_3, d_7	d_1, d_2, d_3, d_6, d_7	d_1, d_2, d_5, d_6
B_4	d_6, d_7	d_2, d_3, d_4, d_5	d_1, d_2, d_5, d_6	d_1, d_6, d_7
B_5	d_8	d_1	$d_1, d_2, d_3, d_4, d_5, d_6$	$d_2, d_3, d_4, d_5, d_6, d_8$

Iterative Solution of Data-Flow Equations

Input: the CFG; the sets Gen(B) and Kill(B) for each basic block B
Output: the sets In(B) and Out(B)

Algorithm:

```
repeat
   stable := true;
   for all B ≠ entry   {*}
   do begin
      for all V ∈ pred(B) do
         In(B):= In(B) Θ Out(V);
      oldout:= Out(B);
      Out(B):= Gen(B) ∪ (In(B)-Kill(B));
      stable:= stable and Out(B)=oldout
   end
until stable
```

Initialization

Union: empty sets

```
for all B do
begin
   In(B):=∅;
   Out(B):=Gen(B)
end;
```

Intersect: full sets

```
for all B do
begin
   In(B) := U;
   Out(B):=
      Gen(B)∪
         (U - Kill(B))
end;
```

Complexity: $O(n^3)$ with n number of basic blocks
$O(n^2)$ if $|$**pred (B)**$|$ ≤ **k << n** for all B

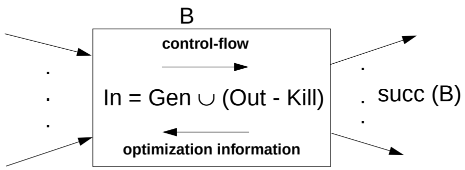
Backward Problems

System of Equations for **backward problems** propagate information against control-flow edges:

2 equations for each basic block:

$$In(B) = f_B(Out(B)) = Gen(B) \cup (Out(B) - Kill(B))$$

$$Out(B) = \underset{h \in succ(B)}{\Theta} In(h)$$



Example **Live variables:**

1. Description: Is variable **v** alive at a given point **p** in the program, i. e. **is there a path** from **p** to the exit where **v** is used but not defined before the use?
2. backward problem
3. optimization information: sets of variables
4. meet operator: Θ **=** ∪ union
5. Gen (B): variables that are used in B, but not defined before they are used there.
6. Kill (B): variables that are defined in B, but not used before they are defined there.

Important Data-Flow Problems

1. **Reaching definitions:** A definiton **d** of a variable **v** reaches the beginning of a block **B** if there is a path from **d** to **B** on which **v** is not assigned again.
   DFA variant: forward; union; set of assignments
   Transformations: use-def-chains, constant propagation, loop invariant computations
2. **Live variables:** Is variable **v** alive at a given point **p** in the program, i. e. there is a path from **p** to the exit where **v** is used but not defined before the use.
   DFA variant: backward; union; set of variables
   Transformations: eliminate redundant assignments
3. **Available expressions:** Is expression **e** computed on every path from the entry to a program position **p** and none of its variables is defined after the last computation before **p**.
   DFA variant: forward; intersect; set of expressions
   Transformations: eliminate redundant computations
4. **Copy propagation:** Is a copy assignment **c: x = y** redundant, i.e. on every path from **c** to a use of **x** there is no assignment to **y**?
   DFA variant: forward; intersect; set of copy assignments
   Transformations: remove copy assignments and rename use
5. **Constant propagation:** Has variable **x** at position **p** a known value, i.e. on every path from the entry to **p** the last definition of **x** is an assignment of the same known value.
   DFA variant: forward; combine function; vector of values
   Transformations: substitution of variable uses by constants

Algebraic Foundation of DFA

DFA performs computations on a **lattice (dt. Verband)** of values, e. g. bit-vectors representing finite sets. It guarantees termination of computation and well-defined solutions. see [Muchnick, pp 223-228]

A **lattice L** is a set of values with two operations: $\cap$ **meet** and $\cup$ **join**

Required properties:

1. **closure**: $x, y \in L$ implies $x \cap y \in L$, $x \cup y \in L$
2. **commutativity**: $x \cap y = y \cap x$ and $x \cup y = y \cup x$
3. **associativity**: $(x \cap y) \cap z = x \cap (y \cap z)$ and $(x \cup y) \cup z = x \cup (y \cup z)$
4. **absorption**: $x \cap (x \cup y) = x = x \cup (x \cap y)$
5. unique elements **bottom** $\perp$, **top** T:
   $x \cap \perp = \perp$ and $x \cup T = T$

In most DFA problems only a **semilattice** is used with L, $\cap$, $\perp$ or L, $\cup$, T

Partial order defined by meet, $x \subseteq y$: $x \cap y = x$; defined by join: $x \supseteq y$: $x \cup y = x$
(transitive, antisymmetric, reflexive)

Some DFA Lattices

2 Bool
$\cap$ = and
$\cup$ = or
T = true
|
$\perp$ = false

1 Bit-Vector $BV^{n=3}$
$\cap$ = bitwise and
$\cup$ = bitwise or
111
110 101 011
100 010 001
000

3 Variable usage
{defined, used}
{defined} {used}
{}

4 ICP Integer Constant Propagation Lattice
T
false ··· -1 0 1 ··· true
$\perp$

$n \cap \perp = \perp$ $n \cap n = n$ $n \cap m = \perp$ if $n \neq m$
$n \cup T = T$ $n \cup n = n$ $n \cup m = T$ if $n \neq m$

5 Range Lattice: $[lo, hi] \in (Z \cup \{-\infty, \infty\})^2$

$\perp$ = [ ] empty range, T = $[-\infty, \infty]$,
$x \subseteq y$: x is contained in y

$\cap$: $[l1, h1] \cap [l2, h2] = x$
let l = max (l1, l2),
h = min (h1, h2),
x = if h < l then $\perp$ else [l, h]

$\cup$: $[l1, h1] \cup [l2, h2] =$
[min(l1, l2), max(h1, h2)]

6 Semilattice of types
$\cup$: $x \cup y$ = smallest common supertype of x and y
Object
A B
C D E F

Monotone Functions Over Lattices

The **effects of program constructs on DFA information** are described by functions over a suitable lattice,

e. g. the function for basic block B_3 on C-2.22:

$f_3(<x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8>) = <x_1\ x_2\ 0\ x_4\ 1\ x_6\ 0\ x_8> \in BV^8$

Gen-Kill pair encoded as function

$f: L \rightarrow L$ is a **monotone function** over the lattice L if
$\forall\ x, y \in L: x \subseteq y \Rightarrow f(x) \subseteq f(y)$

Finite height of the lattice and **monotonicity** of the functions guarantee **termination** of the algorithms.

Fixed points z of the function f, with f(z) = z, is a solution of the set of DFA equations.

MOP: Meet over all paths solution is desired, i. e. the „best“ with respect to L

MFP: Maximum fixed point is computed by algorithms, if functions are monotone

If the functions f are additionally **distributive**, then **MFP = MOP**.
$f: L \rightarrow L$ is a **distributive function** over the lattice L if
$\forall\ x, y \in L: f(x \cap y) = f(x) \cap f(y)$

Variants of DFA Algorithms

Heuristic improvement:

Goal: propagate changes in the In and Out sets as fast as possible.
Technique: visit CFG nodes in topological order in the outer for-loop {*}.
Then the number of iterations of the outer repeat-loop is only determined by back edges in the CFG

Algorithm for backward problems:

Exchange In and Out sets symmetrically in the algorithm of C-2.22b.
The nodes should be visited in topological order as if the directions of edges were flipped.

Hierarchical algorithms, interval analysis:

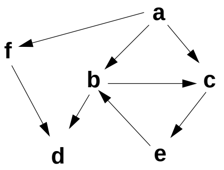
Regions of the CFG are considered nodes of a CFG on a higher level.
That abstraction is recursively applied until a single root node is reached.
The Gen, Kill sets are combined in upward direction;
the In, Out sets are refined downward.

Program Analysis: Call Graph (context-insensitive)

Nodes: defined functions

Arc g -> h: function g contains a call h(),
i. e. a call g() **may** cause the execution of a call h()

```
void a () {...b()...c()...f()...}
void b () {...d()...c()...}
void c()  {...e()...}
void d()  {...}
void e()  {...v++;...b()...}
void f()  {...d()...}
```



Analysis of structure:
b, c, e are recursive;
a, d, f are non-recursive

Propagation of properties:
assume a call e() may **modify a global variable** v
then calls a(), b(), c() may indirectly cause modification of v

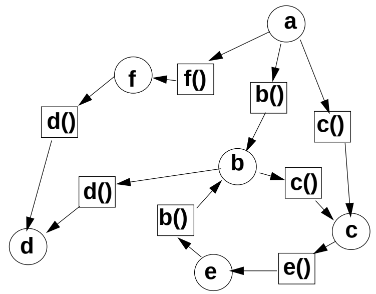
```
v = f(); cnt = 0; while(...){...b(); cnt += v;}
```


Program Analysis: Call Graph (context-sensitive)

Nodes: defined functions and calls (bipartite)

Arc g -> h: function g contains a call h(),i.e a call g() **may** cause the execution of a call h()
or call g() leads to function g

```
void a () {...b()...c()...f()...}
void b () {...d()...c()...}
void c()  {...e()...}
void d()  {...}
void e()  {...v++;...b()...}
void f()  {...d()...}
```



Calls of the same function in different contexts are distinguished by **different nodes**, e.g. the call of c in a and in b.

Analysis can be **more precise** in that aspect.

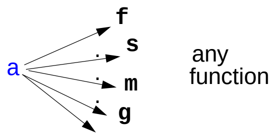
Calls Using Function Variables

Contents of **function variables is assigned at run-time**.

Static analysis does not know (precisely) which function is called.

Call graph has to assume that **any function may be called**.

```
void a()
  {...(*h)(0.3, 27)...}
```



Analysis for a better approximation of potential callees:

only those functions which

1. fit to the type of h
2. are assigned somewhere in the program
3. can be derived from the reaching definitions at the call

```
void m (int j) {...}

void g (float x, int i) {...}

...k = m;... f(g); ...

void a()
  { void (*h)(float,int) = g;
    ...
    if(...) h = s;

    ...(*h)(0.3, 27)...
  }
```


Analysis of Object-Oriented Programs

Aspects specific for object-oriented analysis:

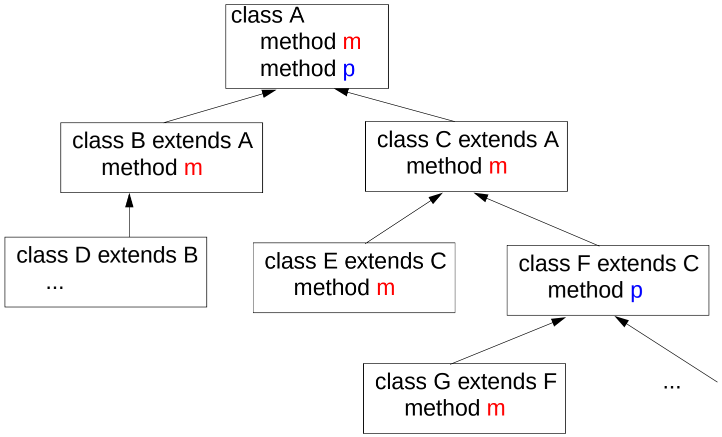
1. **hierarchy of classes and interfaces**
   specifies a complex **system of subtypes**
2. **hierarchy of classes and interfaces**
   specifies **inheritance and overriding** relation for methods
3. **dynamic method binding**
   for method calls **`v.m(...)`** the **callee is determined at run-time**
   good object-oriented style relies on that feature
4. **many small methods** are typical object-oriented style
5. **library use and reuse of modules**
   complete program contains many **unused classes and methods**

Static predictions for dynamically bound method calls
are essential for most analyses

Class Hierarchy Graph

Node: **class or interface**

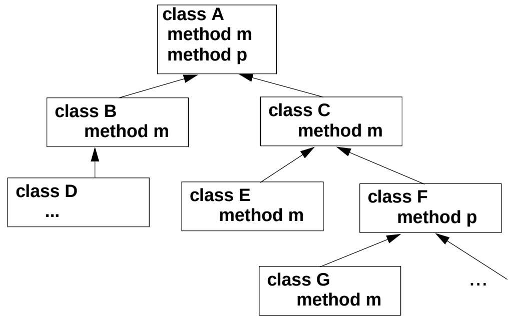
Arc a -> b: a is **subclass of** b or a **implements interface** b



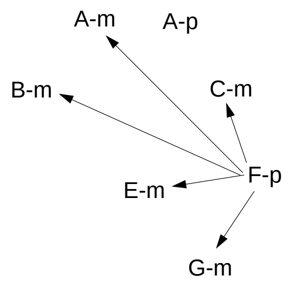
Object-Oriented Call Graph

Node: **implemented method**, identified by class name, method name: X-a

Arc X-a -> Y-b: method **X-a contains a call** v.b(...) that **may be bound to Y-b**



Call graph for F-p containing v.m(...)



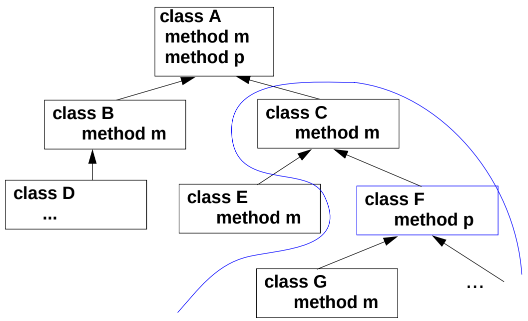
Call graph: **any method m** may be bound to that call in F-p
(compare to function variables)
analysis yields better approximations

Call Graphs Constructed by Class Hierarchy Analysis (CHA)

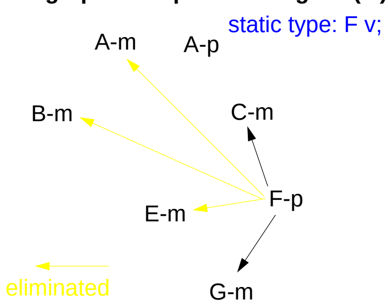
The call graph is reduced to a set of **reachable methods** using the **class hierarchy** and the **static type of the receiver** expression in the call:

If a method F-p **is reachable** and
if it contains a **dynamically bound call v.m(...)** and
T is the static type of v,

then every method **m that is inherited by T or by a subtype of T is also reachable**, and arcs go from F-p to them.



Call graph for F-p containing v.m(...)



Refined Approximations for Call Graph Construction

Class Hierarchy Analysis (CHA): (see C-2.32)

Rapid Type Analysis (RTA):

As CHA, but only methods of those classes C are considered which are instantiated (`new C()`) in a reachable method.

Reaching Type Analysis:

Approximations of run-time types is propagated through a graph: nodes represent variables, arcs represent copy assignments.

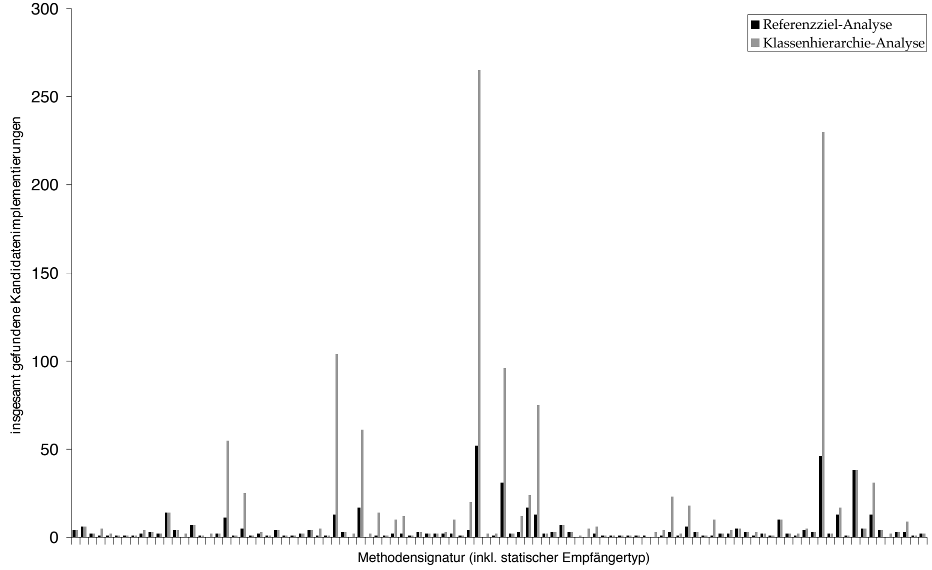
Declared Type Analysis: one node T represents all variables declared to have type T

Variable Type Analysis: one node V represents a single variable

Points-to Analysis:

Information on object identities is propagated through the control-flow graph

Results of Analysis of Dynamically Bound Calls



Modules of a Toolset for Program Analysis

analysis module	purpose	category
ClassMemberVisibility	examines visibility levels of declarations	visualization
MethodSizeStatistics	examines length of method implementations in bytecode operations and frequency of different bytecode operations	
ExternalEntities	histogram of references to program entities that reside outside a group of classes	
InheritanceBoundary	histogram of lowest superclass outside a group of classes	
SimpleSetterGetter	recognizes simple access methods with bytecode patterns	
MethodInspector	decomposes the raw bytecode array of a method implementation into a list of instruction objects	auxiliary analysis
ControlFlow	builds a control flow graph for method implementations	fundamental analyses
Dominator	constructs the dominator tree for a control flow graph	
Loop	uses the dominator tree to augment the control flow graph with loop and loop nesting information	
InstrDefUse	models operand accesses for each bytecode instruction	
LocalDefUse	builds intraprocedural def/use chains	
LifeSpan	analyzes lifeness of local variables and stack locations	
DefUseTypeInfo	infers type information for operand accesses	analysis of incomplete programs
Hierarchy	class hierarchy analysis based on a horizontal slice of the hierarchy	
PreciseCallGraph	builds call graph based on inferred type information, copes with incomplete class hierarchy	
ParamEscape	transitively traces propagation of actual parameters in a method call (escape = leaves analyzed library)	
ReadWriteFields	transitive liveness and access analysis for instance fields accessed by a method call	

Table 0-1. Analysis plug-ins in our framework

[Michael Thies: Combining Static Analysis of Java Libraries with Dynamic Optimization, Dissertation, Shaker Verlag, April 2001]

3. Code Generation

Input: Program in intermediate language

Tasks:

Storage mapping	properties of program objects (size, address) in the definition module
Code selection	generate instruction sequence, optimizing selection
Register allocation	use of registers for intermediate results and for variables

Output: abstract machine program, stored in a data structure

Design of code generation:

- analyze **properties of the target processor**
- plan **storage mapping**
- design at least one **instruction sequence** for each operation of the intermediate language

Implementation of code generation:

- Storage mapping:
  a traversal through the program and the definition module computes
  sizes and addresses of storage objects
- Code selection: use a generator for pattern matching in trees
- Register allocation:
  methods for expression trees, basic blocks, and for CFGs

3.1 Storage Mapping

Objective:
for each storable program object compute storage class, relative address, size

Implementation:
use properties in the definition module, traverse defined program objects

Design the use of storage areas:

code storage	progam code
global data	to be linked for all compilation units
run-time stack	activation records for function calls
heap	storage for dynamically allocated objects, garbage collection
registers for	addressing of storage areas (e. g. stack pointer) function results, arguments local variables, intermediate results (**register allocation**)

Design the mapping of data types (next slides)
Design activation records and translation of function calls (next section)

Storage Mapping for Data Types

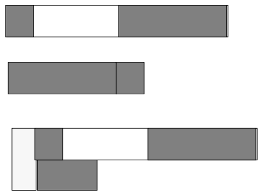
Basic types

arithmetic, boolean, character types

match language requirements and machine properties:
data format, available instructions,
size and alignment in memory

Structured types

for each type	representation in memory and code sequences for operations, e. g. assignment, selection, ...
record	relative address and alignment of components; reorder components for optimization
union	storage overlay, tag field for discriminated union
set	bit vectors, set operations



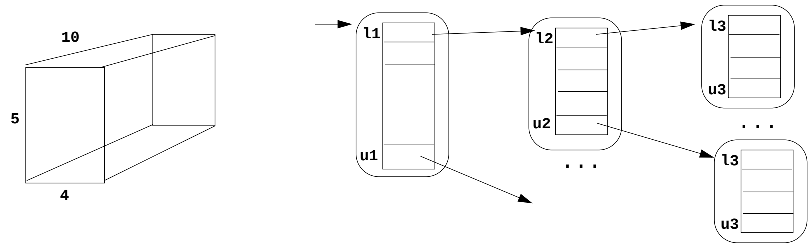
for **arrays** and **functions** see next slides

Array Implementation: Pointer Trees

An n-dimensional array

```
a: array[l1..u1, l2..u2, ..., ln..un] of real;
```

is implemented by a **tree of linear arrays**;
n-1 levels of pointer arrays and data arrays on the n-th level



Each single array can be allocated separately, dynamically; scattered in memory

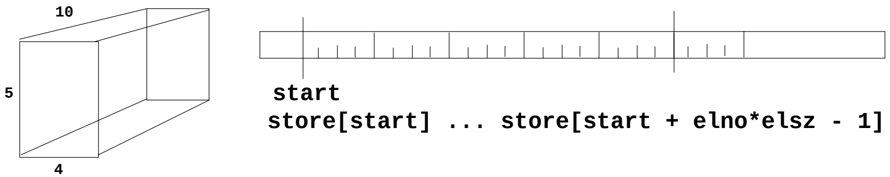
In **Java arrays** are implemented this way.

Array Implementation: Contiguous Storage

An n-dimensional array

```
a: array[l1..u1, l2..u2, ..., ln..un] of real;
```

is mapped to **one contiguous storage area**
linearized in row-major order:




```
start
store[start] ... store[start + elno*elsz - 1]
```

linear storage map of array a onto byte-array **`store`** from index **`start`**:

number of elements	**`elno = st1 * st2 * ... * stn`**
i-th index stride	**`sti = ui - li + 1`**
element size in bytes	**`elsz`**

Index map of **`a[i1, i2, ..., in]`**:

```
store[start+ (..((i1-l1)*st2 + (i2-l2))*st3 +..)*stn + (in-ln))*elsz]

store[const + (..(i1*st2     + i2)*st3     +..)*stn +  in)*elsz]
```


Functions as Data Objects

Functions may occur **as data objects**:

- variables
- parameters
- function results
- lambda expressions
(in functional languages)

Functions that are defined on the **outermost program level** (non-nested)

can be implemented by just the **address of the code**.

Functions that are **defined in nested structures** have to be implemented by a **pair: (closure, code)**

The **closure** contains all **bindings** of names to variables or values that are valid when the **function definition is executed**.

In **run-time stack** implementations the
closure is a sequence of activation records on the static predecessor chain.

3.2 Run-Time Stack Activation Records

Run-time stack contains one **activation record** for each active function call.

Activation record:
provides storage for the data of a function call.

dynamic link:
link from callee to caller,
to the preceding record on the stack

static link:
link from callee c to the record s where c is defined

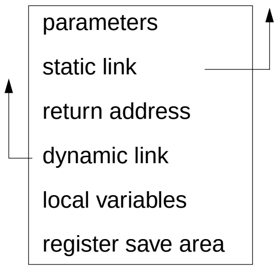
s is a call of a function which contains the definition
of the function, the call of which created c.

Variables of surrounding functions are
accessed via the static predecessor chain.

Only relevant for languages which allow
nested functions, classes, objects.

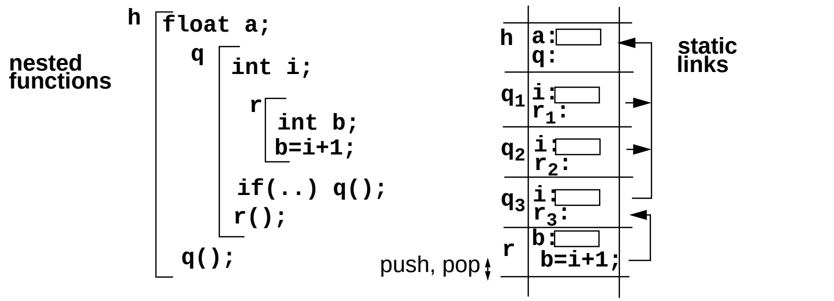
closure of a function call:
the **activation records on the static predecessor chain**

activation record:



Example for a Run-Time Stack

Run-time stack:
A call creates an activation record and pushes it onto the stack.
It is popped on termination of the call.



The **static link** points to the activation record where the called function is defined, e. g. r_3 in q_3

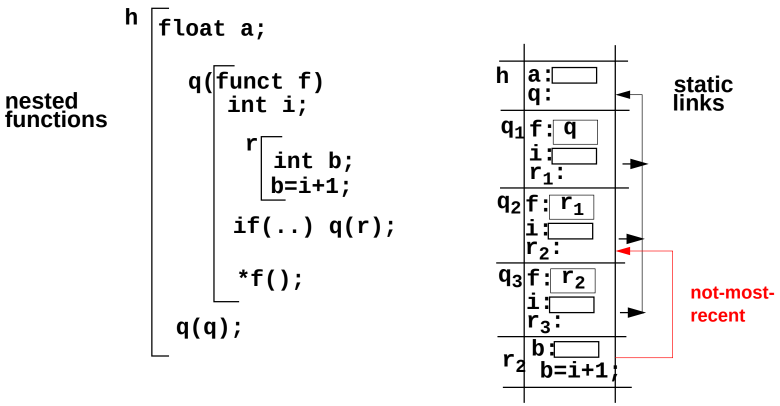
Optimization: activation records of **non-recursive functions** may be allocated statically.
Languages without recursive functions (FORTRAN) do not need a run-time stack.

Parallel processes, threads, and coroutines need a **separate run-time stack** each.

Not-Most-Recent Property

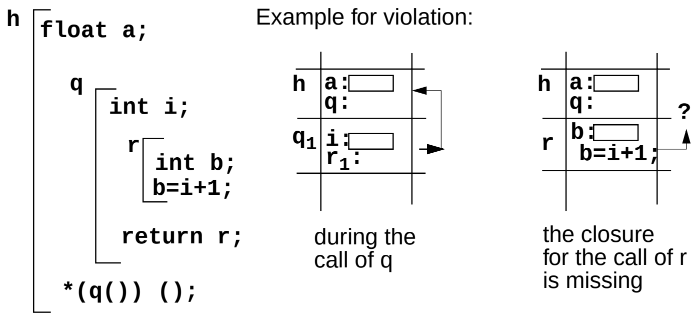
The **static link** of an activation record c for a function r
points to an activation record d for a function q where r is defined in.
If there are activation records for q on the stack, that are more recently created than d,
the **static link to d is not-most-recent**.

That effect can be achieved by using functional parameters or variables.
Example:



Closures on Run-Time Stacks

Function calls can be implemented by a run-time stack if the
closure of a function is still on the run-time stack when the function is called.



Language conditions to guarantee run-time stack discipline:

Pascal:	functions not allowed as function results, or variables
C:	no nested functions
Modula-2:	nested functions not allowed as values of variables

Functional languages maintain activation records on the heap instead of the run-time stack

Activation Records and Call Code

activation record:

- result
- parameters
- static link
- return address
- dynamic link
- local variables
- register save area

\- ↑ +

← 0 base address

\+ ↓ -

call code

push parameter values
push static link
subroutine jump

function code

push dynamic link
stack register := top of stack
increment top of stack
for local variables
save registers
...
function body
...
restore registers
deallocate local variables
pop stack register
return jump

pop static link
pop parameter area
use and pop result

3.3 Code Sequences for Control Statements

A **code sequence** defines how a **control statement** is transformed into jumps and labels.

Notation of the **Code** constructs:

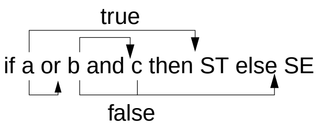
`Code (S)`	generate code for statements **S**
`Code (C, true, M)`	generate code for condition **C** such that it branches to M if **C** is true, otherwise control continues without branching
`Code (A, Ri)`	generate code for expression **A** such that the result is in register **Ri**

Code sequence for if-else statement:

```
if (cond) ST; else SE;:

        Code (cond, false, M1)
        Code (ST)
        goto M2
  M1:   Code (SE)
  M2:
```


Short Circuit Translation of Boolean Expressions

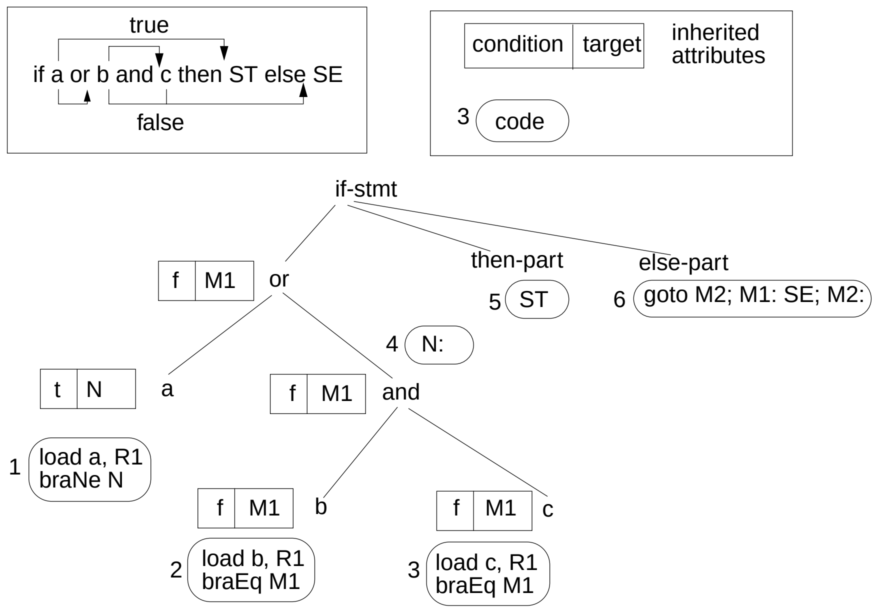
Boolean expressions are translated into **sequences of conditional branches**.
Operands are evaluated from left to right until the result is determined.



2 code sequences for each operator; applied to condition tree on a top-down traversal:

Code (A and B, true, M):	Code (A, false, N) Code (B, true, M) N:
Code (A and B, false, M):	Code (A, false, M) Code (B, false, M)
Code (A or B, true, M):	Code (A, true, M) Code (B, true M)
Code (A or B, false, M):	Code (A, true, N) Code (B, false, M) N:
Code (not A, X, M):	Code (A, not X, M)
Code (A < B, true, M):	Code (A, Ri); Code (B, Rj) cmp Ri, Rj braLt M
Code (A < B, false, M):	Code (A, Ri); Code (B, Rj) cmp Ri, Rj braGe M
Code for a leaf:	conditional jump

Example for Short Circuit Translation



Code Sequences for Loops

While-loop variant 1:

```
while (Condition) Body

   M1:   Code (Condition, false, M2)
         Code (Body)
         goto M1
   M2:
```

While-loop variant 2:

```
while (Condition) Body

         goto M2
   M1:   Code (Body)
   M2:   Code (Condition, true, M1)
```

Pascal for-loop unsafe variant:

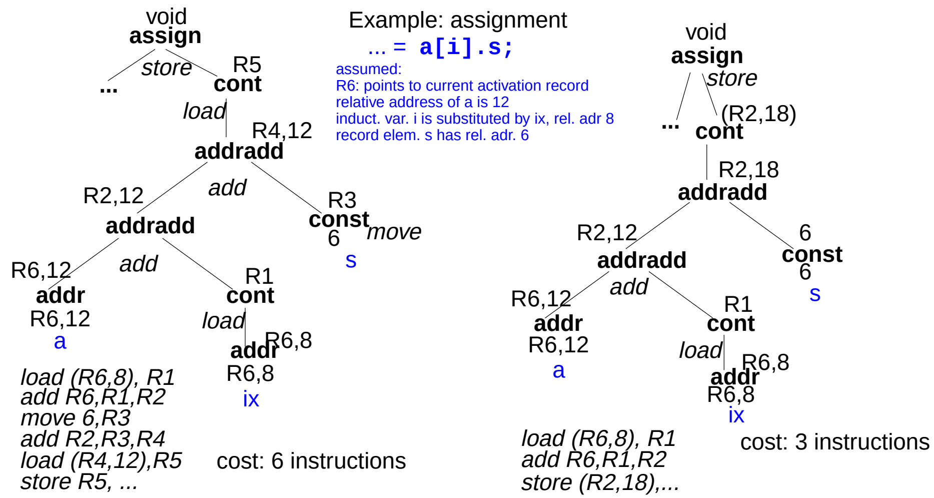
```
for i:= Init to Final do Body

      i = Init
   L: if (i>Final) goto M
      Code (Body)
      i++
      goto L
   M:
```

Pascal for-loop safe variant:

```
for i:= Init to Final do Body

      if (Init==minint) goto L
      i = Init - 1
      goto N
   L: Code (Body)
   N: if (i>= Final) goto M
      i++
      goto L
   M:
```


3.4 Code Selection

- Given: target tree in intermediate language.
- **Optimizing selection: Select patterns** that translate single nodes or small subtrees into machine instructions; cover the whole tree with as few instructions as possible.
- Method: **Tree pattern matching,** several techniques



Selection Technique: Value Descriptors

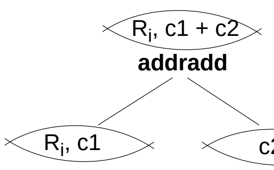
Intermediate language **tree node operators**; e.g.:

addr	address of variable
const	constant value
cont	load contents of address
addradd	address + value

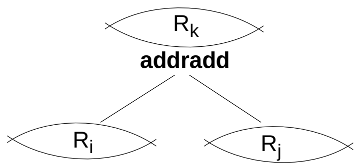
Value descriptors state how/where the value of a tree node is represented, e. g.

$\mathbf{R_i}$	value in register R_i
c	constant value c
$\mathbf{R_i,c}$	address R_i + c
(adr)	contents at the address adr

alternative **translation patterns** to be selected context dependend:



addradd R_i, c1 c2 -> R_i, c1 + c2 ./.



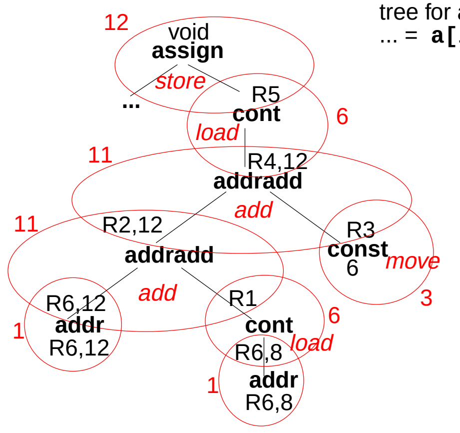
addradd R_i R_j -> R_k add R_i, R_j, R_k

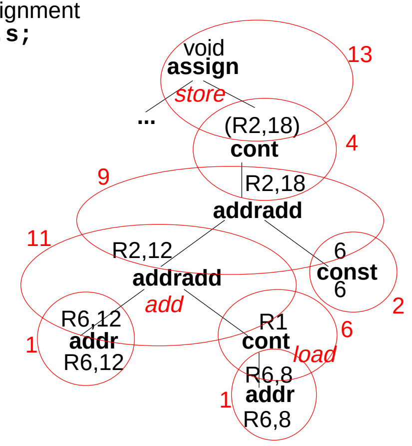
Example for a Set of Translation Patterns

#	operator	operands		result	code
1	addr	R_i, c		-> R_i,c	./.
2	const	c		-> c	./.
3	const	c		-> R_i	move c, R_i
4	cont	R_i, c		-> (R_i, c)	./.
5	cont	R_i		-> (R_i)	./.
6	cont	R_i, c		-> R_j	load (R_i, c), R_j
7	cont	R_i		-> R_j	load (R_i), R_j
8	addradd	R_i	c	-> R_i, c	./.
9	addradd	R_i, c1	c2	-> R_i, c1 + c2	./.
10	addradd	R_i	R_j	-> R_k	add Ri, R_j, R_k
11	addradd	R_i, c	R_j	-> R_k, c	add R_i, R_j, R_k
12	assign	R_i	R_j	-> void	store R_j, R_i
13	assign	R_i	(R_j, c)	-> void	store (R_j,c), R_i
14	assign	R_i,c	R_j	-> void	store R_j, R_i,c

Tree Covered with Translation Patterns

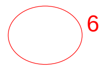
tree for assignment
... = `a[i].s;`





load (R6,8), R1
add R6,R1,R2
move 6,R3
add R2,R3,R4
load (R4,12),R5
store R5, ...

cost: 6 instructions



application of pattern #6

load (R6,8), R1
add R6,R1,R2
store (R2,18),...

cost: 3 instructions

Pattern Selection

Pass 1 bottom-up:

Annotate the nodes with sets of pairs
{ (v, c) | v is a kind of value descriptor that an applicable pattern yields,
c are the accumulated subtree costs}

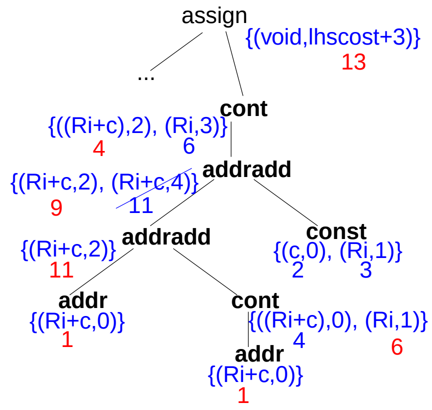
If (v, c1), (v, c2) keep only the cheaper pair.

Pass 2 top-down:

Select for each node the cheapest pattern, that fits to the selection made above.

Pass 3 bottom-up:

Emit code.



load (R6,8), R1
add R6,R1,R2
store (R2,18),...

cost: 3 instructions

Improved technique:

relative costs per sets =>
finite number of potential sets
integer encoding of the sets at generation time

Pattern Matching in Trees: Bottom-up Rewrite

Bottom-up Rewrite Systems (BURS) :
a general approach of the pattern matching method:

Specification in form of tree patterns, similar to C-3.18 - C-3.20

Set of patterns is **analyzed at generation** time.

Generator produces a **tree automaton** with a finite set of states.

On the bottom-up traversal it annotates each tree node with
a **set of states**:
those selection decisions which may lead to an optimal solution.

Decisions are made on the base of the **costs of subtrees** rather than costs of nodes.

Generator: BURG

Tree Pattern Matching by Parsing

The tree is represented in prefix form.

Translation patterns are specified by tuples (CFG production, code, cost),
Value descriptors are the nonterminals of the grammar, e. g.

8	RegConst ::= **addradd** Reg Const	nop	0
11	RegConst ::= **addradd** RegConst Reg	add R_i, R_j, R_k	1

Deeper patterns allow for more effective optimization:

Void ::= **assign** RegConst **addradd** Reg Const	store (Ri, c1),(Rj, c2)	1

Parsing for an ambiguous CFG:
application of a production is decided on the base of the production costs rather than the accumulated subtree costs!

Technique „Graham, Glanville“
Generators: GG, GGSS

4 Register Allocation

Use of registers:

1. intermediate **results of expression evaluation**
2. reused results of expression evaluation (CSE)
3. contents of frequently used **variables**
4. **parameters** of functions, **function result** (cf. register windowing)
5. stack pointer, **frame pointer**, heap pointer, ...

Number of registers is limited - for each register class: address, integer, floating point

Specific allocation methods for different context ranges:

- 4.1 expression trees (Sethi, Ullman)
- 4.2 basic blocks (Belady)
- 4.3 control flow graphs (graph coloring)

Register allocation aims at reduction of

- number of memory accesses
- spill code, i. e. instructions that store and reload the contents of registers

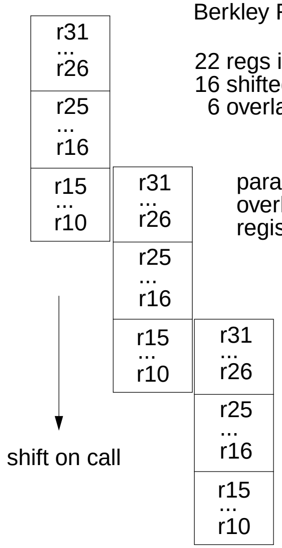
Symbolic registers: allocate a new symbolic register to each value assignment (single assignment, no re-writing); defer allocation of real registers to a later phase.

Register Windowing

Register windowing:

- Fast storage of the processor is accessed through a window.
- The n elements of the window are used as registers in instructions.
- On a call the window is shifted by m<n registers.
- Overlapping registers can be used under different names from both the caller and the callee.
- Parameters are passed without copying.
- Storage is organized in a ring; 4-8 windows; saved and restored as needed

Typical for Risc processors, e.g. Berkley RISC, SPARC



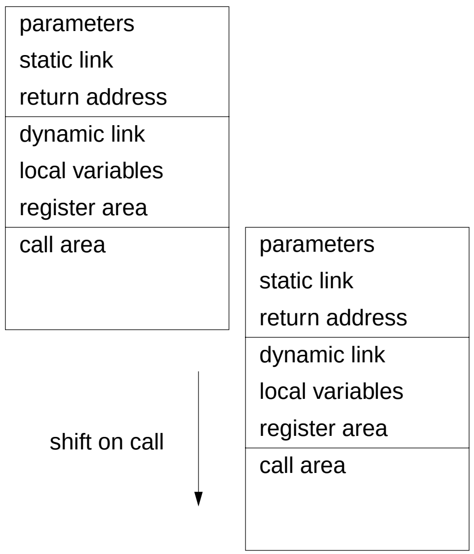
Berkley Risc:

22 regs in window
16 shifted
6 overlapped

parameters in overlapping registers

Activation Records in Register Windows

- **Parameters** are passed in overlap area **without copying**.
- **Registers need not be saved** explicitly.
- If **window is too small** for an activation record, the remainder is allocated on the **run-time stack**; pointer to it in window.



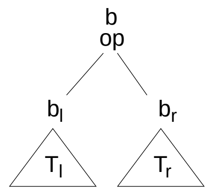
4.1 Register Allocation for Expression Trees

Problem:
Generate code for **expression** evaluation.
Intermediate results are stored in registers.
Not enough registers:
spill code saves and restores.

Goal:
Minimize amount of spillcode.
see C-4.5a for optimality condition

Basic idea (Sethi, Ullman):
For each subtree minimize the **number of needed registes**:

evaluate **first the subtree that needs most** registers



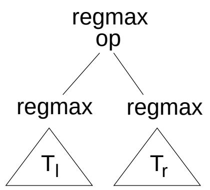
assume the results of T_l and T_r are in registers

eval. order	needed registers b =	
T_l T_r op	max (b_l, b_r + 1)	minimize
T_r T_l op	max (b_r, b_l + 1)	

number of available registers (regmax) is upper limit for needed registers

Expression Tree Attribution

Spill code needed:



Code (T_r)
store R_r, h
Code (T_l)
load h, R_r
op R_r, R_l

load h, R_r is not needed if h can be a memory operand in op h, R_l

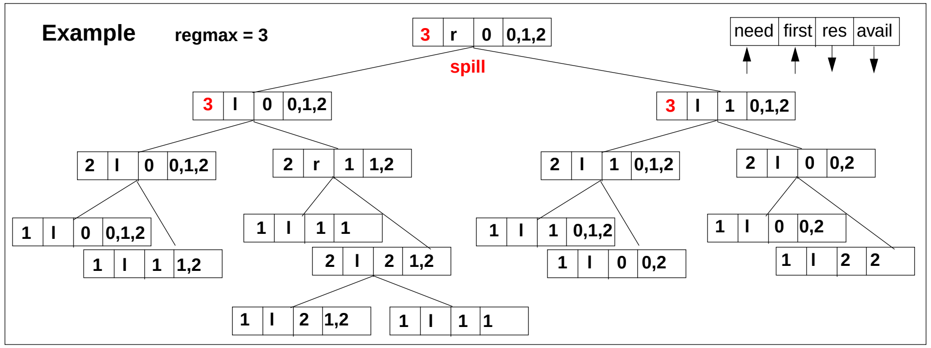
Implementation by attribution of trees:

Phase 1 bottom-up:
needed registers, evaluation order

Phase 2 top-down**:**
allocate registers

Phase 3 bottom-up:
compose code in evaluation order

Example regmax = 3



Contiguous code vs. optimal code

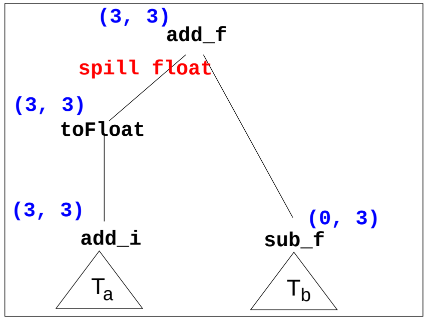
The method assumes that the **code for every subtree is contiguous**.
(I.e. there is no interleaving between the code of any two disjoint subtrees.)

The **method is optimal** for a certain **configuration of registers and operations**, iff every **optimal evaluation code** can be arranged to be **contiguous**.

Counter example:

Registers: 3 `int` and 3 `float`
Register need: (i, f) from (0, 0) to (3, 3)

Operations: `int`- and `float`- arithmetic, `toFloat` (widening)



register use:	(3, 3)	(1, 0)	(0, 1)	(0, 0)	(0, 3)	(0, 1)	(0, 2)	(0, 1)	
contiguous:	T_a	`add_i`	`toFloat`	`store_f`	T_b	`sub_f`	`load_f`	`add_f`	
optimal:	T_a	`add_i`	T_b	`sub_f`	`toFloat`	`add_f`			
register use:	(3, 3)	(1, 0)	(1, 3)	(1, 1)	(1, 2)	(0, 1)			

4.2 Register Allocation for Basic Blocks by Life-Time Analysis

Lifetimes of values in a basic block are used to minimize the number of registers needed.

1st Pass: Determine the **life-times** of values: from the definition to the last use (there may be several uses!).

Life-times are represented by intervals in a graph

cut of the graph = number of **registers needed** at that point

at the end of 1st pass:
maximal cut = number of register needed for the basic block

allocate registers **in the graph**:

In case of shortage of registers: select values to be **spilled**; **criteria**:

- a **value that is already in memory** - store instruction is saved

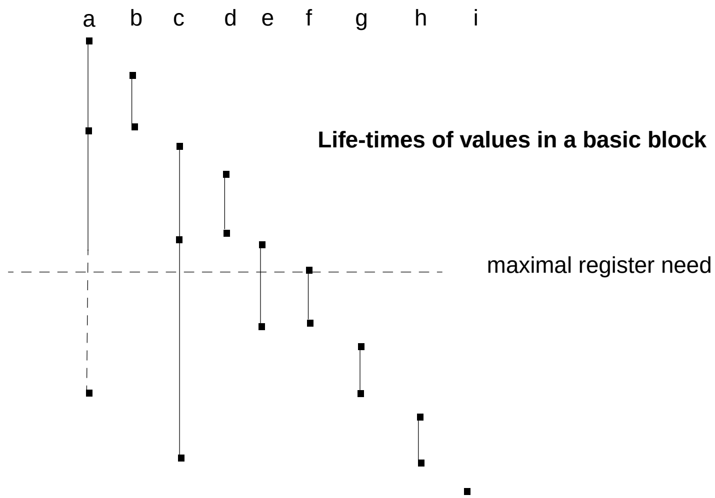
- the **value that is latest used again**

2nd Pass: allocate registers **in the instructions**; evaluation order remains unchanged

The technique has been presented originally 1966 by
Belady as a **paging technique for storage allocation**.

Example for Belady's Technique

a := x
b := y
c := b + a
d := z
e := d * c
f := s
g := e / f
h := g + a
i := h * c



register allocations

		a	b	c	d	e	f	g	h	i	
4 regs	(a)	d1	d2	d2	d3	d3	d4	d3	d3	d3	
3 regs	(b)	d1	d2	d2*	d3	d3	d2	d3	d3	d3	* spilled: store;...; load
3 regs	(c)	d1*	d2	d2	d3	d3	d1	d3	d3	d3	* spilled: reloaded from x

4.3 Register Allocation by Graph Coloring

Definitions and uses of variables in control-flow graphs for **function bodies** are analyzed (DFA).
Conflicting life-times are modelled. Presented by **Chaitin**.

Construct an interference graph:

Nodes: Variables that are candidates for being kept in registers

Edge {a, b}: **Life-times** of variables a and b overlap
=> a, b have to be kept in different registers

Life-times for CFGs are determined by **data-flow analysis**.

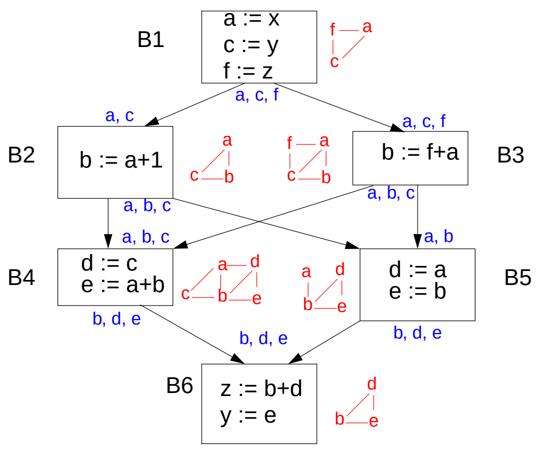
Graph is „colored“ with register numbers.

NP complete problem; **heuristic technique** for coloring with k colors (registers):

eliminate nodes of degree < k (and its edges)
if the graph is finally empty:
    graph can be colored with k colors
    assign colors to nodes in reverse order of elimination
else
    graph can not be colored this way
    select a node for spilling
    repeat the algorithm without that node

Example for Graph Coloring

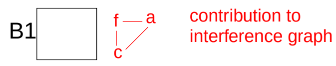
CFG with definitions and uses of variables

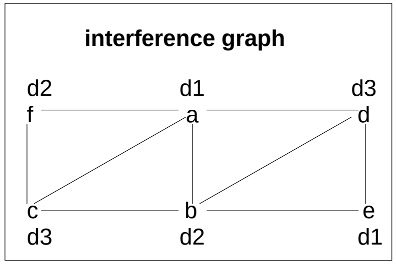


variables in memory: x, y, z

variables considered for register alloc.:
a, b, c, d, e, f

results of live variable analysis:
b, d, e

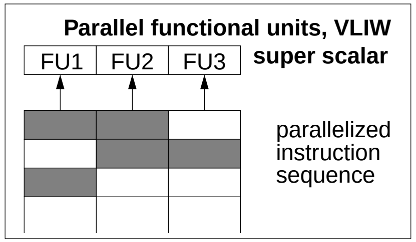




5 Code Parallelization

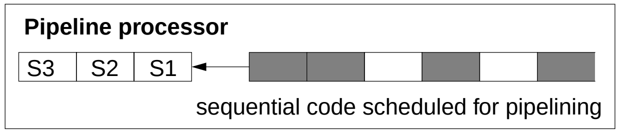
Processor with **instruction level parallelism (ILP)**
executes several instructions in parallel.

Classes of processors and parallelism:
VLIW, super scalar
Pipelined processors
Data parallel processors

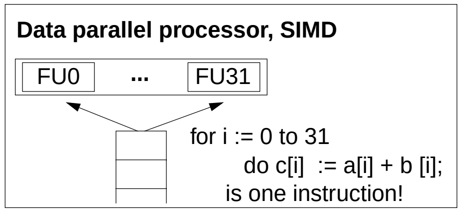


Compiler **analyzes sequential programs** to
exhibit potential parallelism
on instruction level;

model **dependences between computations**



Compiler arranges instructions for
shortest execution time:
instruction scheduling



Compiler **analyzes loops**
to execute them in parallel
loop transformation
array transformation

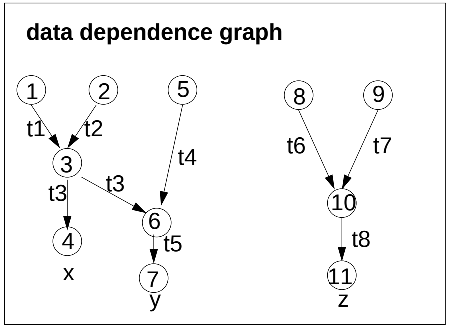
5.1 Instruction Scheduling Data Dependence Graph

Exhibit potential **fine-grained parallelism** among operations.
Sequential code is over-specified!

Data dependence graph (DDG) for a basic block:
Node: operation;
Edge a -> b: operation b uses the result of operation a

Example for a basic block:

1:	t1	:= a
2:	t2	:= b
3:	t3	:= t1 + t2
4:	x	:= t3
5:	t4	:= c
6:	t5	:= t3 + t4
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
10:	t8	:= t6 + t7
11:	z	:= t8

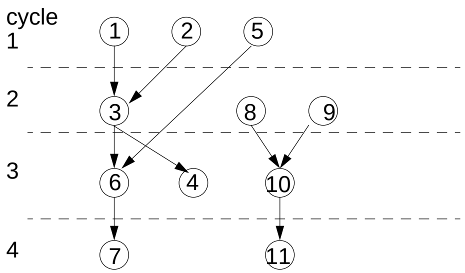


ti are symbolic registers, store intermediate results, obey single assignment rule

List Scheduling

Input: data dependence graph
Output: a schedule of **at most k operations per cycle**,
such that all **dependences point forward;** DDG arranged in levels

Algorithm: A **ready list** contains all operations that are **not yet scheduled**,
but whose **predecessors are scheduled**
Iterate: **select** from the ready list up to k operations for the next cycle (heuristic),
update the ready list



- Algorithm is **optimal** only for **trees**.
- **Heuristic**: Keep ready list sorted by distance to an end node, e. g.

  (1 2 5) (8 9 3) (6 10 4) (7 11)

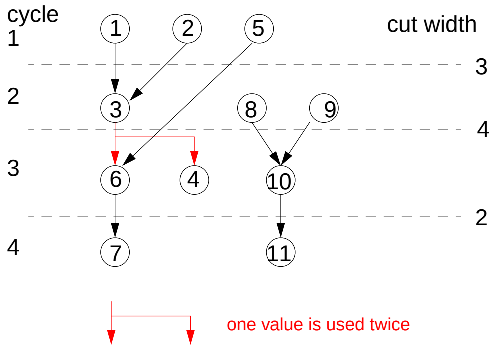
  without this heuristic:
  (1 8 9) (2 5 10) (3 11) (6 4) (7)

  ( ) operations in one cycle

Critical paths determine minimal schedule length: e. g. 1 -> 3 -> 6 -> 7

Variants and Restrictions for List Scheduling

- Allocate **as soon as possible**, ASAP (C-5.3); as **late** as possible, ALAP
- Operations have **unit execution time** (C-5.3); **different execution times:** selection avoids conflicts with already allocated operations
- Operations only on **specific functional units** (e. g. 2 int FUs, 2 float FUs)
- **Resource restrictions** between operations, e. g. <= 1 load or store per cycle



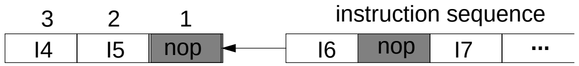
Scheduled DDG models **number of needed registers**:

- arc represents the use of an intermediate result
- **cut width** through a level gives the number of **registers needed**

The tighter the schedule the more registers are needed (*register pressure*).

Instruction Scheduling for Pipelining

Instruction pipeline
with 3 stages:



Dependent instructions may not follow one another immediately.

without scheduling:

1:	t1	:= a
2:	t2	:= b
	nop	
3:	t3	:= t1 + t2
	nop	
4:	x	:= t3
5:	t4	:= c
	nop	
6:	t5	:= t3 + t4
	nop	
7:	y	:= t5
8:	t6	:= d
9:	t7	:= e
	nop	
10:	t8	:= t6 + t7
	nop	
11:	z	:= t8

Schedule rearranges the operation sequence, to minimize the number of delays:

1:	t1	:= a
2:	t2	:= b
5:	t4	:= c
3:	t3	:= t1 + t2
8:	t6	:= d
9:	t7	:= e
6:	t5	:= t3 + t4
10:	t8	:= t6 + t7
4:	x	:= t3
7:	y	:= t5
11:	z	:= t8

with scheduling

no delays

Instruction Scheduling Algorithm for Pipelining

Algorithm: modified list scheduling:

Select from the ready list such that the selected operation

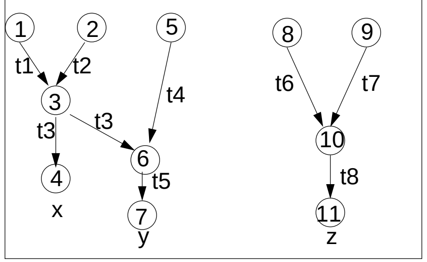
- has a sufficient **distance to all predecessors** in DDG
- has **many successors** (heuristic)
- has a **long path to the end** node (heuristic)

Insert an empty operation if none is selectable.

Ready list with additional information:

opr.	1	2	5	8	9	3	6	4	10	7	11
succ #	1	1	1	1	1	2	1	0	1	0	0
to end	3	3	2	2	2	2	1	1	1	0	0
sched. cycle	1	2	3	5	6	4	7	9	8	10	11

data dependence graph



cycle			
1	1:	t1	:= a
2	2:	t2	:= b
3	5:	t4	:= c
4	3:	t3	:= t1 + t2
5	8:	t6	:= d
6	9:	t7	:= e
7	6:	t5	:= t3 + t4
8	10:	t8	:= t6 + t7
9	4:	x	:= t3
10	7:	y	:= t5
11	11:	z	:= t8

with scheduling

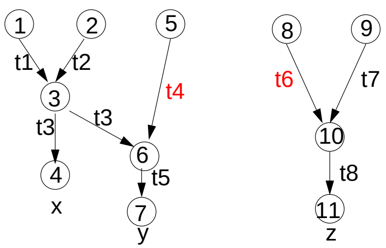
Reused registers: anti- and output-dependences

u → v **flow-dependence**:
u writes before v uses

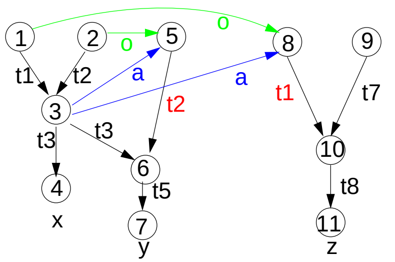
u –a→ v **anti-dependence**:
u uses a value
before v overwrites it

u –o→ v **output-dependence**:
u writes before v overwrites

DDG with symbolic registers ti
flow-dependences only



DDG with reused registers ti
flow, anti-, and output-dependences



DDG with Loop Carried Dependences

Factorial computation:

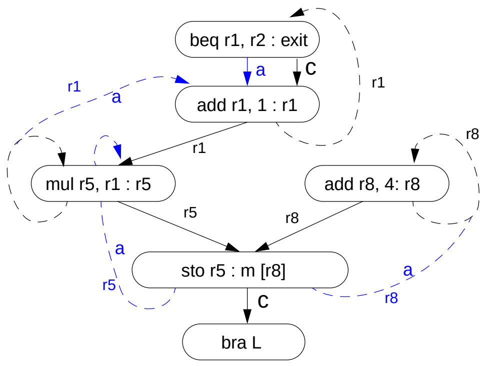
program:

```
i = 0; f = 1;
while ( i != n)
{   i = i + 1;
    f = f * i;

    m[i] = f;
}
```

seq. machine code:

```
L:  beq  r1, r2 : exit
    add  r1, 1  : r1
    mul  r5, r1 : r5
    add  r8, 4  : r8
    sto  r5 : m[r8]
    bra  L
```

Data dependence graph:



u → v **flow-dependence**:
u writes before v uses

u - - → v **flow-dependence** into
subsequent iteration

u - a → v **anti-dependence**:
u uses a value
before v overwrites it

u - o → v **output-dependence**:
u writes before v overwrites

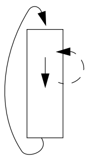
u –c→ v **control-dependence**:
u has to be executed before v
(u or v may branch)

Loop unrolling

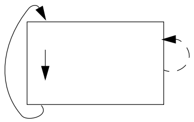
Loop unrolling: A technique for parallelization of loops.

A single loop body does not exhibit enough parallelism => sparse schedule.
Schedule the code (copies) of several adjacent iterations together
=> more compact schedule

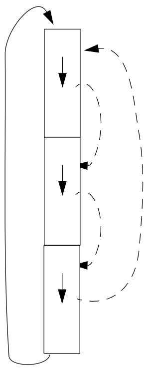
sequential loop



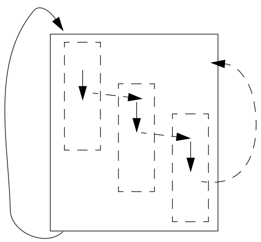
parallel schedule for single body



unrolled loop (3 times)



parallel schedule for unrolled loop



Prologue and epilogue needed to take care of iteration numbers that are not multiples of the unroll factor

Software Pipelining

Software Pipelining: A technique for parallelization of loops.

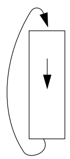
A single loop body does not exhibit enough parallelism => sparse schedule.
Overlap the execution of several adjacent iterations => compact schedule

The **pipelined loop body**

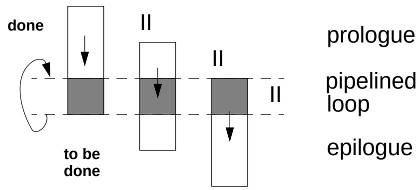
has **each operation** of the original sequential body,
they belong to **several iterations**,
they are **tightly scheduled,**
its length is the **initiation interval II**,
is **shorter** than the original body.

Prologue, epilogue: initiation and finalization code

sequential



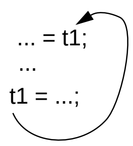
software pipelined



Transform Loops by Software Pipelining

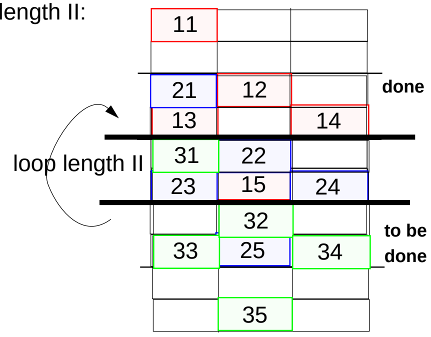
Technique:

1. **Data dependence graph** for the loop body, include **loop carried dependences**.
2. Chose a **small initiation interval II** - not smaller than #instructions / #FUs
3. Make a **„Modulo Schedule“** s for the loop body: Two instructions can not be scheduled on the same FU, i_1 in cycle c_1 and i_2 in cycle c_2, if c_1 mod II = c_2 mod II
4. If (3) does not succeed without conflict, increase II and repeat from 3
5. Allocate the instructions of s in the new loop of length II: i_j scheduled in cycle c_j is allocated to c_j mod II
6. Construct prologue and epilogue.



Modulo schedule for a loop body

cycle				
0	0	11		
1	1			
2	0		12	
3	1	13		14
4	0			
5	1		15	



Result of Software Pipelining

t	t_m		ADD	MUL	MEM	CTR
0	0	L:				beq r1, r2:exit
1	1		add r1, 1: r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		
3	1			... mul		
4	0				sto r5 : m r8	
5	1				... sto	
6	0					
7	1					bra L

4 dedicated FUs schedule of the loop body for II = 2

mul and sto need 2 cycles

add and sto in t_m=0, sto reads r8 before add writes it

bra not in cycle 6, it collides with beq: t_m=0

t	t_m		ADD	MUL	MEM	CTR
0	0					beq r1;r2:exit
1	1		add r1, 1 : r1			
2	0		add r8, 4 : r8	mul r5, r1 : r5		beq r1; r2 : ex
3	1		add r1, 1 : r1	... mul		
4	0	L:	add r8, 4 : r8	mul r5, r1 : r5	sto r5 : m r8	beq r1; r2 : ex
5	1		add r1, 1 : r1	... mul	... sto	bra L
6	1	ex:		... mul	... sto	
7	0				sto r5 : m r8	
8	1				... sto	
9	0					bra exit

prologue

software pipline with II = 2

epilogue

5.2 / 6. Data Parallelism: Loop Parallelization

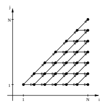
Regular loops on orthogonal data structures - parallelized for **data parallel** processors

Development steps (automated by compilers):

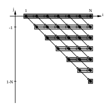
- **nested loops** operating on **arrays**, sequential execution of iteration space

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```

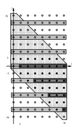
- analyze **data dependences** data-flow: definition and use of array elements



- **transform loops** keep data dependences forward in time



- **parallelize inner loop(s)** map to field or vector of processors
- **map arrays to processors** such that many accesses are local, transform index spaces



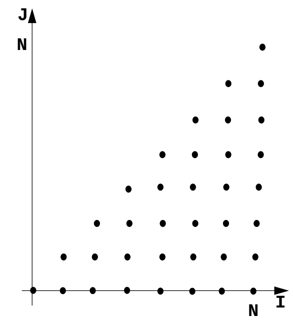
Iteration space of loop nests

Iteration space of a loop nest of depth n:

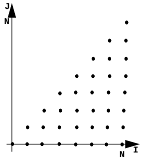
- **n-dimensional space of integral points** (polytope)
- each point (i_1, ..., i_n) represents an execution of the innermost loop body
- loop bounds are in general not known before run-time
- iteration need not have orthogonal borders
- iteration is elaborated sequentially

example:
computation of Pascal's triangle

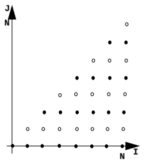
```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
    FOR J := 0 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```



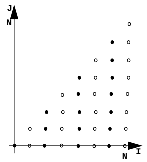
Examples for Iteration spaces of loop nests



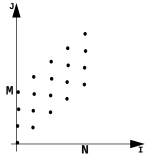

```
FOR I := 0 .. N
   FOR J := 0 .. I
```



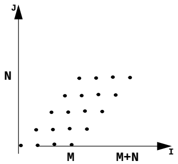

```
FOR I := 0 .. N
   FOR J := 0..I BY 2
```




```
FOR I := 0..N BY 2
   FOR J := 0 .. I
```




```
FOR I := 0 .. N
   FOR J := I..I+M

M = 3, N = 4
```

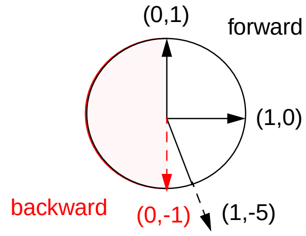



```
FOR I := 0 .. M+N
   FOR J := max(0, I-M)..
            min (I, N)
```


Data Dependences in Iteration Spaces

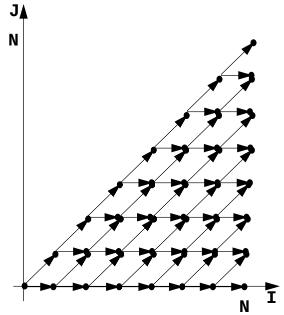
Data dependence from iteration point i1 to i2:

- Iteration **i1** computes a value that is used in iteration **i2** (flow dependence)
- relative **dependence vector**
  d = **i2 - i1** = ($i2_1$ - $i1_1$, ..., $i2_n$ - $i1_n$)
  holds for all iteration points except at the border
- Flow-dependences can **not be directed against the execution order**, can not point backward in time: each dependence vector must be **lexicographically positive**, i. e. **d** = (0, ..., 0, d_i, ...), d_i > 0



Example:
Computation of Pascal´s triangle

```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
   FOR J := 0 .. I
      B[I,J] :=
         B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```



Loop Transformation

The **iteration space** of a loop nest is transformed to **new coordinates**. Goals:

- **execute innermost loop(s) in parallel**
- improve **locality** of data accesses;
  in space: use storage of executing processor,
  in time: reuse values stored in cache
- **systolic** computation and communication scheme

Data dependences must **point forward in time**, i.e.
lexicographically positive and
not within parallel dimensions

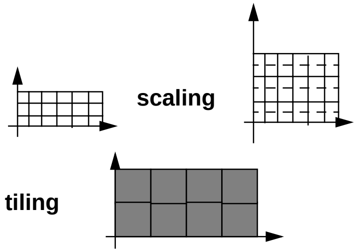
linear basic transformations:

- **Skewing**: add iteration count of an outer loop to that of an inner one
- **Reversal**: flip execution order for one dimension
- **Permutation**: exchange two loops of the loop nest

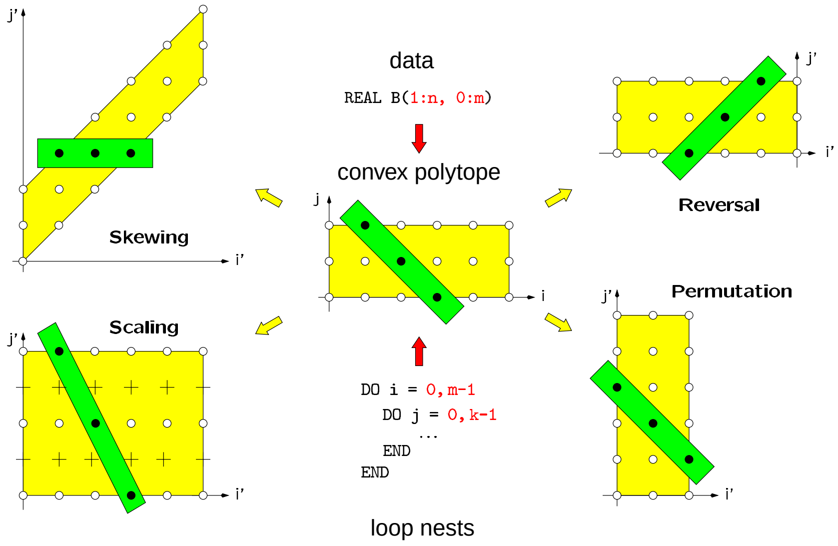
SRP transformations (next slides)

non-linear transformations, e. g.

- **Scaling**: **stretch the iteration space** in one dimension, causes gaps
- **Tiling**: introduce **additional inner loops** that **cover tiles** of fixed size



Transformations of



Transformations defined by matrices

Transformation matrices: systematic transformation, check dependence vectors

Reversal $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$

Skewing $\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$

Permutation $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$

Reversal

Iteration count of one loop is negated, that dimension is enumerated backward

general transformation matrix

$$\begin{pmatrix} 1 & & & & & 0 \\ & \ldots & & & & \\ & & 1 & & & \\ & & & \mathbf{-1} & & \\ & & & & 1 & \ldots \\ 0 & & & & & 1 \end{pmatrix}$$

2-dimensional:

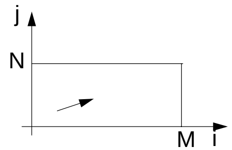
loop variables
old new

$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} ir \\ jr \end{pmatrix}$

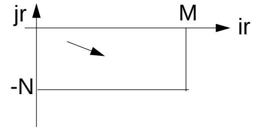
```
for i = 0 to M
   for j = 0 to N
      ...
```

```
for ir = 0 to M
   for jr = -N to 0
      ...
```

original



transformed



Skewing

The **iteration count** of an outer loop is **added to the count of an inner loop;**
iteration space is shifted; **execution order** of iteration points **remains unchanged**

general transformation matrix:

$$\begin{pmatrix} 1 & & & & & 0 \\ & \ldots & & & & \\ & & 1 & & & \\ & \mathbf{f} & & 1 & & \\ & & & & 1 & \ldots \\ 0 & & & & & 1 \end{pmatrix}$$

2-dimensional:

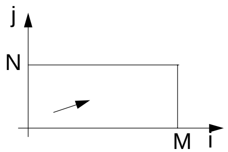
loop variables
old new

$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} is \\ js \end{pmatrix}$

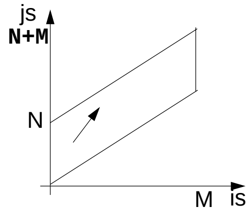
```
for i = 0 to M
   for j = 0 to N
      ...
```

```
for is = 0 to M
   for js = f*is to N+f*is
      ...
```

original



transformed



Permutation

Two loops of the loop nest are interchanged; the iteration space is flipped;
the **execution order** of iteration points **changes;** new dependence vectors must be legal.

general transformation matrix:

$$\begin{array}{c} \\ i \\ \\ j \\ \\ \\ \end{array} \begin{pmatrix} 1 & & & & & 0 \\ & \mathbf{0} & \mathbf{1} & & & \\ & & 1 & & & \\ & \mathbf{1} & \mathbf{0} & & & \\ 0 & & & 1 & \ldots & \\ & & & & & 1 \end{pmatrix}$$
$$\quad i \quad j$$

2-dimensional:

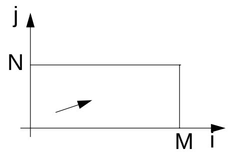
loop variables
old new

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} ip \\ jp \end{pmatrix}$

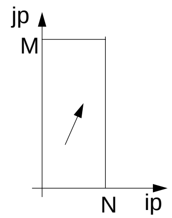
```
for i = 0 to M
   for j = 0 to N
      ...
```

original

```
for ip = 0 to N
   for jp = 0 to M
      ...
```



transformed



Use of Transformation Matrices

- Transformation matrix **T** defines **new iteration counts** in terms of the old ones: **T * i = i´**

  e. g. Reversal $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$

- Transformation matrix **T** transforms old **dependence vectors** into new ones: **T * d = d´**

  e. g. $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$

- inverse Transformation matrix $\mathbf{T}^{-1}$ defines **old iteration counts** in terms of new ones, for transformation of index expressions in the loop body: $\mathbf{T}^{-1} * \mathbf{i'} = \mathbf{i}$

  e. g. $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} * \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} i' \\ -j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}$

- **concatenation of transformations** first T_1 then T_2 : $\mathbf{T_2 * T_1 = T}$

  e. g. $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$

Inequalities Describe Loop Bounds

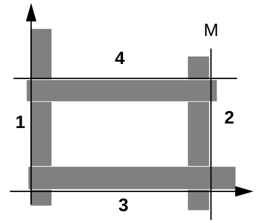
The bounds of a loop nest are described by a **set of linear inequalities**.
Each **inequality separates the space** in „inside and outside of the iteration space":

$$\mathbf{B * i \le c}$$

example 1

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \le \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

1 $-i \le 0$
2 $i \le M$
3 $-j \le 0$
4 $j \le N$

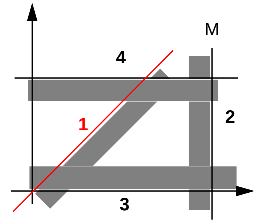


example 2

$$\begin{pmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \le \begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}$$

1 $-i + j \le 0$
2 $i \le M$
3 $-j \le 0$
4 $j \le N$

transformed



positive factors represent **upper** bounds
negative factors represent **lower** bounds

1, 4: $j \le \min(i, N)$ **1+ 3:** $0 \le i$
3: $0 \le j$ **2:** $i \le M$

Transformation of Loop Bounds

The inverse of a transformation matrix $\mathbf{T}^{-1}$ transforms a set of inequalities: $\mathbf{B * T^{-1} i' \le c}$

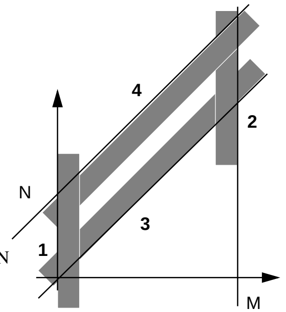
skewing $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ inverse $\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$

$$\underset{B}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}} * \underset{T^{-1}}{\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}} = \underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}}$$

example 1
new bounds:

$$\underset{B * T^{-1}}{\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}} * \underset{i'}{\begin{pmatrix} i' \\ j' \end{pmatrix}} \le \underset{c}{\begin{pmatrix} 0 \\ M \\ 0 \\ N \end{pmatrix}}$$

1 $-i' \le 0$
2 $i' \le M$
3 $i' - j' \le 0$
4 $-i' + j' \le N$



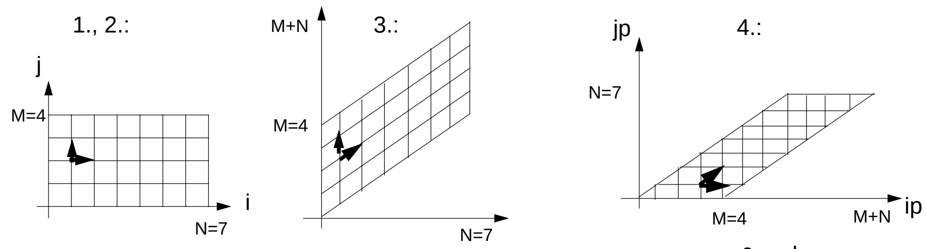
Example for Transformation and Parallelization of a Loop

```
for i = 0 to N
   for j = 0 to M
      a[i, j] = (a[i, j-1] + a[i-1, j]) / 2;
```

Parallelize the above loop.

1. Draw the iteration space.
2. Compute the dependence vectors and draw examples of them into the iteration space.
   Why can the inner loop not be executed in parallel?
3. Apply a skewing transformation and draw the iteration space.
4. Apply a permutation transformation and draw the iteration space.
   Explain why the inner loop now can be executed in parallel.
5. Compute the matrix of the composed transformation and
   use it to transform the dependence vectors.
6. Compute the inverse of the transformation matrix and
   use it to transform the index expressions.
7. Specify the loop bounds by inequalities and
   transform them by the inverse of the transformation matrix.
8. Write the complete loops with new loop variables ip and jp and new loop bounds.

Solution of the Transformation and Parallelization Example



5.: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $\quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

6.: Inverse $\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$

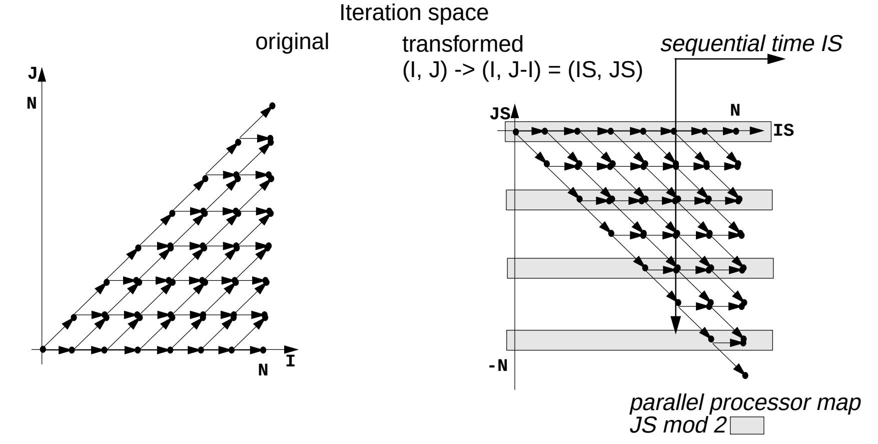
7. Bounds: B c B * T $^{-1}$

orig.: $\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ N \\ 0 \\ M \end{pmatrix}$ new: $\begin{pmatrix} 0 & -1 \\ 0 & 1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix}$

1 $-jp \le 0$ **1, 3 => $0 \le ip$**
2 $jp \le N$ **2, 4 => $ip \le M+N$**
3 $-ip+jp \le 0$ **1, 4 => $\max(0, ip-M) \le jp$**
4 $ip - jp \le M$ **2, 3 => $jp \le \min(ip, N)$**

8.
```
for ip = 0 to M+N
   for jp = max (0, ip-M) to min (ip, N)
      a[jp, ip-jp] = (a[jp, ip-jp-1] + a[jp-1, ip-jp]) / 2;
```


Transformation and Parallelization




```
DECLARE B[-1..N,-1..N]

FOR I := 0 .. N
   FOR J := 0 .. I
      B[I,J] :=
         B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```

```
DECLARE B[-1..N,-1..N]

FOR IS := 0.. N
   FOR JS := -IS .. 0
      B[IS,JS+IS] :=
         B[IS-1,JS+IS]+B[IS-1,JS-1+IS]
   END FOR
END FOR
```


Data Mapping

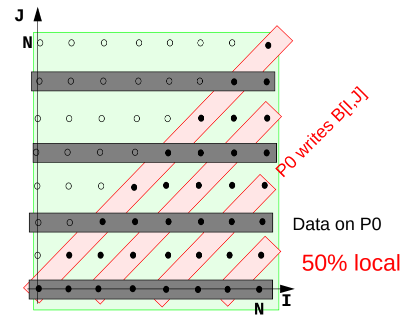
Goal:
Distribute array elements over processors, such that as many **accesses as possible are local.**

Index space of an array:
n-dimensional space of integral index points (polytope)

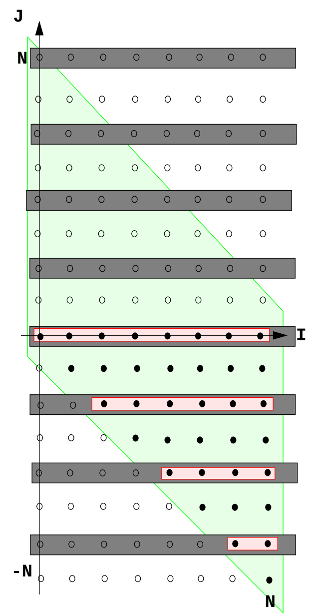
- **same properties as iteration space**
- same mathematical model
- same **transformations** are applicable (Skewing, Reversal, Permutation, ...)
- **no restrictions** by data dependences

Data distribution for parallel loops

index space of B
original transformed
skewing f=-1
(i,j) -> (i,j-i)



100%local




```
DECLARE B[-1..N,-1..N]

FOR IS := 0.. N
   FOR JS := -IS .. 0
      B[IS,JS+IS] :=
         B[IS-1,JS+IS]+B[IS-1,JS-1+IS]
   END FOR
END FOR
```

```
DECLARE B[-1..N,-N..N]
...
   B[IS,JS] :=
      B[IS-1,JS-1]+B[IS-1,JS-1]
```

Check Your Knowledge (1)

Optimization, CFA:

1. Explain graphs that are used in program analysis.
2. Which optimizing transformations need analysis of execution pathes?
3. Which optimizing transformations do not need analysis of execution pathes?
4. Give an example for a pair of transformations such that one enables the other.
5. Define the control-flow graph. Describe transformations on the CFG.
6. Define the dominator relation. What is it used for?
7. Describe an algorithm for computing dominator sets.
8. Define natural loops.
9. What is the role of the loop header and of the pre-header.
10. Show a graph that has a cycle but no natural loop.
11. Define induction variables, and explain the transformation technique.

Check Your Knowledge (2)

Optimization, DFA:

12. Describe the schema for DFA equations for the four problem categories.
13. Explain the relation of the meet operator, the paths in the graph, and the DFA solutions.
14. Describe the DFA problem reaching definitions.
15. Describe the DFA problem live variables.
16. Describe the DFA problem available expressions.
17. Describe the DFA problem copy propagation.
18. Describe the DFA problem constant propagation.
19. Describe the iterative DFA algorithm; its termination; its complexity.
20. Describe an heuristic improvement of the iterative DFA algorithm.
21. Extend constant propagation to interval propagation for bounds checks. Explain the interval lattice.
22. What is the role of lattices in DFA?
23. Describe lattices that are common for DFA.

Check Your Knowledge (3)

Object Oriented Program Analysis:

24. Describe techniques to reduce the number of arcs in call graphs.
25. Describe call graphs for object oriented programs.
26. Describe techniques to reduce the number of arcs in object oriented call graphs.

Code Generation, Storage mapping:

27. Explain the notions of storage classes, relative addresses, alignment, overlay.
28. Compare storage mapping of arrays by pointer trees to mapping on contiguous storage.
29. Explain storage mapping of arrays for C. What is different for C, for Fortran?
30. For what purpose are array descriptors needed? What do they contain?
31. What is the closure of a function? In which situation is it needed?
32. Why must a functional parameter in Pascal be represented by a pair of pointers?
33. What does an activation record contain?
34. Explain static links in the run-time stack. What is the not-most-recent property?
35. How do C, Pascal, and Modula-2 ensure that the run-time stack discipline is obeyed?
36. Why do threads need a separate run-time stack each?

Check Your Knowledge (4)

37. Explain the code for function calls in relation to the structure of activation records.
38. Explain addressing relative to activation records.
39. Explain sequences for loops.
40. Explain the translation of short circuit evaluation of boolean expressions. Which attributes are used?
41. Explain code selection by covering trees with translation patterns.
42. Explain a technique for tree pattern selection using 3 passes.
43. Explain code selection using parsing. What is the role of the grammar?

Register Allocation

44. How is register windowing used for implementation of function calls?
45. Which allocation technique is applied for which program context?
46. Explain register allocation for expression trees. Which attributes are used?
47. How is spill code minimized for expression trees?
48. Explain register allocation for basic blocks? Relate the spill criteria to paging techniques.
49. Explain register allocation by graph coloring. What does the interference graph represent?
50. Explain why DFA life-time analysis is needed for register allocation by graph coloring.

Check Your Knowledge (5)

Instruction Scheduling

51. What does instruction scheduling mean for VLIW, pipeline, and vector processors?
52. Explain the kinds of arcs of DDGs (flow, anti, output).
53. What are loop carried dependences?
54. Explain list scheduling for parallel FUs. How is the register need modelled? Compare it to Belady’s register allocation technique.
55. How is list scheduling applied for arranging instructions for pipeline processors?
56. Explain the basic idea of software pipelining. What does the initiation interval mean?

Loop Parallelization

57. Explain dependence vectors in an iteration space. What are the admissible directions for sequential and for parallelized innermost loops?
58. What is tiling, what is scaling?
59. Explain SRP transformations.
60. How are the transformation matrices used?
61. How are loop bounds transformed?
62. Parallelize the inner loop of a nest that has dependence vectors (1,0) and (0, 1)?