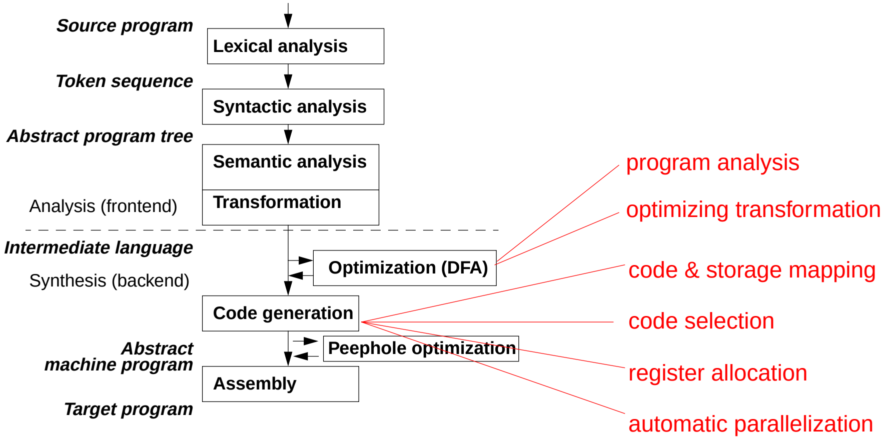
6. Synthesis: An Overview



Optimization

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

Input: Program in intermediate language
Task: **Analysis** (find redundancies), apply **transformations**
Output: Improved program in intermediate language

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

Context of analysis:

Expression	local optimization	
Basic block	local optimization	
Control flow graph (procedure)	global	intra-procedural optimization
Control flow graph, call graph	global	inter-procedural optimization

Optimizing Transformations

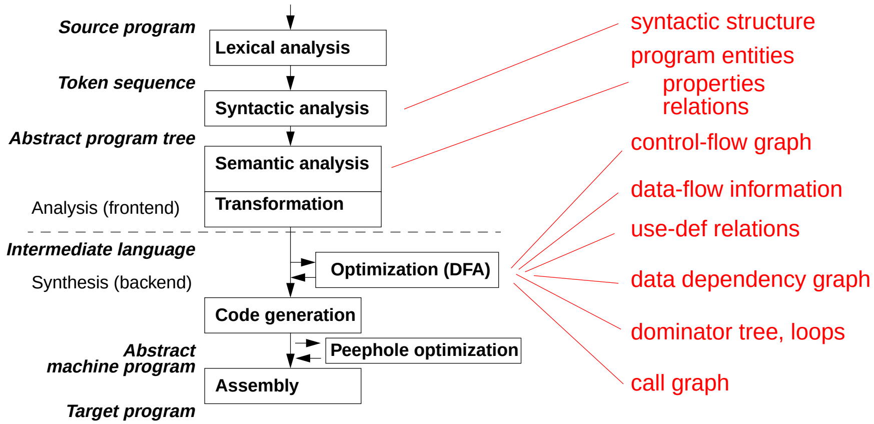
Name of transformation:	Example for its application:
• Algebraic simplification of expressions	`2*3.14   x+0   x*2   x**2`
• Constant propagation (dt. Konstantenweitergabe)	`x = 2; ... y = x * 5;`
• Common subexpressions (Gemeinsame Teilausdrücke)	`x=a*(b+c);...y=(b+c)/2;`
• Dead variables (Überflüssige Zuweisungen)	`x = a + b; ... x = 5;`
• Copy propagation (Überflüssige Kopieranweisungen)	`x = y; ... ; z = x;`
• Dead code (nicht erreichbarer Code)	`b = true;...if (b) x = 5; else y = 7;`
• Code motion (Code-Verschiebung)	`if (c) x = (a+b)*2; else x = (a+b)/2;`
• Function inlining (Einsetzen von Aufrufen)	`int Sqr (int i) { return i * i; }`
• Loop invariant code	`while (b) {... x = 5; ...}`
• Induction variables in loops	`i = 1; while (b) { k = i*3; f(k); i = i+1;}`

Analysis checks **preconditions for safe application** of each transformation;
more applications, if preconditions are analysed in **larger contexts**.

Interdependences:
Application of a transformation may **enable or inhibit** another application of a transformation.

Order of transformations is relevant.

Analysis in Compilers

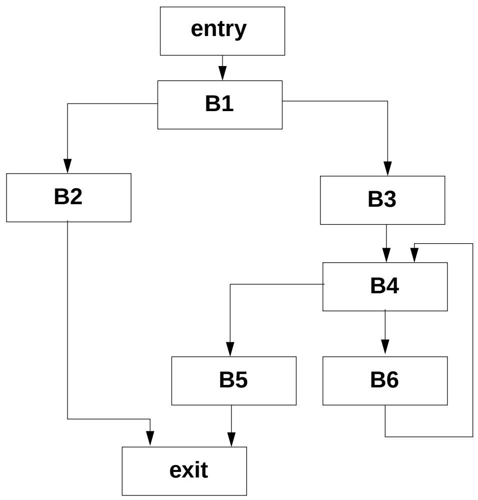


Example for a Control-flow Graph

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]



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

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

Its results are to be interpreted **pessimistic**.

Specification of a DFA Problem

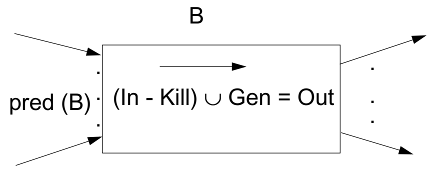
Specification of reaching definitions:

- **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.
- It is a **forward problem**.
- The **meet operator** is union.
- The **analysis information** in the sets are assignments at certain program positions.
- **Gen (B)**:
  contains all definitions **`d: v = e;`** in **B**,
  such that **v** is not defined after **d** in **B**.
- **Kill (B)**:
  if v is assigned in B, then Kill(B)
  contains all definitions **`d: v = e;`**
  in blocks different from **B**,
  such that **B** has a definition of **v**.

2 equations for each basic block:

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

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

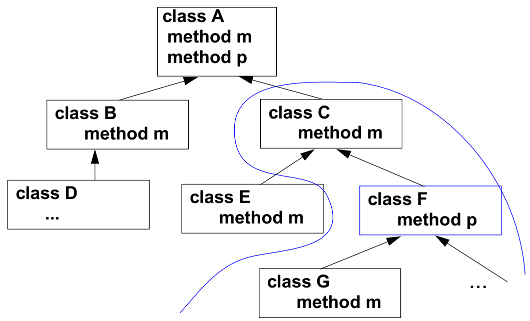


Call Graphs for object-oriented programs

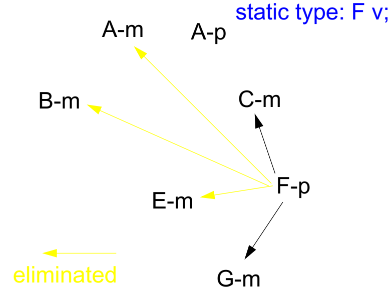
The call graph is reduced to a set of **reachable methods** using the **class hierarchy** and the **static type of the receiver** expression in the call:

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(...)



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

Storage Mapping

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

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

Design the use of storage areas:

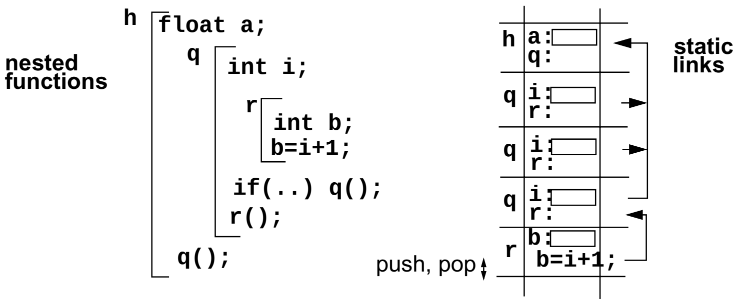
code storage	progam code
global data	to be linked for all compilation units
run-time stack	activation records for function calls
heap	storage for dynamically allocated objects, garbage collection
registers for	addressing of storage areas (e. g. stack pointer) function results, arguments local variables,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

Design the type mapping ... C-29

Run-Time Stack

Run-time stack contains one **activation record** for each active function call.
Activation record provides storage local data of a function call. (see C-31)

Nested functions (nested classes and objects): static predecessor chain
links the accessible activation records, **closure of a function**



Requirement: The closure of a function is still on the run-time stack when the function is called.

Languages without recursive functions (FORTRAN) do not use a run-time stack.

Optimization: activation records of **non-recursive functions** may be allocated statically.

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

Code Sequences for Control Statements

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

Several variants of code sequences may be defined for one statement.

Example:

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

variant:

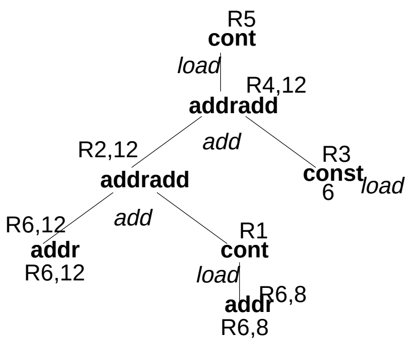
```
                                 goto M2
                            M1:  Code (Body)
                            M2:  Code (Condition, true, M1)
```

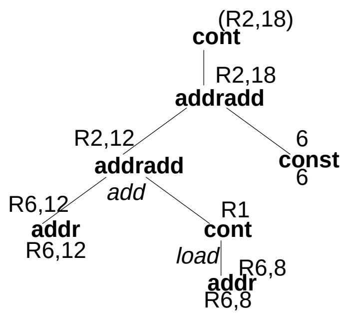
Meaning of the **Code** constructs:

`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

Example for Code Selection

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





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

cost: 6 instructions

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

cost: 3 instructions

Register Allocation

Use of registers:

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

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

register allocation aims at ruduction of

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

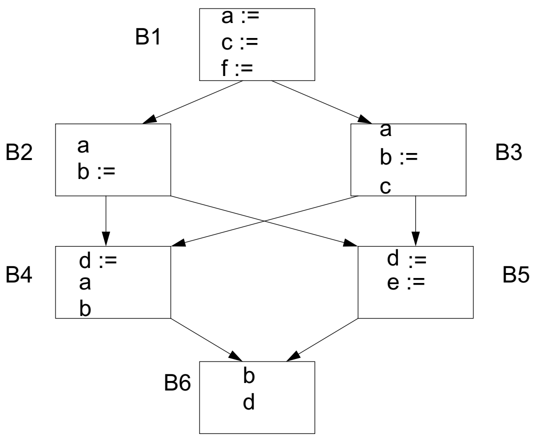
specific allocation methods for different context ranges:

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

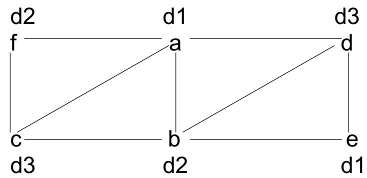
useful technique: defer register allocation until a later phase,
use an unbound set of **symbolic registers** instead

Example for Graph Coloring

CFG with definitions and uses of variables



interference graph

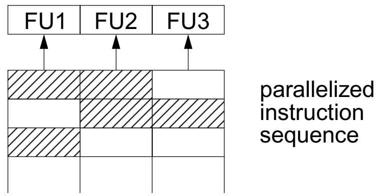


Code Parallelization

Target processor executes several instructions in parallel.
Compiler arranges instruction sequence for shortest execution time: **instruction scheduling**

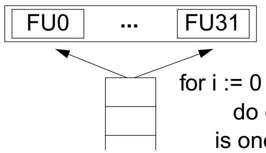
Principles of parallelism in processors:

Parallel functional units (FU)
super scalar, VLIW:



Data parallel processor
vector processor

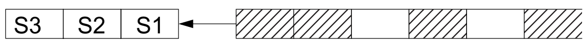
all FUs execute the same instruction
on individual data (SIMD)



for i := 0 to 31
do c[i] := a[i] + b [i];
is one instruction!

Analyze and transform loops

Pipeline processor



sequential code scheduled for pipelining

Software Pipelining

Technique for parallelization of loops.

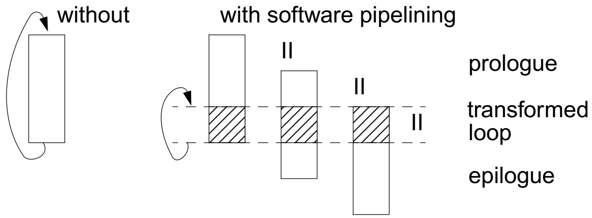
A single loop body does not exhibit enough parallelism => sparse schedule.

Idea of software pipelining:
transformed loop body executes several loop iterations in parallel,
iterations are shifted in time => compact schedule

Prologue, epilogue: initiation and finalization code

Technique:

1. **DDG** for loop body with dependencies into later iterations
2. Find a schedule such that iterations can begin with a **short initiation interval II**
3. Construct new loop, prologue, and epilogue



II: Initiation Interval

Loop Parallelization

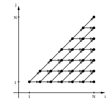
Compilation steps:

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

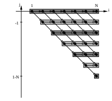
```
DECLARE B[0..N,0..N+1]

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 dependencies** data-flow: definition and use of array elements



- **transform loops** keep data dependencies intact
- **parallelize inner loop(s)** map onto field or vector of processors



- **map arrays onto processors** such that many acceses are local, transform index spaces