

Languages and Compilers (SProg og Oversættere)

Lecture 20

Compiler Optimizations

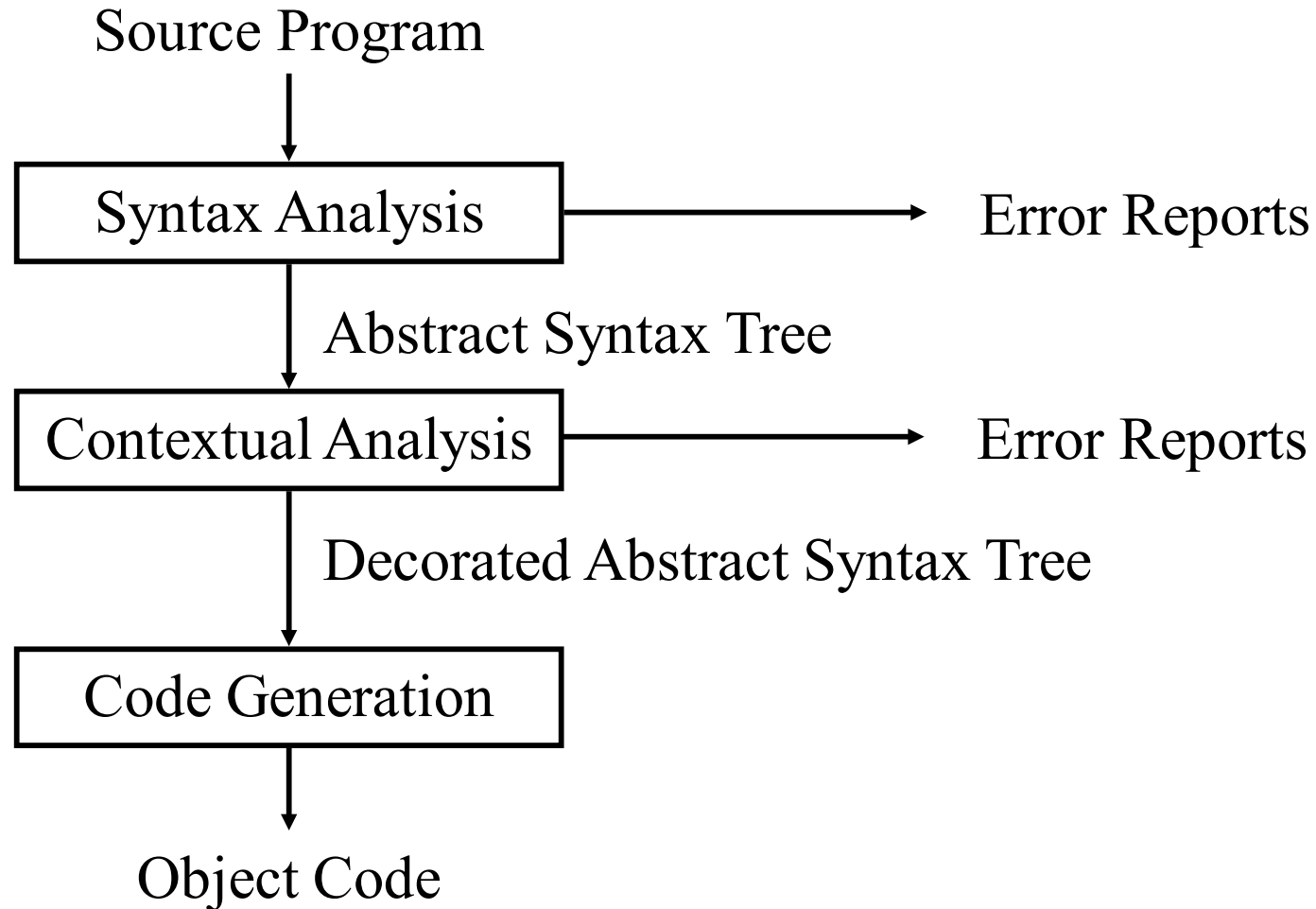
Bent Thomsen

Department of Computer Science

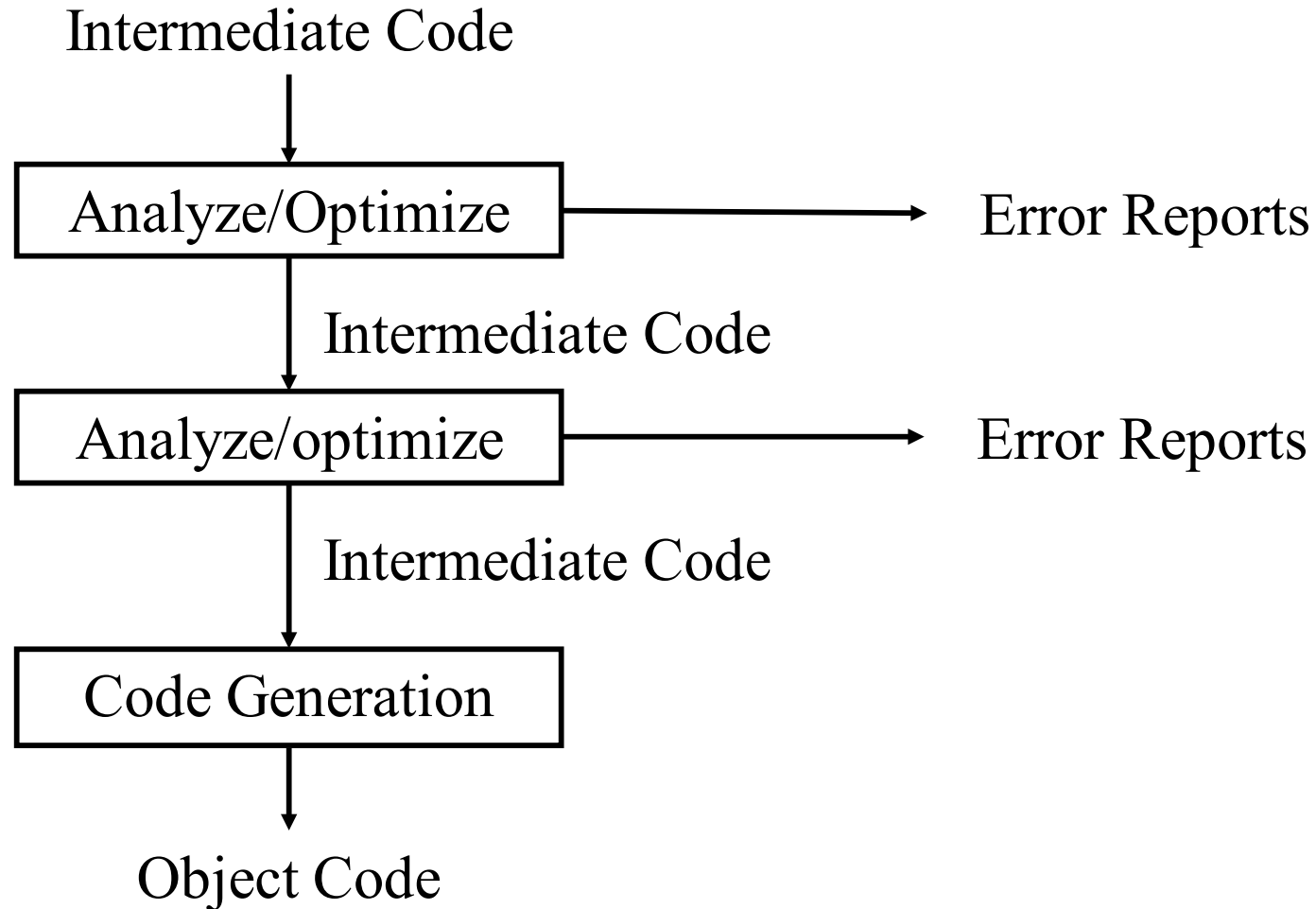
Aalborg University

With acknowledgement to Norm Hutchinson and Mooly Sagiv whose slides this lecture is based on.

The “Phases” of a Compiler



The “Phases” of a Compiler



Compiler Optimizations

The code generated by the code generators discussed so far are not very efficient:

- They compute some values at runtime that could be known at compile time
- They compute values more times than necessary
- They produce code that will never be executed

We can do better! We can do code transformations

- Code transformations are performed for a variety of reasons among which are:
 - To reduce the size of the code
 - To reduce the running time of the program
 - To take advantage of machine idioms
- Code optimizations include:
 - Peephole optimizations
 - Constant folding
 - Common sub-expression elimination
 - Code motion
 - Dead code elimination
- Mathematically, the generation of optimal code is undecidable.

Criteria for code-improving transformations

- Preserve meaning of programs (safety)
 - Potentially unsafe transformations
 - Associative reorder of operands
 - Movement of expressions and code sequences
 - Loop unrolling
- Must be worth the effort (profitability) and
 - on average, speed up programs
- **90/10 Rule:** Programs spend 90% of their execution time in 10% of the code. Identify and improve "hot spots" rather than trying to improve everything.

Peephole optimizations

- Recognition of program patterns that could be rewritten to produce faster code
- Can be done at several levels in the compiler:
 - AST rewrite
 - IR level rewrite
 - Bytecode
 - Target Code
- The general idea:
 - Pattern \Rightarrow replacement

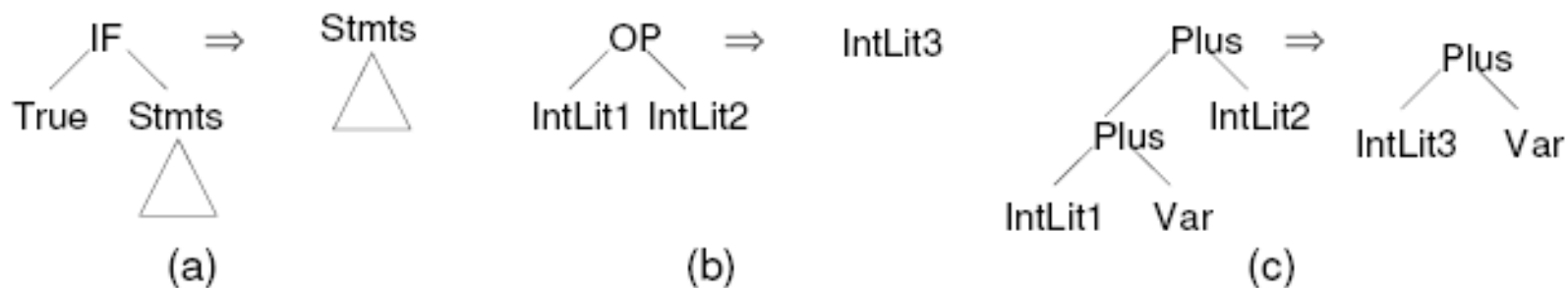


Figure 13.31: AST-Level Peephole Optimization

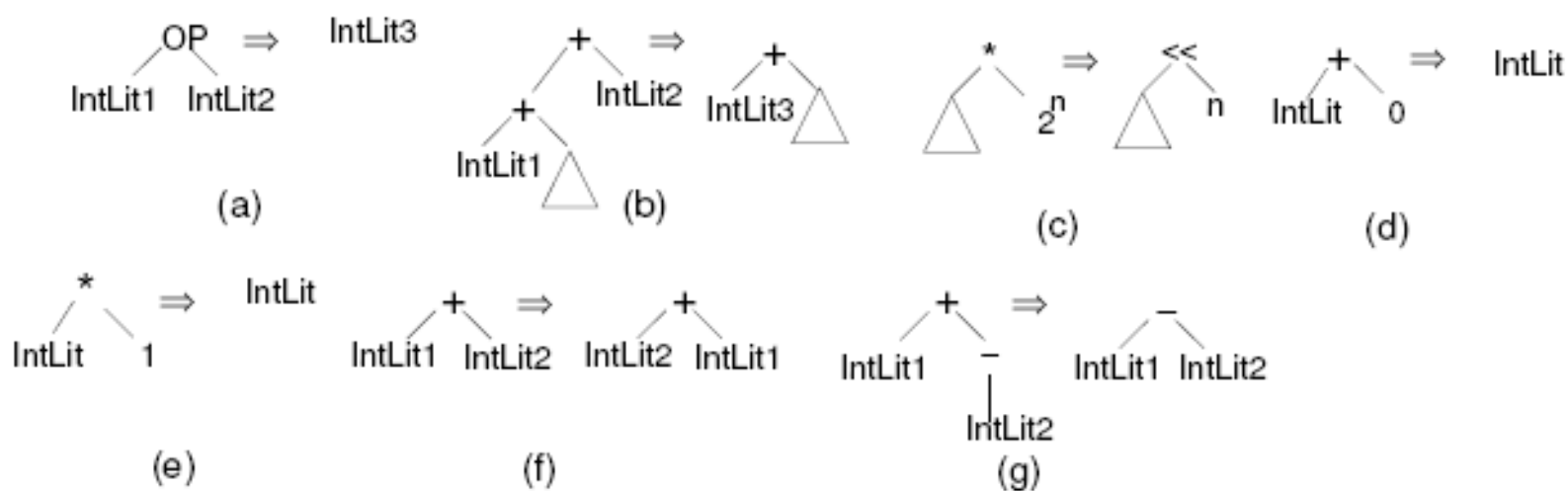


Figure 13.32: IR-Level Peephole Optimizations

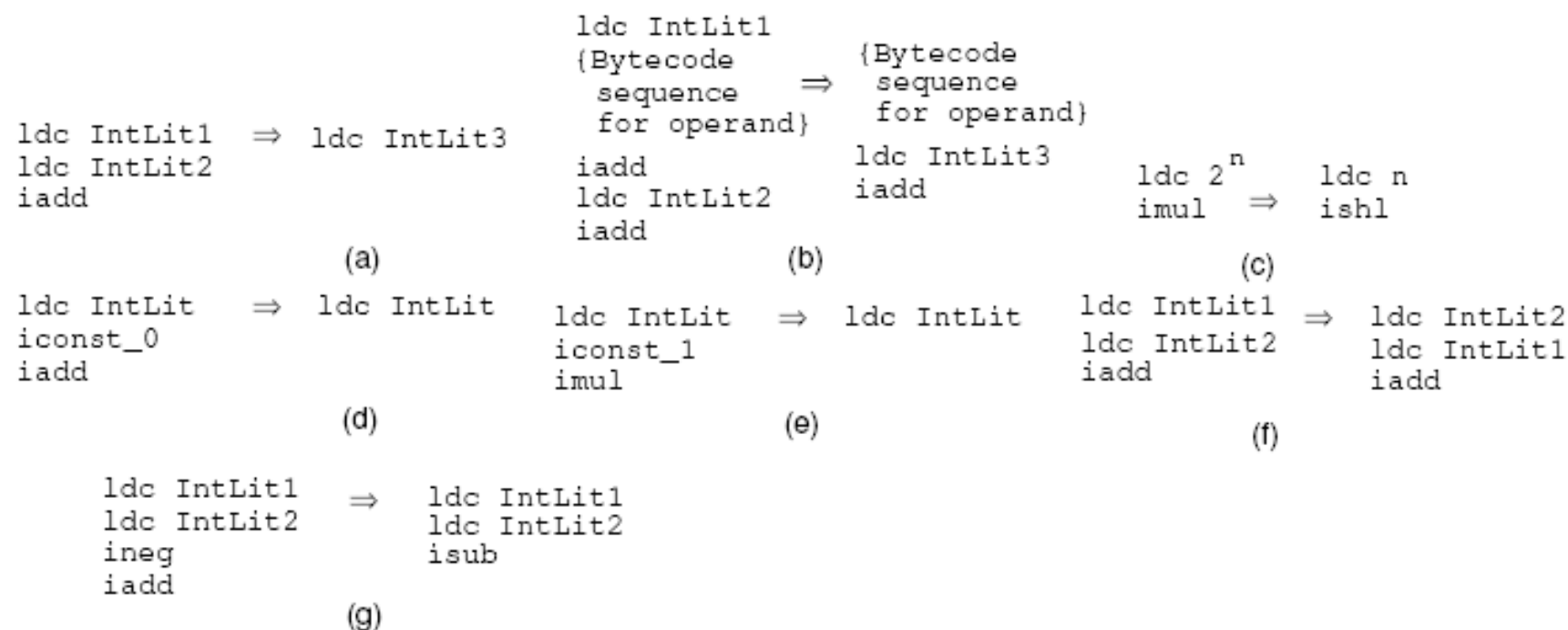


Figure 13.33: Bytecode-Level Peephole Optimizations

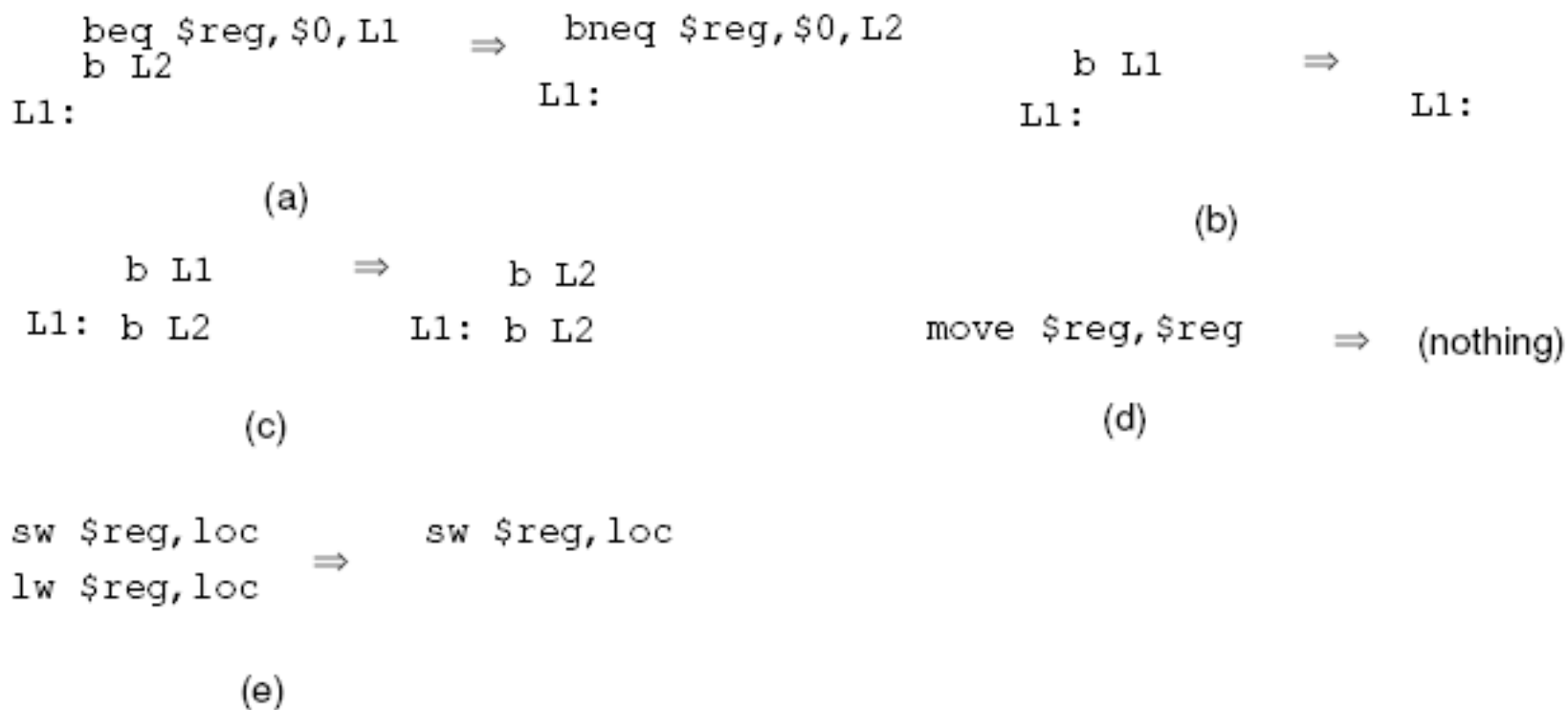


Figure 13.34: Code-Level Peephole Optimizations

Constant folding

- Consider:

```
static double pi = 3.1416;  
double volume = 4/3 * pi * r * r * r;
```

- The compiler could compute $4 / 3 * \pi$ as 4.1888 before the program runs. This saves how many instructions?
- What is wrong with the programmer writing
 $4.1888 * r * r * r$?

Constant folding II

- Consider:

```
struct { int y, m, d; } holidays[6];  
holidays[2].m = 12;  
holidays[2].d = 25;
```

- If the address of `holidays` is `x`, what is the address of `holidays[2].m`?
- Could the programmer evaluate this at compile time?
Safely?

Common sub-expression elimination

- Consider:

```
int t = (x - y) * (x - y + z);
```

- Computing $x - y$ takes three instructions, could we save some of them?

Common sub-expression elimination II

```
int t = (x - y) * (x - y + z);
```

Naïve code:

```
iload x  
iload y  
isub  
iload x  
iload y  
isub  
iload z  
iadd  
Imult  
istore t
```

Common sub-expression elimination II

Programmer tries to be clever

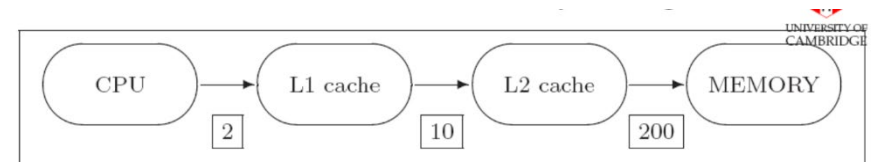
```
int tmp = (x - y)
int t = tmp * (tmp + z);
```

Naïve code: New code:

```
iload x
iload y
isub
iload x
iload y
isub
iload z
iadd
Imult
istore t
```

```
iload x
iload y
isub
istore tmp
iload tmp
iload tmp
iload z
iadd
Imult
istore t
```

Is this code better or worse?



Common sub-expression elimination II

```
int t = (x - y) * (x - y + z);
```

Naïve code:

```
iload x  
iload y  
isub  
iload x  
iload y  
isub  
iload z  
iadd  
Imult  
istore t
```

Better code:

```
iload x  
iload y  
isub  
dup  
iload z  
iadd  
Imult  
istore t
```

Common sub-expression elimination III

- Consider:

```
struct { int y, m, d; } holidays[6];  
holidays[i].m = 12;  
holidays[i].d = 25;
```

- The address of `holidays[i]` is a common subexpression.

Common sub-expression elimination IV

- But, be careful!

```
int t = (x - y++) * (x - y++ + z);
```

- Is `x - y++` still a common sub-expression?

Code motion

- Consider:

```
char name[3][10];  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 10; j++) {  
        name[i][j] = 'a';  
    }  
}
```

- Computing the address of `name[i][j]` is
 $\text{address}[\text{name}] + (i * 10) + j$
- Most of that computation is constant throughout the inner loop

$$\text{address}[\text{name}] + (i * 10)$$

Code motion II

- You can think of this as rewriting the original code:

```
char name[3][10];  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 10; j++) {  
        name[i][j] = 'a';  
    }  
}
```

as

```
char name[3][10];  
for (int i = 0; i < 3; i++) {  
    char *x = &(name[i][0]);  
    for (int j = 0; j < 10; j++) {  
        x[j] = 'a';  
    }  
}
```

Dead code elimination

- Consider:

```
int f(int x, int y, int z)
{
    int t = (x - y) * (x - y + z);
    return 6;
}
```

- Computing t takes many instructions, but the value of t is never used.
- We call the value of t “dead” (or the variable t dead) because it can never affect the final value of the computation. Computing dead values and assigning to dead variables is wasteful.

Dead code elimination II

- But consider:

```
int f(int x, int y, int z)
{
    int t = x * y;
    int r = t * z;
    t = (x - y) * (x - y + z);
    return r;
}
```

- Now t is only dead for part of its existence. Hmm...

Optimization implementation

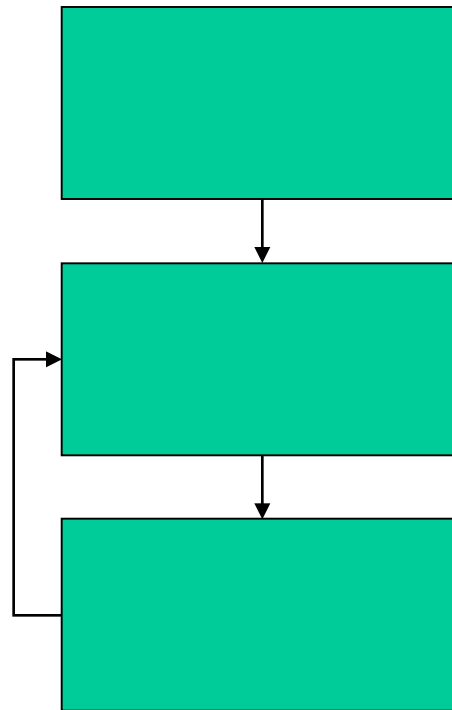
- What do we need to know in order to apply an optimization?
 - Constant folding
 - Common sub-expression elimination
 - Code motion
 - Dead code elimination
- Is the optimization correct or safe?
- Is the optimization an improvement?
- What sort of analyses do we need to perform to get the required information?

Control-Flow Analysis

- The purpose of Control-Flow Analysis is to determine the control structure of a program
 - determine possible control flow paths
 - find basic blocks and loops
- A Basic Block (BB) is a sequence of instructions entered only at the beginning and left only at the end.
- The Control-Flow Graph (CFG) of a program is a directed graph $G=(N, E)$ whose nodes N represent the basic blocks in the program and whose edges E represent transfers of control between basic blocks.

Basic blocks

- A basic block is a sequence of instructions entered only at the beginning and left only at the end.
- A flow graph is a collection of basic blocks connected by edges indicating the flow of control.



Finding basic blocks

iconst_1

istore 2

iconst_2

istore 3

Label_1:

iload 3

iload 1

if_icmplt Label_4

iconst_0

goto Label_5

Label_4:

iconst_1

Label_5:

ifeq Label_2

iload 2

iload 3

imul

dup

istore 2

pop

Label_3:

iload 3

dup

iconst_1

iadd

istore 3

pop

goto Label_1

Label_2:

iload 2

ireturn

Finding basic blocks II

iconst_1
istore 2
iconst_2
istore 3

Label_1:
 iload 3
 iload 1
 if_icmplt Label_4

iconst_0
goto Label_5

Label_4:
 iconst_1

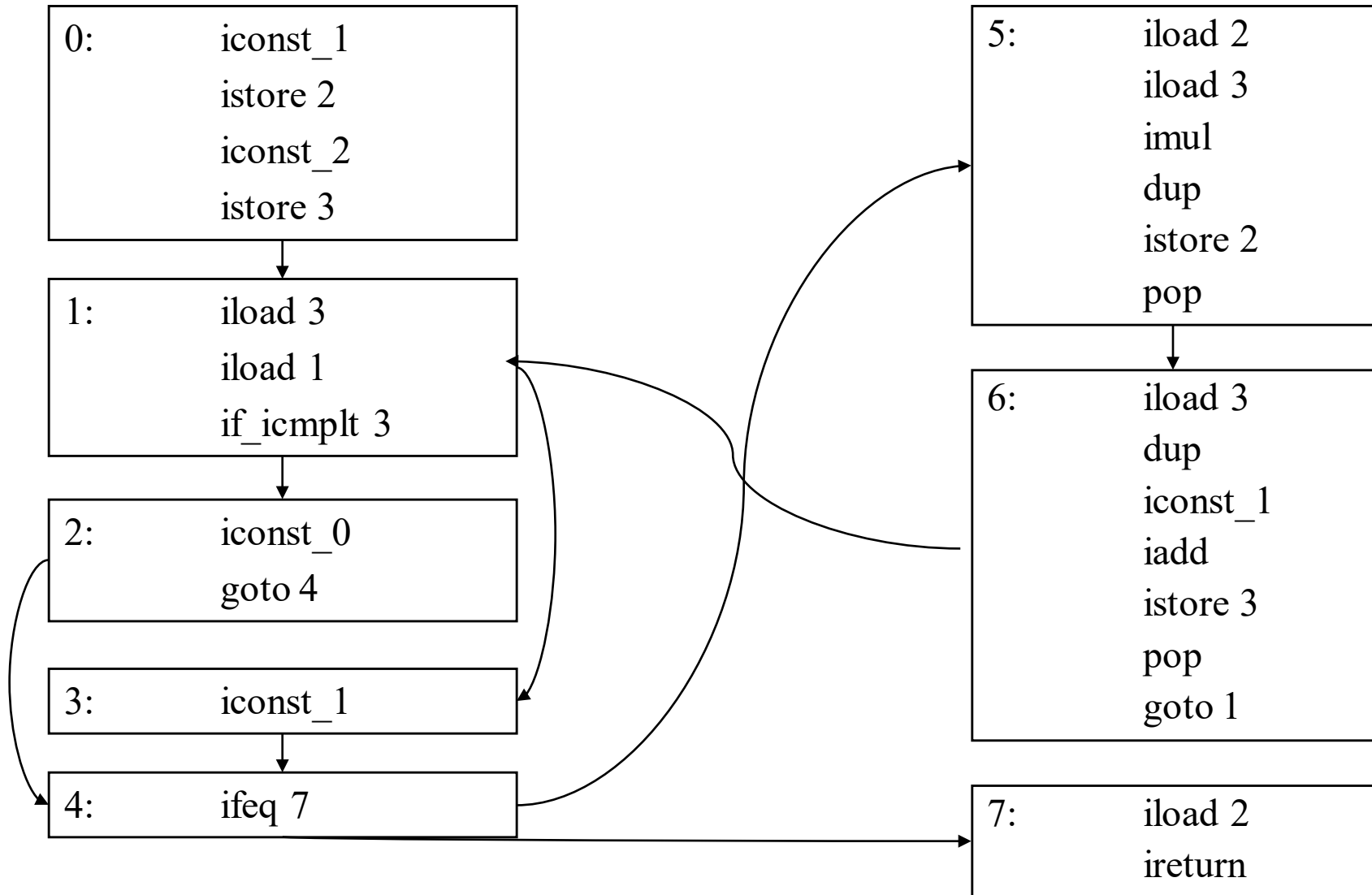
Label_5:
 ifeq Label_2

iload 2
iload 3
imul
dup
istore 2
pop

Label_3:
 iload 3
 dup
 iconst_1
 iadd
 istore 3
 pop
 goto Label_1

Label_2:
 iload 2
 ireturn

Flow graphs



```

procedure FORMBASICBLOCKS( )
    leaders  $\leftarrow$  { first instruction in stream }
    foreach instruction s in the stream do                                (23)
        targets  $\leftarrow$  { distinct targets branched to from s }          (24)
        if  $|targets| > 1$ 
            then
                foreach  $t \in targets$  do  $leaders \leftarrow leaders \cup \{t\}$ 
            foreach  $l \in leaders$  do                                         (25)
                 $block(l) \leftarrow \{l\}$ 
                 $s \leftarrow$  next instruction after  $l$ 
                while  $s \notin leaders$  and  $s \neq \perp$  do
                     $block(l) \leftarrow Block(l) \cup \{s\}$ 
                     $s \leftarrow$  Next instruction in stream
    end

```

Figure 14.7: Partitioning of instructions into basic blocks. The \perp value in pseudocode means *undefined* and is typically denoted as **null** in most programming languages.

Data-Flow Analysis

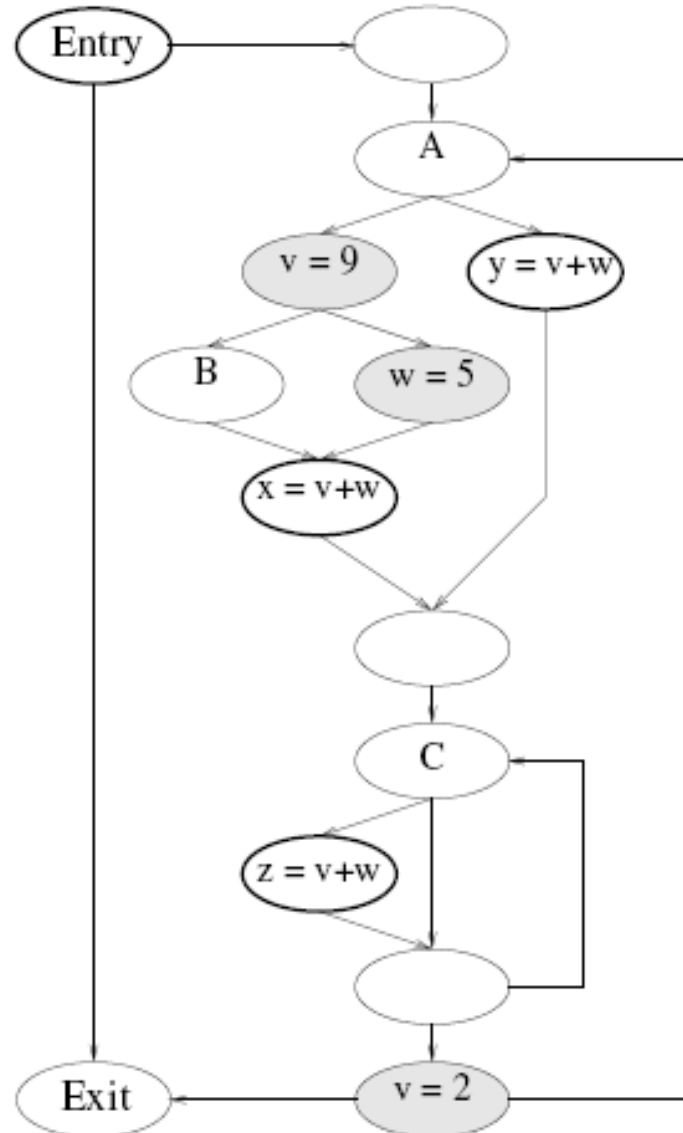
- The purpose of Data-Flow Analysis is to provide global information about how a procedure manipulates its data.
- Examples:
 - Live variable analysis
 - Which variable are still alive?
 - Needed for: register allocation, dead-code elimination
 - Reaching definitions
 - What points in program does each variable definition reach?
 - Needed for: copy- and constant propagation
- Available expressions
 - Which expressions computed earlier still have same value?
 - Needed for: common sub-expression elimination.

```

 $u \leftarrow 5$ 
repeat
  if  $r$ 
  then
     $v \leftarrow 9$ 
    if  $p$ 
    then  $u \leftarrow 6$ 
    else  $w \leftarrow 5$ 
     $x \leftarrow v + w$ 
  else  $y \leftarrow v + w$ 
   $u \leftarrow 7$ 
repeat
  if  $q$ 
  then
     $z \leftarrow v + w$ 
until  $r$ 
 $v \leftarrow 2$ 
until  $s$ 

```

(a)



(b)

Figure 14.41: (a) A program; (b) Its control flow graph.

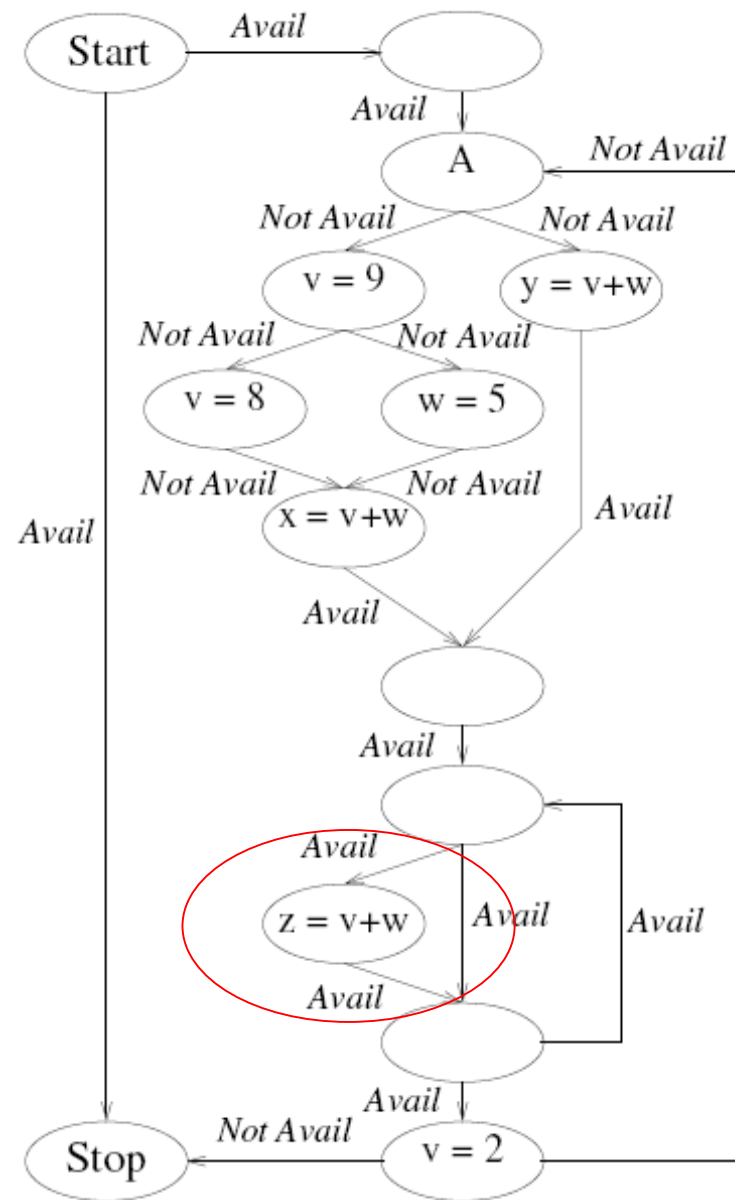


Figure 14.42: Solution throughout the flow graph of Figure 14.41(b) for the availability of expression $v + w$.

Optimizations within a BB

- Everything you need to know is easy to determine
- For example: live variable analysis
 - Start at the end of the block and work backwards
 - Assume everything is live at the end of the BB
 - Copy live/dead info for the instruction
 - If you see an assignment to x, then mark x “dead”
 - If you see a reference to y, then mark y “live”

5:	iload 2	live: 1, 2, 3
	iload 3	live: 1, 3
	imul	live: 1, 3
	dup	live: 1, 3
	istore 2	live: 1, 3
	pop	live: 1, 2, 3
		live: 1, 2, 3

Global optimizations

- Global means “between basic blocks”
- We must know what happens across block boundaries
- For example: live variable analysis
 - The liveness of a value depends on its later uses perhaps in other blocks
 - What values does this block define and use?

5:	iload 2
	iload 3
	imul
	dup
	istore 2
	pop

Define: 2
Use: 2, 3

Global live variable analysis

- We define four sets for each BB
 - $\text{def} ==$ variables with defined values
 - $\text{use} ==$ variables used before they are defined
 - $\text{in} ==$ variables live at the beginning of a BB
 - $\text{out} ==$ variables live at the end of a BB
- These sets are related by the following equations:
 - $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$
 - $\text{out}[B] = \bigcup_S \text{in}[S]$ where S is a successor of B

Solving data flow equations

- Iterative solution:
 - Start with empty set
 - Iteratively apply constraints
 - Stop when we reach a fixed point

For all instructions $\text{in}[I] = \text{out}[I] = \emptyset$

Repeat

For each instruction I

$$\text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I]$$

For each basic block B

$$\text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B']$$

Until no new changes in sets

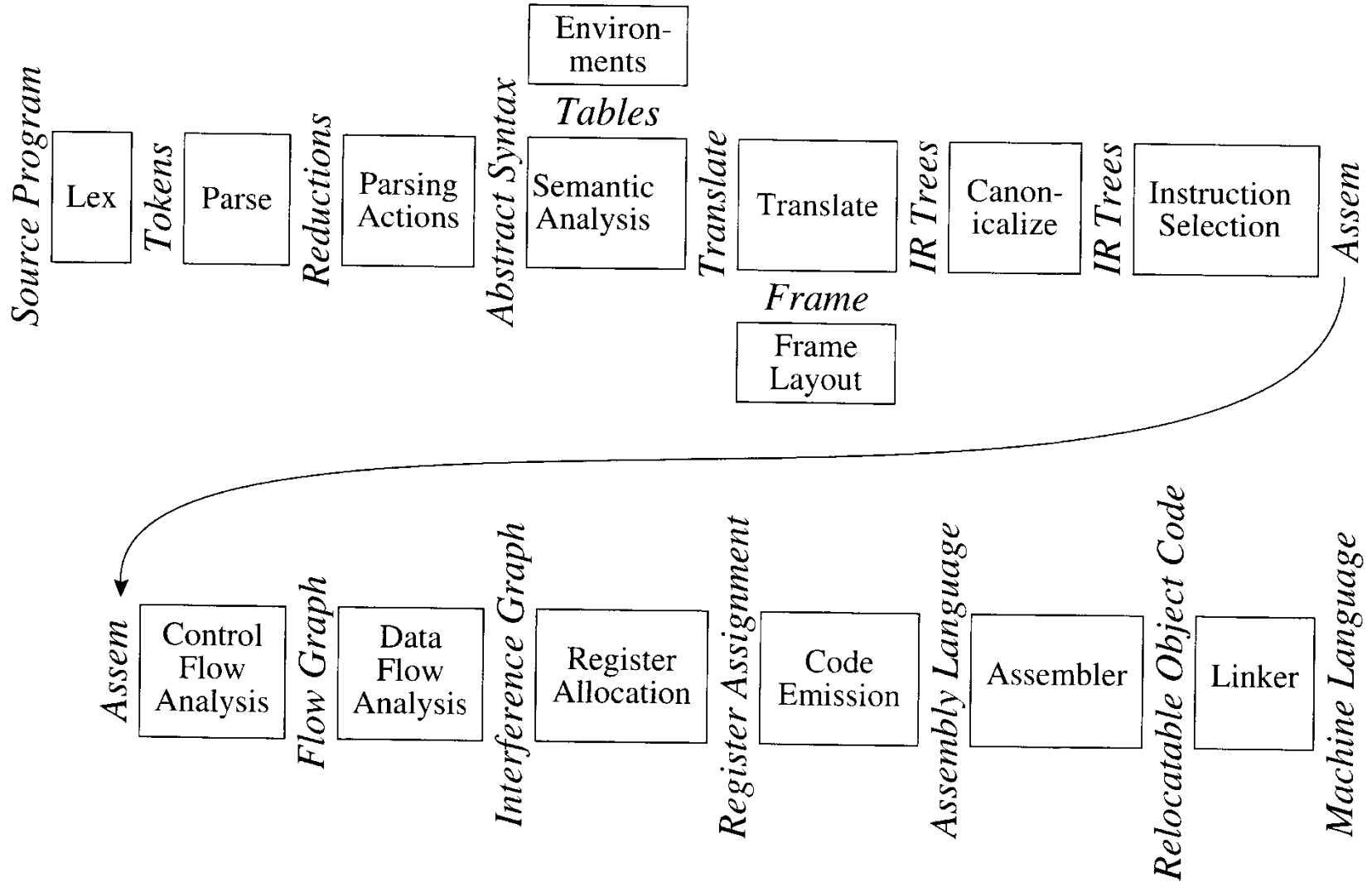
Dead code elimination

- Armed with global live variable information we redo the local live variable analysis with correct liveness information at the end of the block out[B]
- Whenever we see an assignment to a variable that is marked dead, we eliminate it.

Static Analysis

- Automatic derivation of static properties which hold on every execution leading to a program location
- Example Static Analysis Problems
 - Live variables
 - Reaching definitions
 - Expressions that are “available”
 - Dead code
 - Pointer variables that never point into the same location
 - Points in the program in which it is safe to free an object
 - An invocation of a virtual method whose address is unique
 - Statements that can be executed in parallel
 - An access to a variable which must be in cache
 - Integer intervals
 - Security properties
 - WCET and Schedulability
 - ...

A somewhat more complex compiler



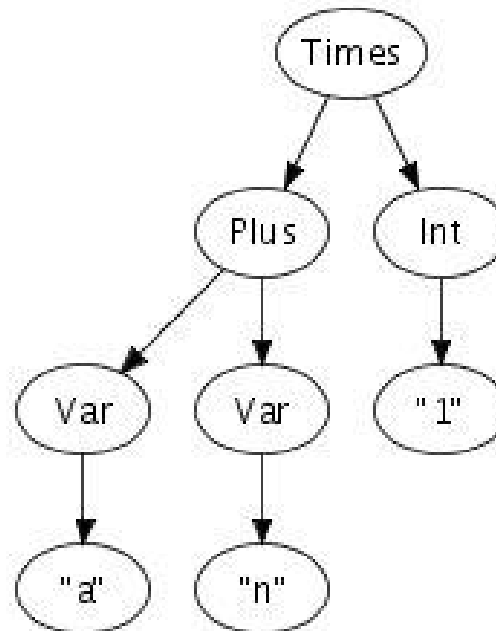
Learning More about Optimizations

- Read chapter 9-12 in the new Dragon Book
 - Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Addison-Wesley, ISBN 0-321-21091-3
- Read the ultimate reference on program analysis
 - Principles of Program Analysis Flemming Nielson, Hanne Riis Nielson, Chris Hankin: Principles of Program Analysis. Springer (Corrected 2nd printing, 452 pages, ISBN 3-540-65410-0), 2005.
- Use one of the frameworks:
 - **Soot: a Java Optimization Framework**
 - <http://www.sable.mcgill.ca/soot>
 - WALA: The T. J. Watson Libraries for Analysis
 - http://wala.sourceforge.net/wiki/index.php/Main_Page

Pause

Remember the exercises before this course?

- 2. Write a Java program that implements a data structure for the following tree
- 3. Extend your Java program to traverse the tree depth-first and print out information in nodes and leaves as it goes along.
- 4. Write a Java program that can read the string "a + n * 1" and produce a collection of objects containing the individual symbols when blank spaces are ignored (or used as separator).



Remember the exercises before this course?

- 2. Make a drawing or description of the phases (internals) of a compiler (without reading the books or searching the Internet) – save this for comparison with your knowledge after the course.
- 4. Create a list of language features group members would like in a new language. Are any of these features in conflict with each other? How would you prioritize the features?
- 5. Discuss what is needed to define a new programming language. Write down your conclusions for comparison with your knowledge after the course.

Organization of a Compiler

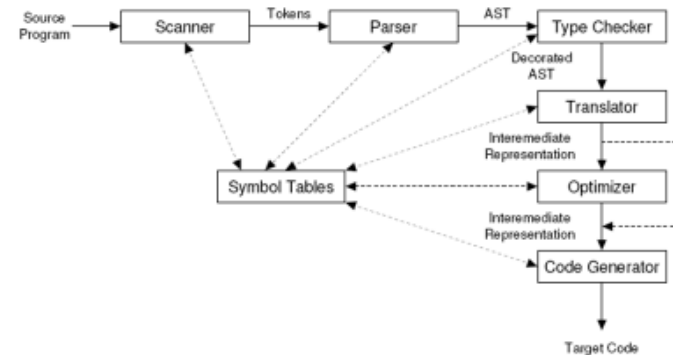


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

What was this course about?

- Programming Language Design
 - Concepts and Paradigms
 - Ideas and philosophy
 - Syntax and Semantics
- Compiler Construction
 - Tools and Techniques
 - Implementations
 - The nuts and bolts

Curricula

Studie ordningen i de gode gamle dage ☺

The purpose of the course is contribute to the student gaining knowledge of important principles in programming languages and understanding of techniques for describing and compiling programming languages.

Sprog og oversættelse / Language and Compiler Construction (SPO)

Omfang: 5 ECTS-point.

Forudsætninger: Programmeringserfaring svarende til projektenheden på 3. semester samt kendskab til imperativ og objektorienteret programmering svarende til 1. - og 2. semesters kurser i programmering.

Mål:

Viden:

Den studerende skal opnå viden om væsentlige principper i programmeringssprog, samt forståelse af teknikker til beskrivelse og oversættelse af sprog generelt, herunder:

- Abstraktionsprincippet, kontrol- og datastrukturer, blokstruktur og scopebegrebet, parametermekanismer og typeækvivalens
- Oversættelse, herunder leksikalsk, syntaktisk, og statisk semantisk analyse, samt kodegenerering
- Køretids-omgivelser, herunder lagerallokering samt strukturer til understøttelse af procedurer og funktioner

Færdigheder:

Den studerende skal opnå følgende færdigheder:

- Kunne redegøre for de berørte teknikker og begreber inden for sprogdesign og oversætterkonstruktion ved brug af fagets terminologi og notation for beskrivelse og implementation af programmeringssprog
- Kunne redegøre for hvordan implementations teknikker influerer sprog design
- Kunne ræsonnere datalogisk om og med de berørte begreber og teknikker

Kompetencer: Den studerende skal kunne beskrive, analysere og implementere programmeringssprog og skal kunne redegøre for de enkelte faser og sammenhængen mellem faserne i en oversætter

Undervisningsform: Kursus

Prøveform: Mundtlig eller skriftlig prøve

Bedømmelse: Ekstern bedømmelse efter 7-trins-skala

Vurderingskriterier: Se Rammestudieordningen.

What is expected of you at the end?

- One goal for this course is for you to be able to explain concepts, techniques, tools and theories to others
 - Your future colleagues, customers and boss
 - (especially me and the examiner at the exam ;-)
- That implies you have to
 - Understand the concepts and theories
 - Know how to use the tools and techniques
 - Be able to put it all together
- I.e. You have to know and know that you know

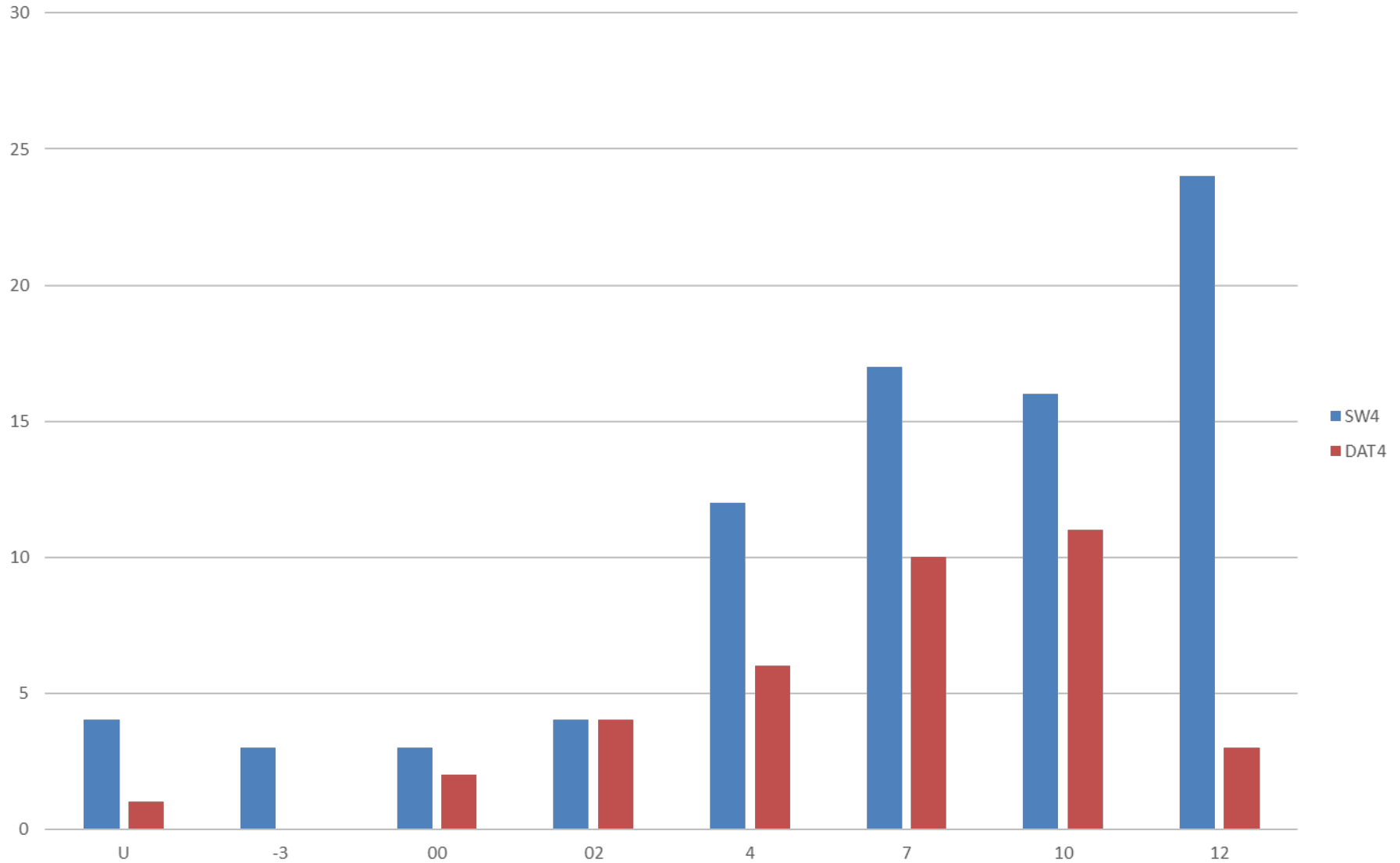
Exam

- 15 minute video presentation exam
 - To be recorded in 1 hours
 - Your subject and questions will be released in DE
- Subjects are already published
 - So you know roughly what we will ask you !!
 - For each published question there will be some questions you do not know before hand.
 - For each question there will be a set of slides available that you can choose to use for your presentation
 - note you do not need to use all the available slides.
 - you may draw on slides, add slides, or choose to only use the slides provided
 - If you modify the provided slides, it is a good idea to state this at the beginning of the presentation.

The 8 Questions

1. Language Design and Control Structures
2. Structure of the compiler
3. Lexical analysis
4. Parsing
5. Semantic Analysis
6. Run-time organization
7. Heap allocation and Garbage Collection
8. Code Generation

And how did it go last year?



Important

- At the end of the course you should ...
- Know
 - Which theories and techniques exist
 - Which tools exist
- Be able to choose “the right ones”
 - Objective criteria
 - Subjective criteria
- Be able to argue and justify your choices!

The Most Important Open Problem in Computing

Increasing Programmer Productivity

- Write programs correctly
 - Write programs quickly
 - Write programs easily
-
- Why?
 - Decreases support cost
 - Decreases development cost
 - Decreases time to market
 - Increases satisfaction

Why Programming Languages?

3 ways of increasing programmer productivity:

1. Process (software engineering)

- Controlling programmers

2. Tools (verification, static analysis, program generation)

- Important, but generally of narrow applicability

3. **Language design** --- the center of the universe!

- Core abstractions, mechanisms, services, guarantees
- Affect how programmers approach a task (C vs. SML)
- Multi-paradigm integration

New Programming Language! Why Should I Care?

- The problem is not designing a new language
 - It's easy! Thousands of languages have been developed
- The problem is how to get wide adoption of the new language
 - It's hard! Challenges include
 - Competition
 - Usefulness
 - Interoperability
 - Fear

“It's a good idea, but it's a new idea; therefore, I fear it and must reject it.”
--- Homer Simpson

- The financial rewards are low, but ...

Famous Danish Computer Scientists

- Peter Nauer
 - BNF and Algol
- Per Brinck Hansen
 - Monitors and Concurrent Pascal
- Dines Bjørner
 - VDM and ADA
- Bjarne Strastrup
 - C++
- Mads Tofte
 - SML
- Rasmus Lerdorf
 - PhP
- Anders Hejlsberg
 - Turbo Pascal and C#
- Lars Bak
 - Java HotSpot VM, V8 and DART
- Jacob Nielsen

Søg:

Danmark ☒

Verden ☐

Firma ☐

Søg



Fredag 24. februar

Internetavisen

Arkiv

Morgenavisen

E-avisen

søg på jp.dk

OK

Forside

Indland

Velfærd

Udland

Børs & Finans

Erhverv

Privatøkonomi

Karriere

IT & Computer

> artikel

Sport

News

Rejser & Ferie

Biler

Meninger

Århus

København

Kultur

TV-guide

Multimedia

Offentliggjort 24. februar 2006 14:01 - opdateret 14:06

Tip en ven

Print-version

Fornem IT-pris til dansker

Som den første dansker nogensinde tildeles Peter Naur, professor emeritus ved Københavns Universitet, ACM's Turing Award - også kaldet datalogiens svar på en Nobelpris.

Prisen er datalogiens højeste udmærkelse og uddeles en gang om året til personer, som har ydet et afgørende og varigt bidrag til området. Den ledsages af et beløb på 100.000 dollars, svarende til ca. 620.000 kroner, som er skænket af virksomheden Intel. Prisen vil blive overrakt ved en banket den 20. maj i San Francisco.

Ifølge ACM's priskomite har Peter Naur fået prisen for "grundlæggende bidrag til udformningen af programmeringssprog og definitionen af Algol 60, til udformningen af oversættere og til det kreative og praktiske arbejde med programmering".

/ritzau/

Tilbage til forsiden

Til toppen af siden

Tip en ven

Print-version

Download musik



Box.dk top 5

Trine Dyrholm
Avenuen

Sidsel Ben Semmane
Twist of Love

Tina Dickow
Nobody's man

Katie Melua
Nine Million Bicycles

Jakob Sveistrup
Book Of Love

Søg alt

Alle formater

SØG MUSIK

- » Log ind | » Ny bruger
- » Internet & styresystemer
- » Forretningssoftware
- » Job & karriere
- » It-styring & outsourcing
- » Server/storage & netværk
- » Sikkerhed
- » Softwareudvikling
- » It-arkitektur
- » Mobil it & tele

Dagligt nyhedsbrev

indtast e-mail...

Nyhedsfeeds (RSS)

ENERGINET.DK

Bliv
graduate

Tre danskere blandt verdens 150 it-helte

Tre danske it-pionerer har fundet vej til listen over alle tiders største it-helte.

Af Torben R. Simonsen, 13. februar 2007 kl. 09:49

It-mediesyndikatet Sys-con har spurgt i sit internationale net af it-medier og ikke mindst deres læsere, hvilke it-personligheder der har haft størst betydning.

Listen med de 150 mest betydningsfulde it-helte er nu færdig og tre danskere har fundet vej til listen.

Sys-con har ikke rangeret heltene indbyrdes, men giver blot en alfabetisk liste.

De tre danskere på listen er:

Anders Hejlsberg, der i dag er ansat i Microsoft, men som tidligere har stået bag udviklingen af Turbo Pascal og programmeringssproget C#.

Rasmus Lerdorf er med på listen for sit arbejde med udvikling af scriptsproget PHP


Bjarne Stroustrup er med for udarbejdelse af det oprindelige design til og implementering af C++.

Hverken Janus Friis eller Niklas Zennström, der blandt andet står bag Kazaa og Skype, har fundet vej til listen.

Mindre overraskende er det, at man kan finde personer som Microsoft-stifter Bill Gates, Intel-stifter Gordon Moore (Moore's Lov) og skaberen af det moderne internet Tim Berners-Lee.

Relaterede links

MØD HP BLADESYSTEM c-Class





Virtual Connect Arkitektur

Du kan forberede kablingen af hele din serverinstallation en gang for alle og derefter tilføje eller erstatte servere uden driftsstop

HP BladeSystem anvender Dual-Core Intel® Xeon® processorer og Intel® Itanium® 2 Processorer

» Download IDC white paper og bliv en del af vores netværk

SENESTE NYT SENEESTE DEBAT

10:39 » Version2 arrangerer format battle

10:20 » Mangel på folk sender løn-opgaver til Multidata

10:13 » Stormløb mod servere forsinket selvangivelser

09:51 » Danskerne er vilde med Microsoft

09:07 » Bankernes EDB Central outsourcer pc-drift

08:38 » Belgisk ODF-pioner på Christiansborg

08:04 » Derfor mangler ÆØÅ på din bon

07:41 » Microsoft tester rettelse til Windows-sårbarhed

15:45 » Softwarefejl lukker bank-it landet over

14:37 » Google på vej med PowerPoint-dræber

// Flere nyheder // Tilmeld nyhedsbrev

SENESTE BLOG-INDLÆG



» **Poul-Henning Kamp:** Utroligt hvad 7Watt kan | 7

» **Peter Toft:** Firefox hitter i Europa - men ikke i stokkonservative Danmark :(| 20

» **Peter Toft:** Bestil mælk på Jensens Býfhus | 16

» **Poul-Henning Kamp:** IPSEC sutter | 13

» **Peter Toft:** Hvilke nyheds-sites bør Linux folk følge? | 21

EMNER [C#](#)[Se kommentarer \(11\)](#)SEKTION [Udvikling](#)

Udvikler Anders Hejlsberg vinder dansk it-hæder

Manden bag både Turbo Pascal, Delphi og C# bliver årets vinder af IT-Prisen 2014, der uddeles af foreningen IT-Branchen.

Af Jesper Kildebogaard Fredag, 14. marts 2014 - 9:45

Mens nogle kalder sig serie-iværksættere, må Anders Hejlsberg kunne kalde sig serie-udvikler af programmeringssprog. Gennem sin lange karriere har han nemlig stået bag det ene toneangivende sprog efter det andet.

Den indsats får han nu et stort skulderklap for af den danske it-branche. På årsmødet for foreningen IT-Branchen blev IT-Prisen 2014 tildelt Anders Hejlsberg, som siden 1996 har arbejdet for Microsoft i USA.

Her har han senest været leder af udviklingen af Typescript, som er Microsofts bud på en forbedring af Javascript - samtidig med at man bevarer fuld kompatibilitet med Javascript. Det kan du læse mere om i Version2's interview med Anders Hejlsberg fra 2012, da Typescript blev lanceret.

Læs også: [Anders Hejlsberg: Sådan styrer jeg C#-udviklingen](#)

Mere berømt er hans arbejde med C# og .Net, der i dag bliver brugt af millioner af udviklere i Microsoft-miljøer. Han arbejdede også på J++ og Microsoft Foundation Classes.

Før jobbet hos Microsoft udviklede han Delphi og Turbo Pascal hos Borland, der var et stort navn tilbage i 1980'erne. Det var et job hos Borland, der i 1987 trak Anders Hejlsberg til USA, hvor han har boet siden.

»Ingen danskere har som Anders Hejlsberg haft indflydelse på den digitale udvikling i verden. Han har gennem tre årtier påvirket udviklingen af de programmeringssprog, som er grundlaget for vores moderne kommunikationssamfund,« udtaler adm. direktør i IT-Branchen Morten Bangsgaard i en pressemeddelelse.

IT-Prisen er 'den største kollegiale hæder', der bliver uddelt i den danske it-branche, skriver foreningen selv. Prisen er de seneste år gået til blandt andet Michael Seifert, direktør for Sitecore, Lars Frelle-Petersen, direktør for Digitaliseringsstyrelsen, og it-iværksætteren Thomas Madsen-Mygdal. It-mediet Computerworld og IT-Branchen står bag prisen.

Fancy joining this crowd?

- Look forward to the PP (Programming Paradigms) course
 - on SW7/DAT7/IT7
- Look forward to the Advanced Programming course
 - On SW8/IT8
- Specialize in Programming Technology
 - on DAT9/DAT10 or SW9/SW10 or IT9/IT10
- Research Programme in Programming Technology
 - Programmatic Program Construction
 - Real-time programming in Java (and C)
 - Big Data and Functional Programming
 - Popular Parallel Programming (P3)
 - Prescriptive Analytics
 - Energy Aware Programming
- "The P-gang":
 - Kurt Nørmark
 - Lone Leth
 - Bent Thomsen
 - Thomas Bøgholm

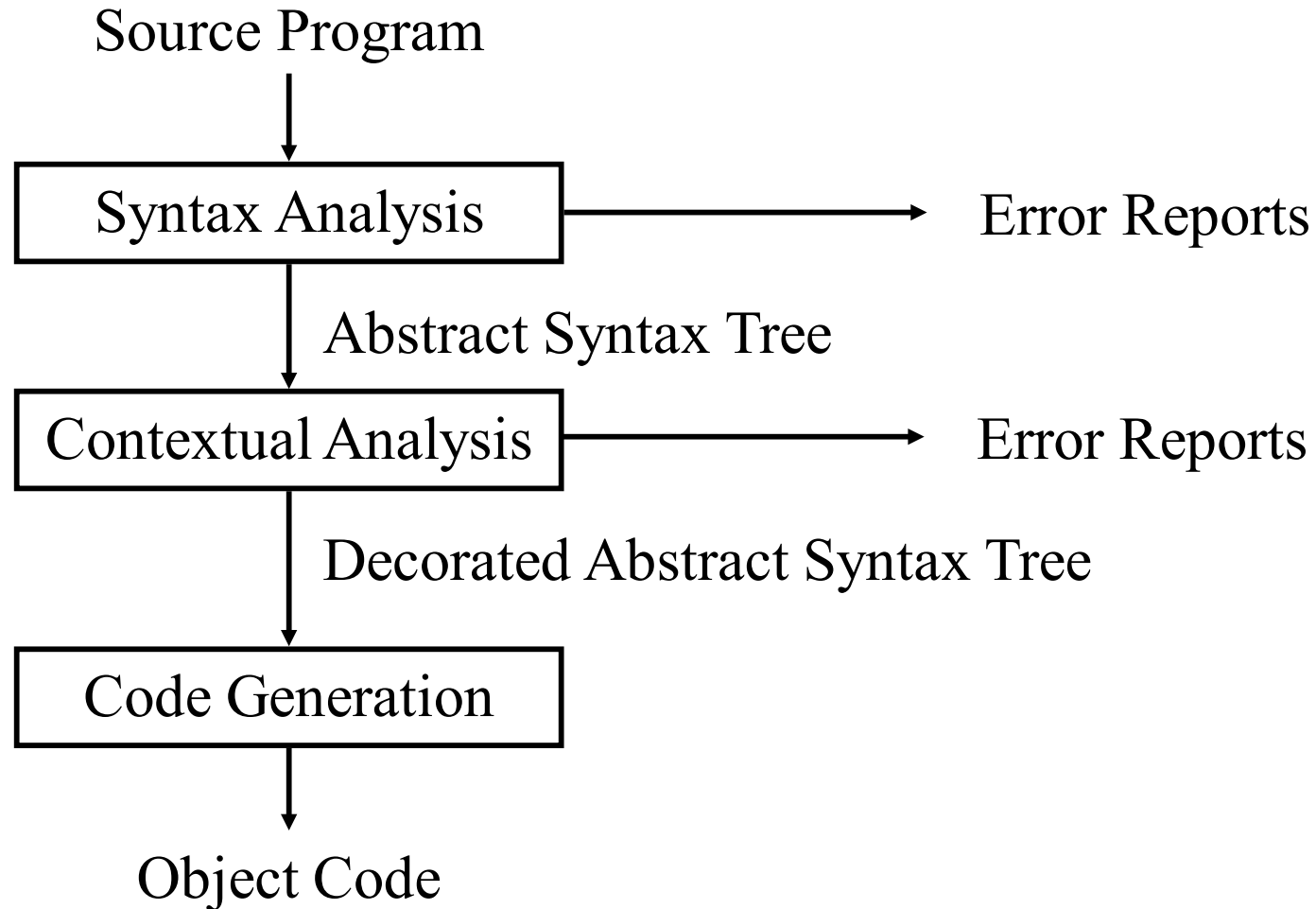
What I promised you at the start of the course

Ideas, principles and techniques to help you

- Design your own programming language or design your own extensions to an existing language
- Tools and techniques to implement a compiler or an interpreter
- Lots of knowledge about programming

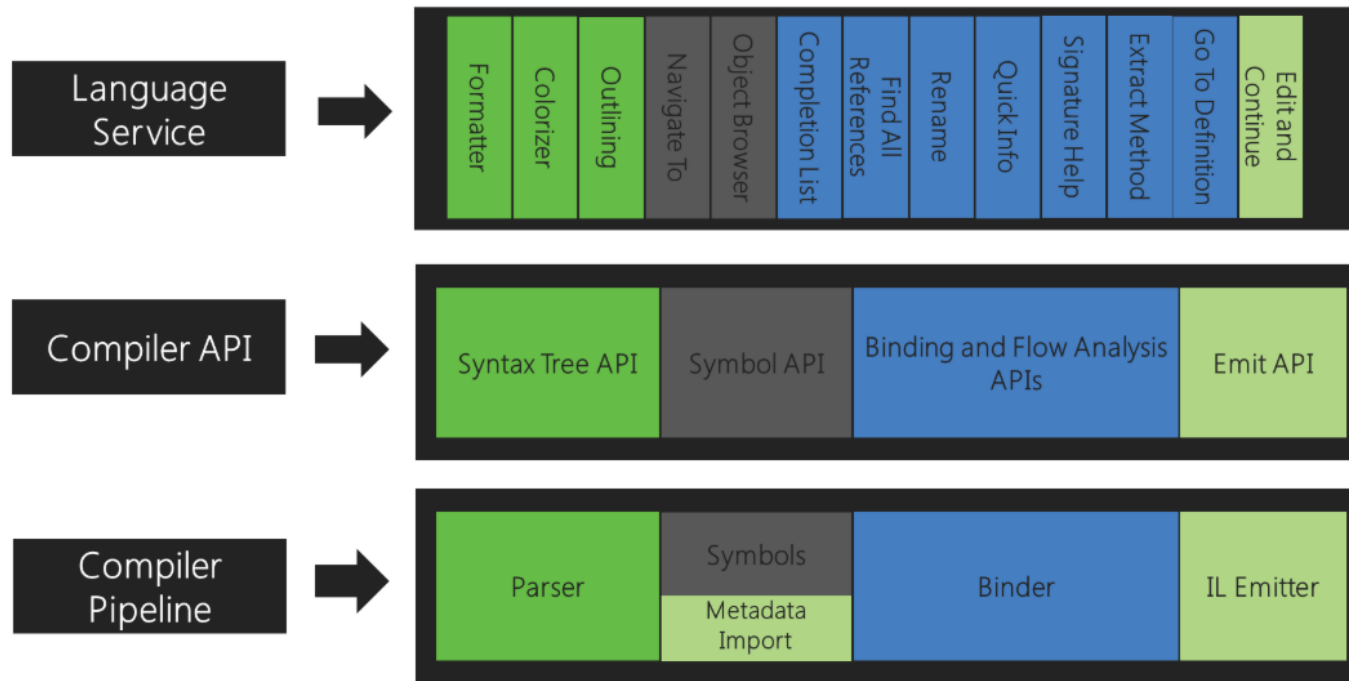
I hope you feel you got what I promised

The “Phases” of a Compiler



Is this picture still valid or is it how compilers were taught 30 years ago?

.NET Compiler Platform ("Roslyn") Overview



Corresponding to each of those phases, an object model is surfaced that allows access to the information at that phase:

The parsing phase is exposed as a syntax tree,
the declaration phase as a hierarchical symbol table,
the binding phase as a model that exposes the result of the compiler's semantic analysis
the emit phase as an API that produces IL byte codes.

Programming Language design

- Designing a new programming language or extending an existing programming language usually follows an iterative approach:
 1. Create ideas for the programming language or extensions
 2. Describe/define the programming language or extensions
 3. Implement the programming language or extensions
 4. Evaluate the programming language or extensions
 5. If not satisfied, goto 1

Discount Method for Evaluating Programming Languages

1. Create tasks specific to the language being tested - tasks that the participants of the experiment should solve.
Estimate the time needed for each task (max 1 hour)
2. Create a short sample sheet of code examples in the language being tested, which the participants can use as a guideline for solving the tasks.
3. Prepare setup (e.g. use of NotePad++ and recorder) and do a sample test with 1 person.
 - Adjust tasks if needed
4. Perform the test on each participant, i.e. make them solve the tasks defined in step 1. (Use approx. 5 test persons)
5. Each participant should be interviewed briefly after the test, where the language and the tasks can be discussed.
6. Analyze the resulting data to produce a list of problems
 - Cosmetic problems, Serious problems, Critical problems

Discount Method for Evaluating Programming Languages

- Method inspired by the Discount Usability Evaluation (DUE) method and Instant Data Analysis (IDA) method
- Reference:
 - Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen.
 - Discount method for programming language evaluation.
 - In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2016). ACM, New York, NY, USA, 1-8.
DOI: <https://doi.org/10.1145/3001878.3001879>

Finally

Keep in mind, the compiler is the program from which all other programs arise. If your compiler is under par, all programs created by the compiler will also be under par. No matter the purpose or use -- your own enlightenment about compilers or commercial applications -- you want to be patient and do a good job with this program; in other words, don't try to throw this together on a weekend.

Asking a computer programmer to tell you how to write a compiler is like saying to Picasso, "Teach me to paint like you."

Sigh Well, Picasso tried.