Compilation Methods

Prof. Dr. Uwe Kastens

Summer 2013

Lecture Compilation Methods SS 2013 / Slide 101

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

Lecture Compilation Methods SS 2013 / Slide 102

Objectives:

The objectives of the course

In the lecture:

The objectives are explained.

Questions:

- What are your objectives?
- Do they match to these objectives?

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	

Lecture Compilation Methods SS 2013 / Slide 104

Objectives:

Overview over the topics of the course

In the lecture:

Comments on the topics

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

Lecture Compilation Methods SS 2013 / Slide 105

Objectives:

Useful books and electronic material in the web

In the lecture:

Comments on the items:

- The material for this course is available.
- The material of "Programming Languages and Compilers" (every winter semester) is a prerequisite for this course.
- The book "Übersetzerbau" isn’t sold anymore. It is available in the library.
- The book by Muchnick contains very deep and concrete treatment of most important topics for optimizing compilers.

Questions:

- Find the referenced material in the web, become familiar with its structure, and set bookmarks for it.

Course Material in the Web: HomePage

Fachgruppe Kastens > Lehre > Compilation Methods SS 2013

Slides
Assignments
Organization
News
Koala

SUCHEN:

Lecture Compilation Methods SS 2013

Slides	Assignments
• Chapters • Slides • Printing	• Assignments • Printing

Organization	Ressources
• General Information • News	• Objectives • Literature • Contents *Kastens: Übersetzerbau* • Internet Links • Material: Programming Languages and Compilers

Veranstaltungs-Nummer: L.079.05810

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 19.02.2013

Lecture Compilation Methods SS 2013 / Slide 106

Objectives:

The root page of the course material.

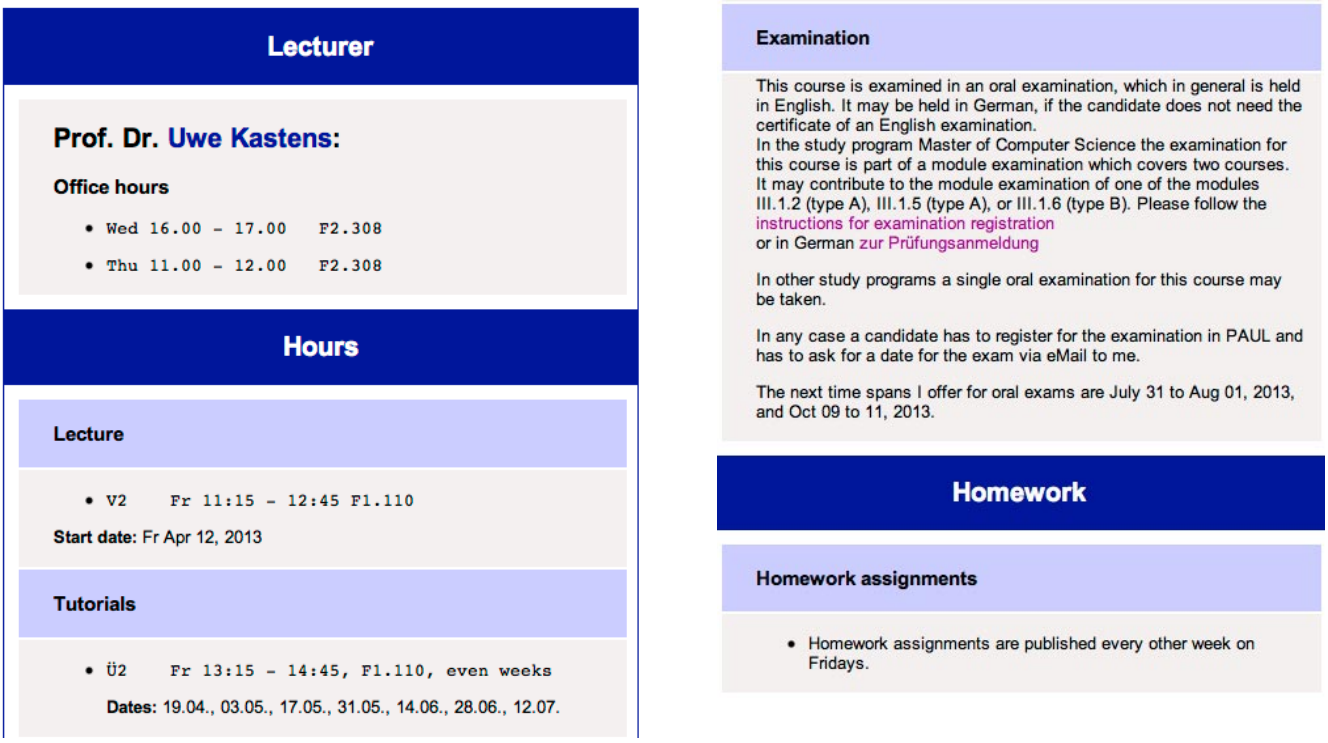
In the lecture:

The navigation structure is explained.

Assignments:

Explore the navigation structure.

Course Material in the Web: Organization



Lecturer

Prof. Dr. Uwe Kastens:

Office hours

- `Wed 16.00 - 17.00   F2.308`
- `Thu 11.00 - 12.00   F2.308`

Hours

Lecture

- `V2   Fr 11:15 - 12:45 F1.110`

Start date: Fr Apr 12, 2013

Tutorials

- `Ü2   Fr 13:15 - 14:45, F1.110, even weeks`

Dates: 19.04., 03.05., 17.05., 31.05., 14.06., 28.06., 12.07.

Examination

This course is examined in an oral examination, which in general is held in English. It may be held in German, if the candidate does not need the certificate of an English examination.
In the study program Master of Computer Science the examination for this course is part of a module examination which covers two courses. It may contribute to the module examination of one of the modules III.1.2 (type A), III.1.5 (type A), or III.1.6 (type B). Please follow the instructions for examination registration
or in German zur Prüfungsanmeldung

In other study programs a single oral examination for this course may be taken.

In any case a candidate has to register for the examination in PAUL and has to ask for a date for the exam via eMail to me.

The next time spans I offer for oral exams are July 31 to Aug 01, 2013, and Oct 09 to 11, 2013.

Homework

Homework assignments

- Homework assignments are published every other week on Fridays.

Lecture Compilation Methods SS 2013 / Slide 106a

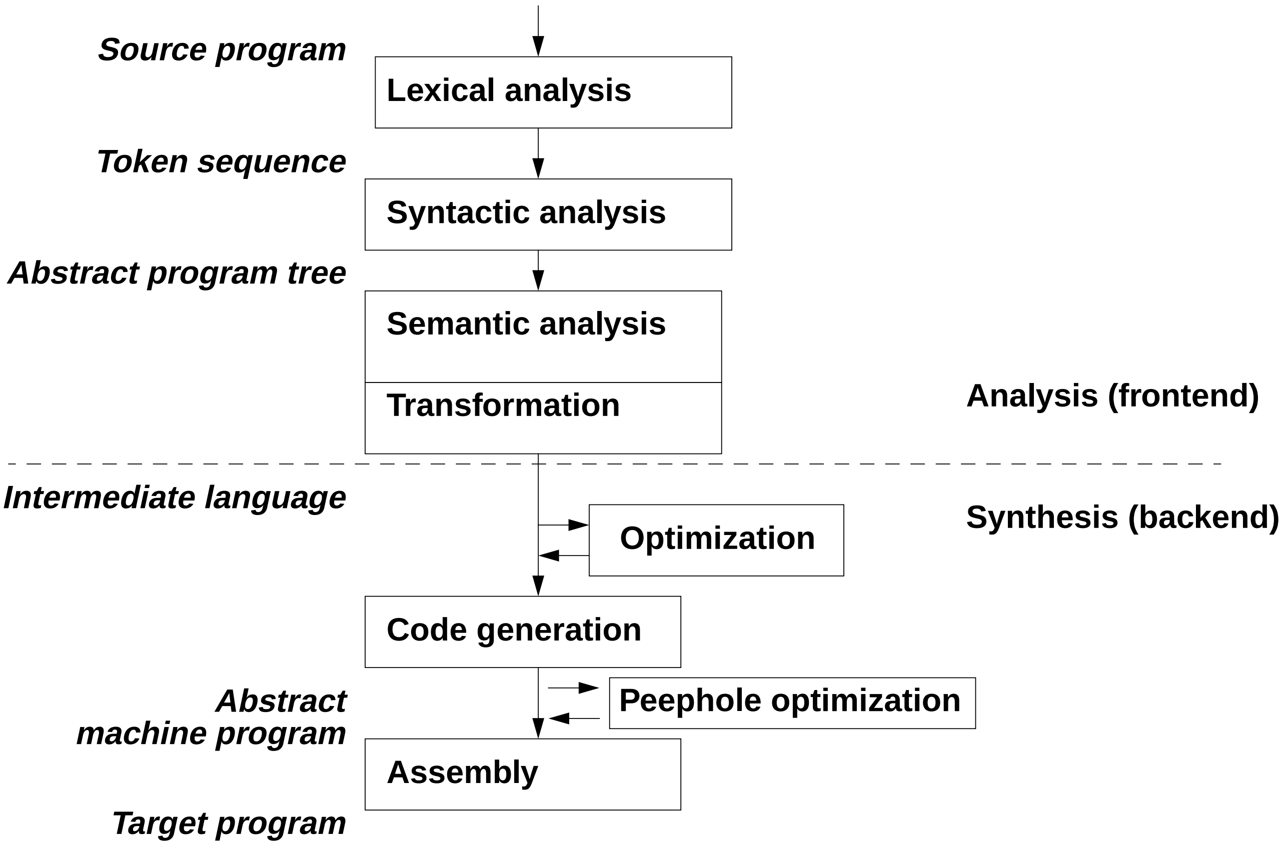
Objectives:

Agree on organizational items

In the lecture:

Check organizational items

Compiler Structure and Interfaces



Source program

Lexical analysis

Token sequence

Syntactic analysis

Abstract program tree

Semantic analysis

Transformation

Analysis (frontend)

Intermediate language

Optimization

Synthesis (backend)

Code generation

Abstract machine program

Peephole optimization

Assembly

Target program

Lecture Compilation Methods SS 2013 / Slide 107

Objectives:

Recall compiler structure and interfaces

In the lecture:

In this course we focus on the synthesis phase (backend).

Suggested reading:

Kastens / Übersetzerbau, Section 2.1

Assignments:

Compare this slide with U-08 and learn the translations of the technical terms used here.

2 Optimization

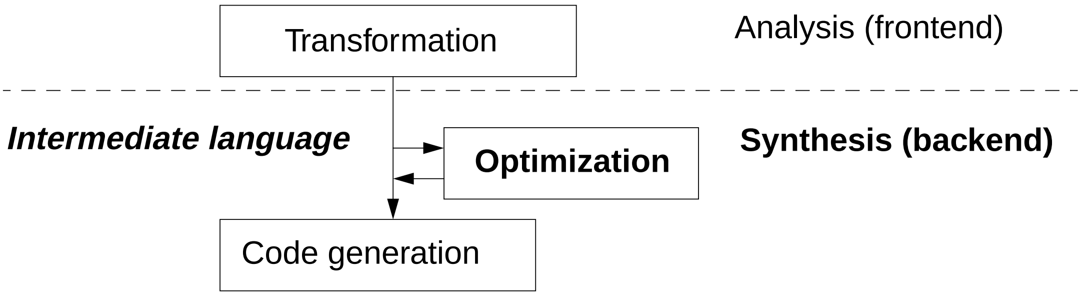
Objective:

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



Lecture Compilation Methods SS 2013 / Slide 201

Objectives:

Overview over optimization

In the lecture:

- Program analysis computes safe assertions at compile time about execution of the program.
- Conventionally this phase is called "Optimization", although in most cases a formal optimum can not be defined or achieved with practical effort.

Suggested reading:

Kastens / Übersetzerbau, Section 8

Questions:

Give examples for observable effects that may not be changed.

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;`

Lecture Compilation Methods SS 2013 / Slide 202

Objectives:

Get an idea of important transformations

In the lecture:

- The transformations are explained.
- The preconditions are discussed for some of them.

Suggested reading:

Kastens / Übersetzerbau, Section 8.1

Assignments:

- Apply as many transformations as possible in a given example program.

Questions:

- Which of the transformations need to analyze pathes through the program?
- Give an example for a pair of transformations, such that an application of the first one enables an application of the second.

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;}`

Lecture Compilation Methods SS 2013 / Slide 202a

Objectives:

Get an idea of important transformations

In the lecture:

- The transformations are explained.
- The preconditions are discussed for some of them.

Suggested reading:

Kastens / Übersetzerbau, Section 8.1

Assignments:

- Apply as many transformations as possible in a given example program.

Questions:

- Which of the transformations need to analyze pathes through the program?
- Give an example for a pair of transformations, such that an application of the first one enables an application of the second.

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**

Lecture Compilation Methods SS 2013 / Slide 203

Objectives:

Overview over optimization

In the lecture:

- Program analysis computes safe assertions at compile time about execution of the program.
- The larger the analysis context, the better the information, the more positions where transformations are applicable.

Suggested reading:

Kastens / Übersetzerbau, Section 8

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**

Lecture Compilation Methods SS 2013 / Slide 204

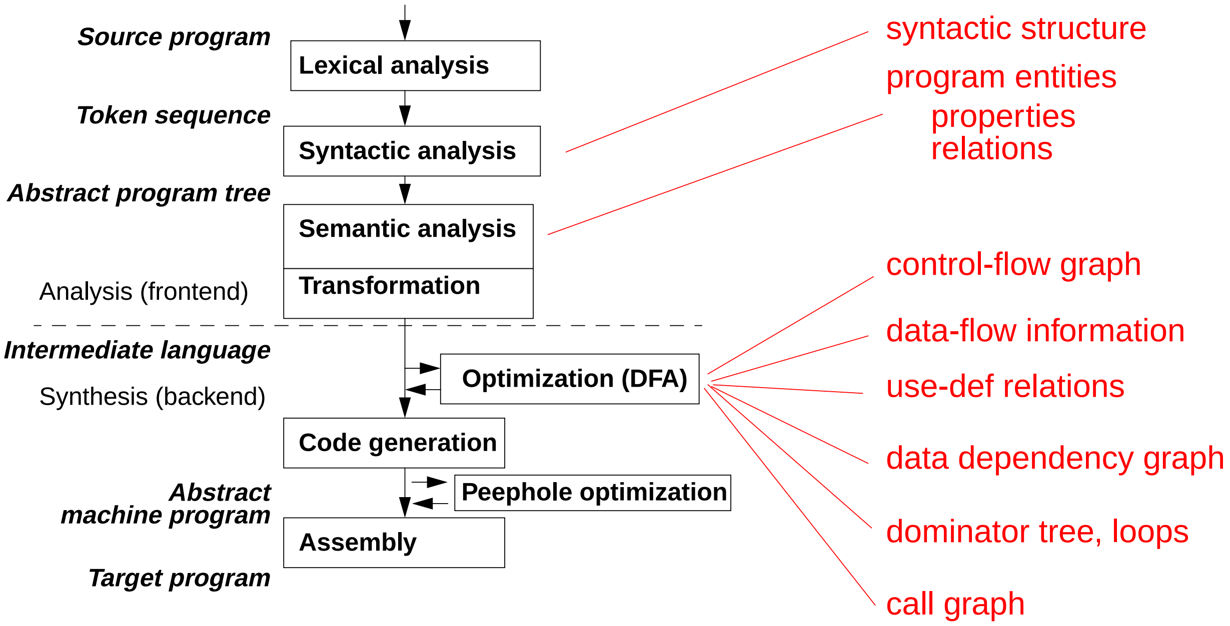
Objectives:

Program analysis beyond optimization

In the lecture:

Examples are given for the objectives

Overview on Program Analysis in Compilers



Source program

Lexical analysis

Token sequence

Syntactic analysis

Abstract program tree

Semantic analysis

Transformation

Analysis (frontend)

Intermediate language

Synthesis (backend)

Optimization (DFA)

Code generation

Abstract machine program

Peephole optimization

Assembly

Target program

syntactic structure

program entities
properties
relations

control-flow graph

data-flow information

use-def relations

data dependency graph

dominator tree, loops

call graph

Lecture Compilation Methods SS 2013 / Slide 205

Objectives:

Analysis methods in compiler structure

In the lecture:

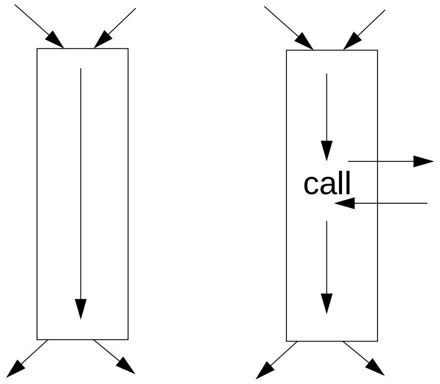
The topics on the slide are explained.

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.

Lecture Compilation Methods SS 2013 / Slide 206

Objectives:

Understand the notion of basic blocks

In the lecture:

The topics on the slide are explained. Examples are given.

- The definition is explained.
- The construction is explained using the example of C-2.7.
- The consequences of having calls in a basic block are discussed.

Questions:

- Explain the decomposition of intermediate code into basic blocks for C-2.7 and for further examples.

Example for Basic Blocks

A C function that computes Fibonacci numbers:

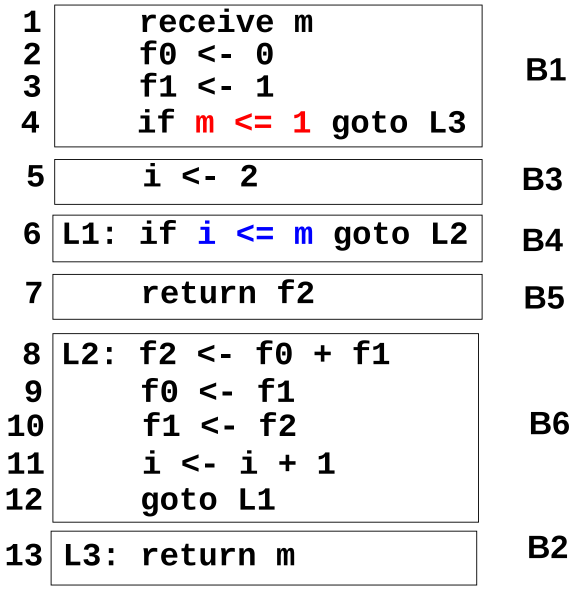
```
int fib (int m)
{  int f0 = 0, f1 = 1, f2, i;
   if (m <= 1)
      return m;
   else
   {  for(i=2; i<=m; i++)
      {  f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
}  }
```

if-condition belongs to the preceding basic block

while-condition does not belong to the preceding basic block

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
```

Lecture Compilation Methods SS 2013 / Slide 207

Objectives:

Example for the construction of basic blocks

In the lecture:

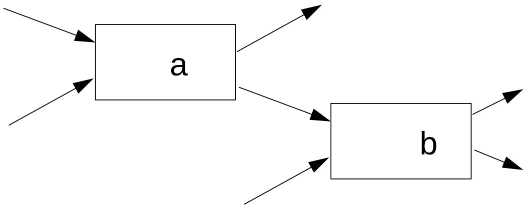
The decomposition into basic blocks is explained according to C-2.6 using the example.

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)

Lecture Compilation Methods SS 2013 / Slide 208

Objectives:

Understand the notion of control-flow graphs

In the lecture:

Examples are given.

- The definition is explained.
- The example of C-2.9 is explained.
- The representation of loops in control-flow graphs is compared to source language representation.
- Algorithms that recognize loops in control-flow graphs are presented in the next section.

Questions:

- Why is the loop structure of source programs not preserved on the level of intermediate languages?

Example for a Control-flow Graph

Intermediate code with basic blocks:

B1

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

B3

```
    i <- 2
```

B4

```
L1: if i <= m goto L2
```

B5

```
    return f2
```

B6

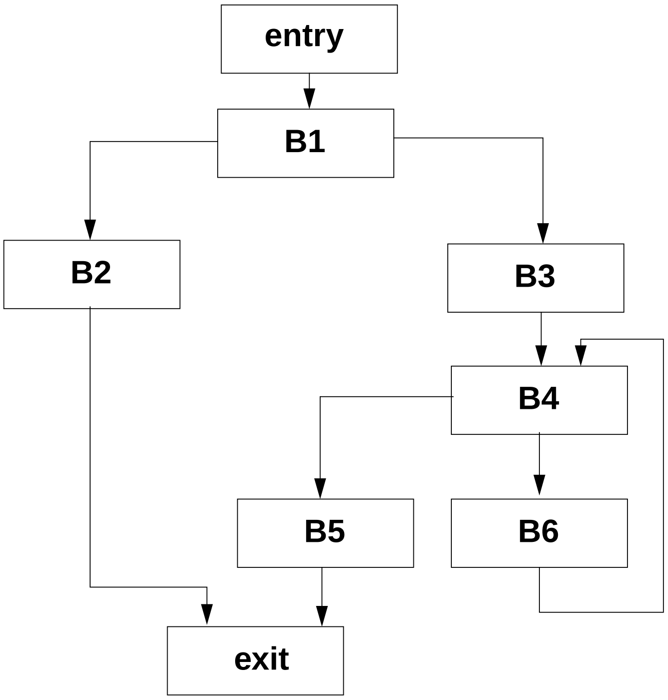
```
L2: f2 <- f0 + f1
    f0 <- f1
    f1 <- f2
    i <- i + 1
    goto L1
```

B2

```
L3: return m
```

Control-flow graph:

[Muchnick, p. 172]



Lecture Compilation Methods SS 2013 / Slide 209

Objectives:

Example for a control-flow graph

In the lecture:

The control-flow graph represents the basic blocks and their branches, as defined in C-2.8.

Questions:

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

---

Lecture Compilation Methods SS 2013 / Slide 210

Objectives:

Overview on control-flow analysis

In the lecture:

The basic ideas of the analysis and transformation techniques are given.

Suggested reading:

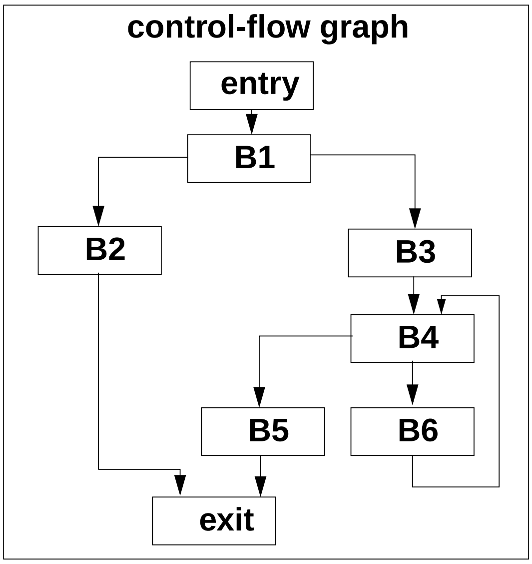
Kastens / Übersetzerbau, Section 8.2.1

Dominator Relation on CFG

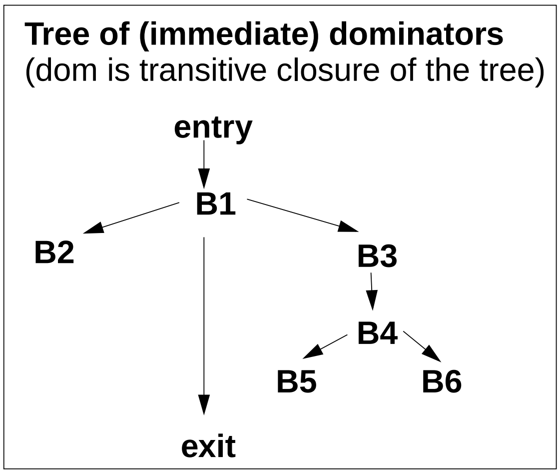
Relation over nodes of a CFG, characterizes paths through CFG,
used for loop recognition, code motion

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)

Lecture Compilation Methods SS 2013 / Slide 211

Objectives:

Understand the dominator relation

In the lecture:

Explain

- the definitions,
- the example.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.2

Questions:

- How is the dominator relation obtained from the immediate dominator relation.
- Why is the dominator relation useful for code motion?

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.

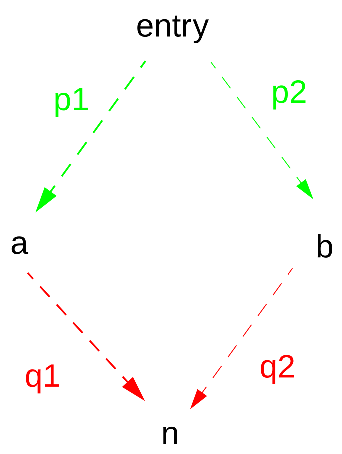
Proof by contradiction:
Assume:
a ≠ b, a dom n, b dom n and
not (a dom b) and not (b dom a)

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



Lecture Compilation Methods SS 2013 / Slide 211a

Objectives:

The set of dominators of a node is ordered

In the lecture:

The proof is explained.

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)

Lecture Compilation Methods SS 2013 / Slide 212

Objectives:

Understand the algorithm

In the lecture:

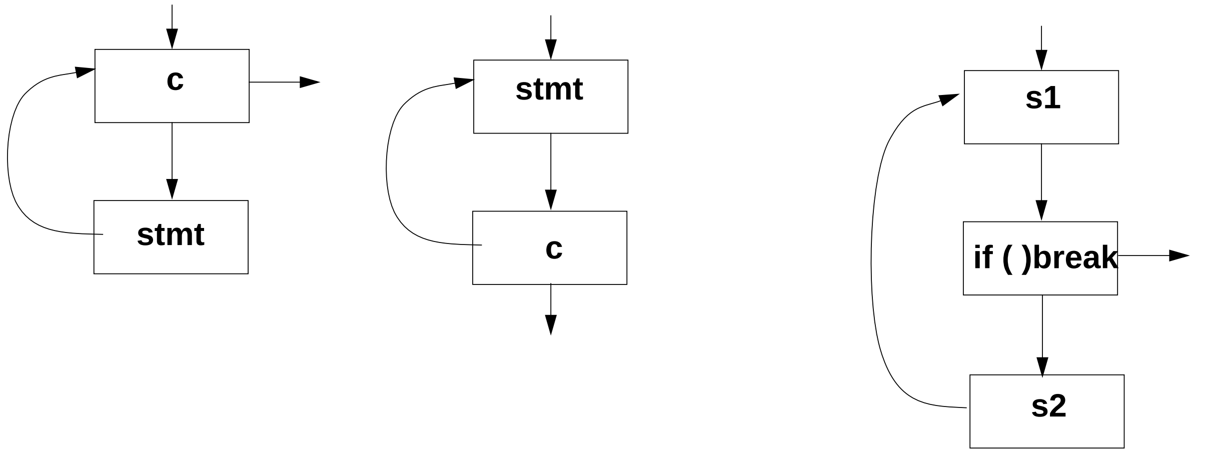
The algorithm is explained using the example of C-2.11

Questions:

What properties and transformations can be characterized using the postdominator relation?

Loop Recognition: Structured Loops

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



Lecture Compilation Methods SS 2013 / Slide 213

Objectives:

Comm on loop structures

In the lecture:

Explain

- the loop structures,
- their occurrences in programming languages,

to get an intuitive understandig of loops;

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.2

Loop Recognition: Natural Loops

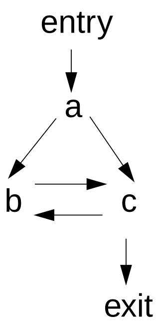
Back edge t->h in a CFG: head h dominates tail t (h dom t).

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:

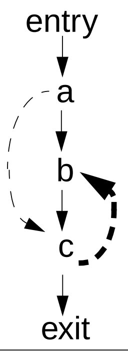
$$S = \{h, t\} \cup \{ p \mid \exists a\ (a \in S \setminus \{h\} \wedge p \in pred(a)) \}$$

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

Lecture Compilation Methods SS 2013 / Slide 213a

Objectives:

Notion of natural loops

In the lecture:

- Explain the definitions;
- give an intuitive understandig of loops;
- show patterns for while and repeat loops, and for loop exit;
- discuss the example of C-2.14.

Suggested reading:

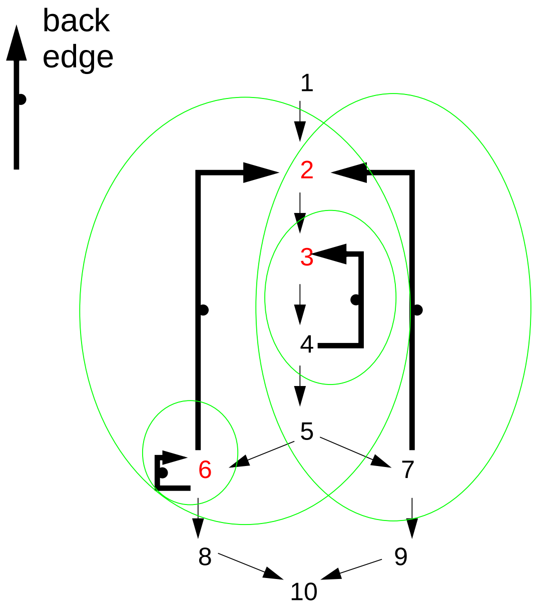
Kastens / Übersetzerbau, Section 8.2.2

Questions:

- What is the role of the loop header?
- Why can’t the graph on the left been derived from structured loops?

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

Lecture Compilation Methods SS 2013 / Slide 214

Objectives:

Recognize natural loops

In the lecture:

- Apply the definitions of C-2.13a to this example;
- discuss nesting of loops.

Suggested reading:

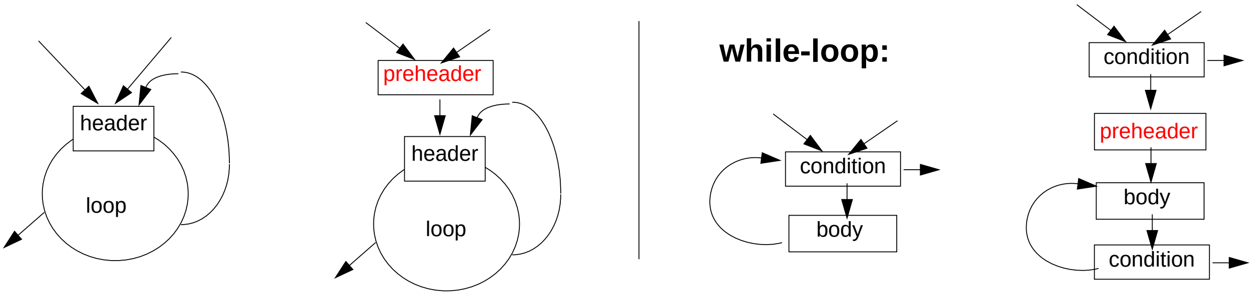
Kastens / Übersetzerbau, Section 8.2.2

Questions:

- Can you give a program structure with repeat-loops, loop-exits, and if-statements for this graph, such that loop S2 is nested in S3?

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:



- 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

---

Lecture Compilation Methods SS 2013 / Slide 215

Objectives:

Get an idea of loop otimization

In the lecture:

- while-loops have to be transformed into repeat-loops, before adding a preheader.
- A use-def-chain links an ocurrence of a variable where it is read (used) to all occurrences where it is written (defined) such that the value may propagate to this point of use. us-def-chains are a result of data flow analysis.
- Explain the optimization techniques.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.3

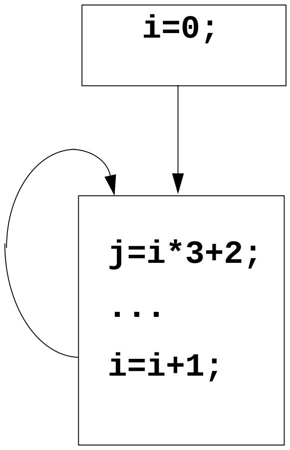
Loop Induction Variables

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).



```
i=0;
```

```
j=i*3+2;
...
i=i+1;
```

Lecture Compilation Methods SS 2013 / Slide 216

Objectives:

Understand the notion of induction variables

In the lecture:

Explain how

- induction variables depend on each other

Suggested reading:

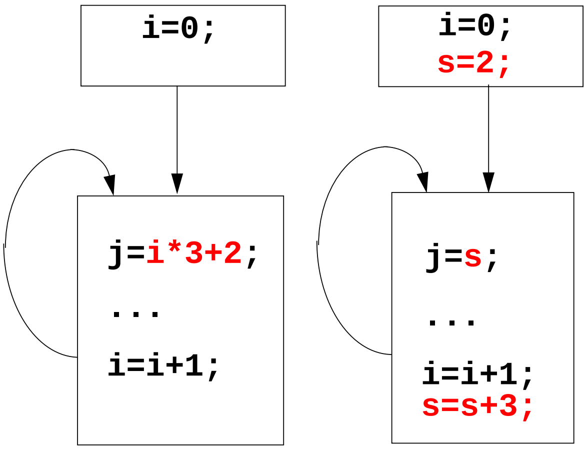
Kastens / Übersetzerbau, Section 8.3.4

Transformation of Induction Variables

Transformation of dependent induction variables:

1. For each (i, a, b) create a temporary variable s.
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

`j: (i, 3, 2)`



```
i=0;
```

```
j=i*3+2;
...
i=i+1;
```

```
i=0;
s=2;
```

```
j=s;
...
i=i+1;
s=s+3;
```

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

Linear increment of array address computation (next slide)

Lecture Compilation Methods SS 2013 / Slide 217

Objectives:

Understand the notion of induction variables

In the lecture:

Explain how

- induction variables are transformed.

Suggested reading:

Kastens / Übersetzerbau, Section 8.3.4

Questions:

- How is the technique applied to array indexing?

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 (E_k)
```

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 (E_k)
```

Lecture Compilation Methods SS 2013 / Slide 217a

Objectives:

Apply the transformation pattern

In the lecture:

The examples are explained:

- expressions linear in induction variables can be transformed, e. g. function arguments;
- multiplications in array addresses are replaced by incrementation.