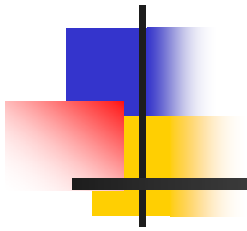
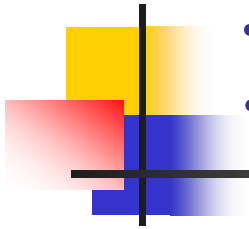


Informatik II
Languages, Compilers, and Theory

Lecture 8:

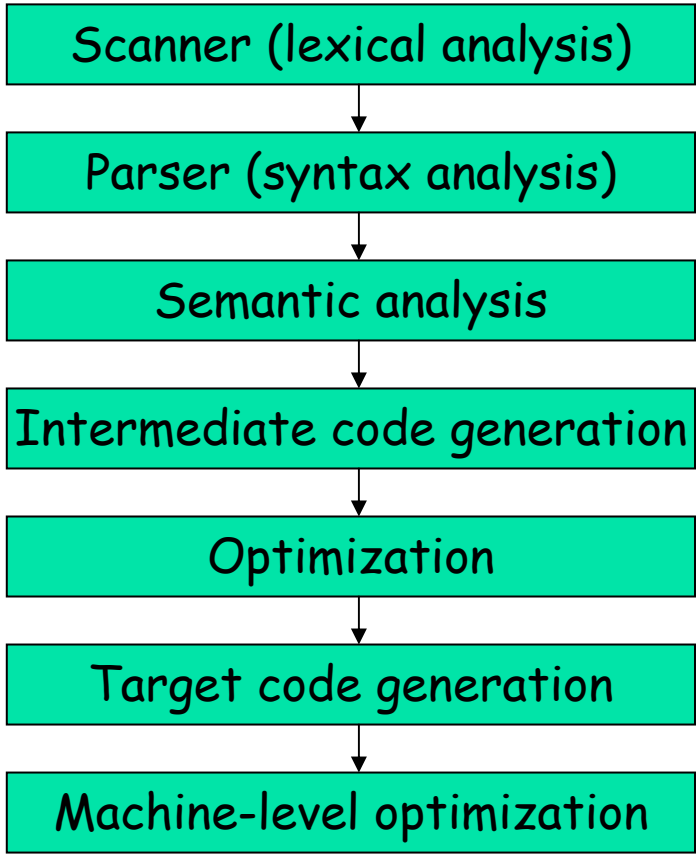
Putting the Pieces Together



Important Announcement

- The final exam room has changed!
- New location:
  - Room HS1 in the Mathematischen Institut
- Date and time are the same:
  - Tuesday July 22, 14:00-16:00
- ***Register in Munopag!***
  - Deadline: Monday July 21, 2003.

The Compilation Process



Read program and convert character stream into tokens.

Theory: Regular expressions, finite automata

Read stream of tokens and generate a parse tree.

Theory: Context free grammars

Traverse parse tree, checking non-syntactic rules.

Traverse parse tree again, emitting intermediate code.

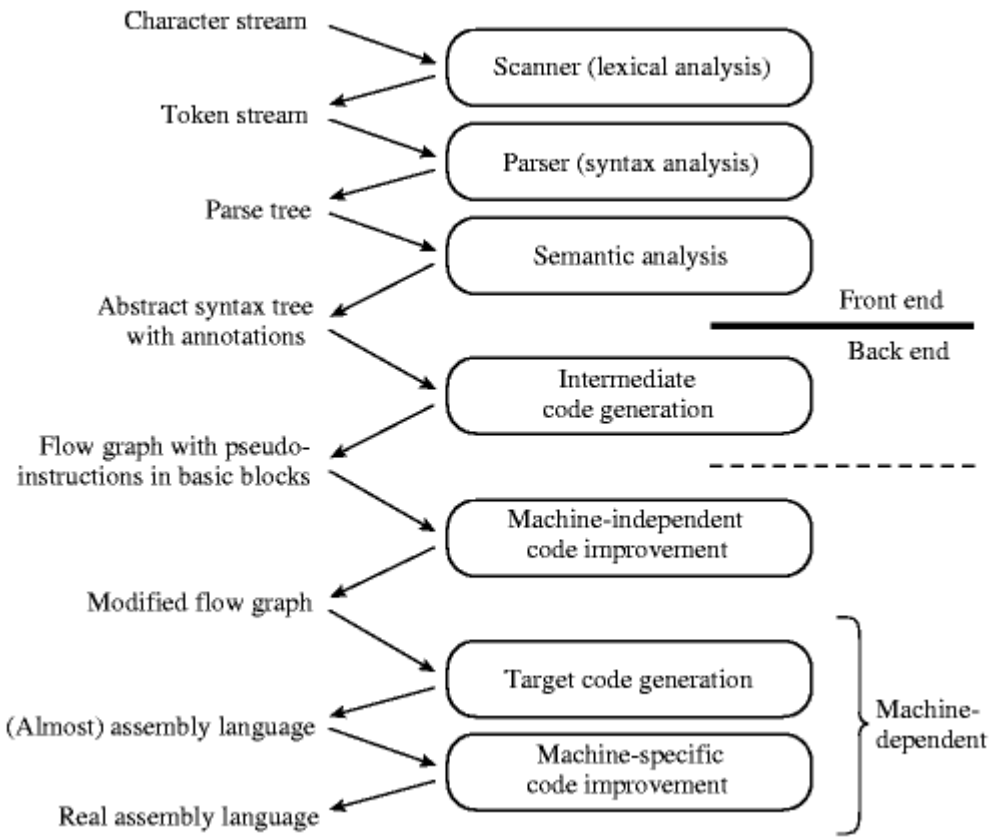
Examine intermediate code, attempting to improve it.

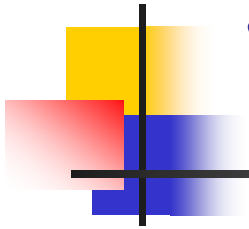
Translate intermediate code to assembly / machine code.

Examine machine code, attempting to improve it.

The Organization of a Typical Compiler

- Front-ends
  - Perform operations which depend on the language being compiled, not on the target machine
- Back-ends
  - Perform operations which must have some knowledge of the target machine





Writing the Program

- A small program, written in Pascal, which computes the greatest common divisor (GCD) of two integers

```
program gcd (input, output);
var i, j : integer;
begin
   read(i,j);
   while i <> j do
      if i > j then i := i – j
      else j :- j – i;
   writeln(i)
end.
```

From Source Code Text to Tokens

Program Source Text

```
program gcd (input, output);
var i, j : integer;
begin
   read(i,j);
   while i <> j do
      if i > j then i := i - j
      else j :- j - i;
   writeln(i)
end.
```

Tokens

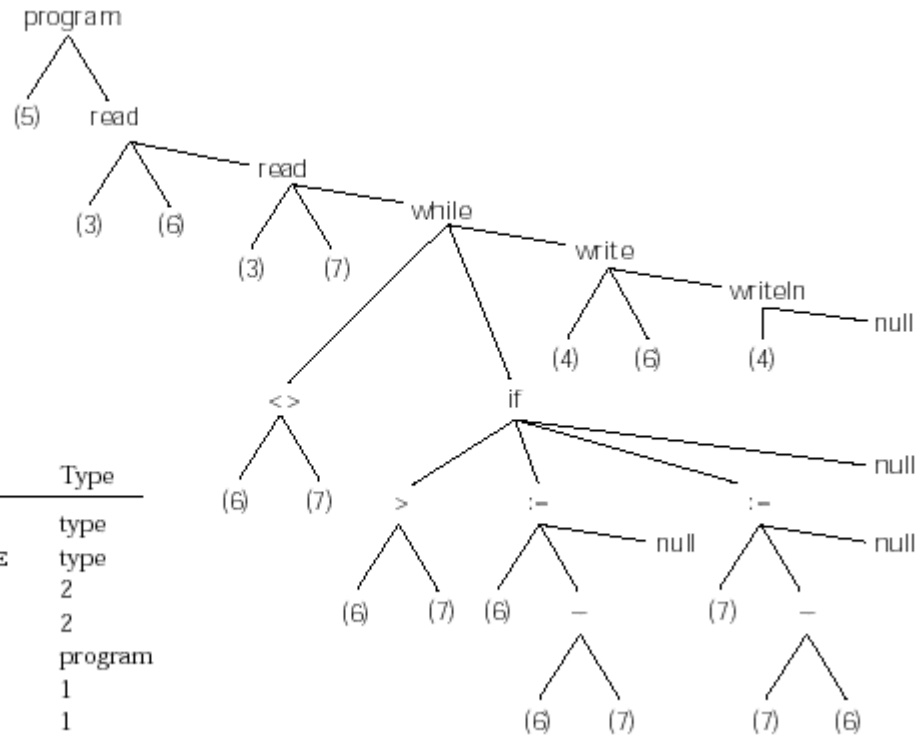
program	gcd	(
input	,	output
)	;	var
i	,	j
:	integer	;
begin	read	(
i	,	j
)	;	while
i	<>	j
do	if	i
…		

From Tokens to Parse Trees

Program Source Text

```
program gcd (input, output);
var i, j : integer;
begin
   read(i,j);
   while i <> j do
      if i > j then i := i - j
      else j :- j - i;
   writeln(i)
end.
```

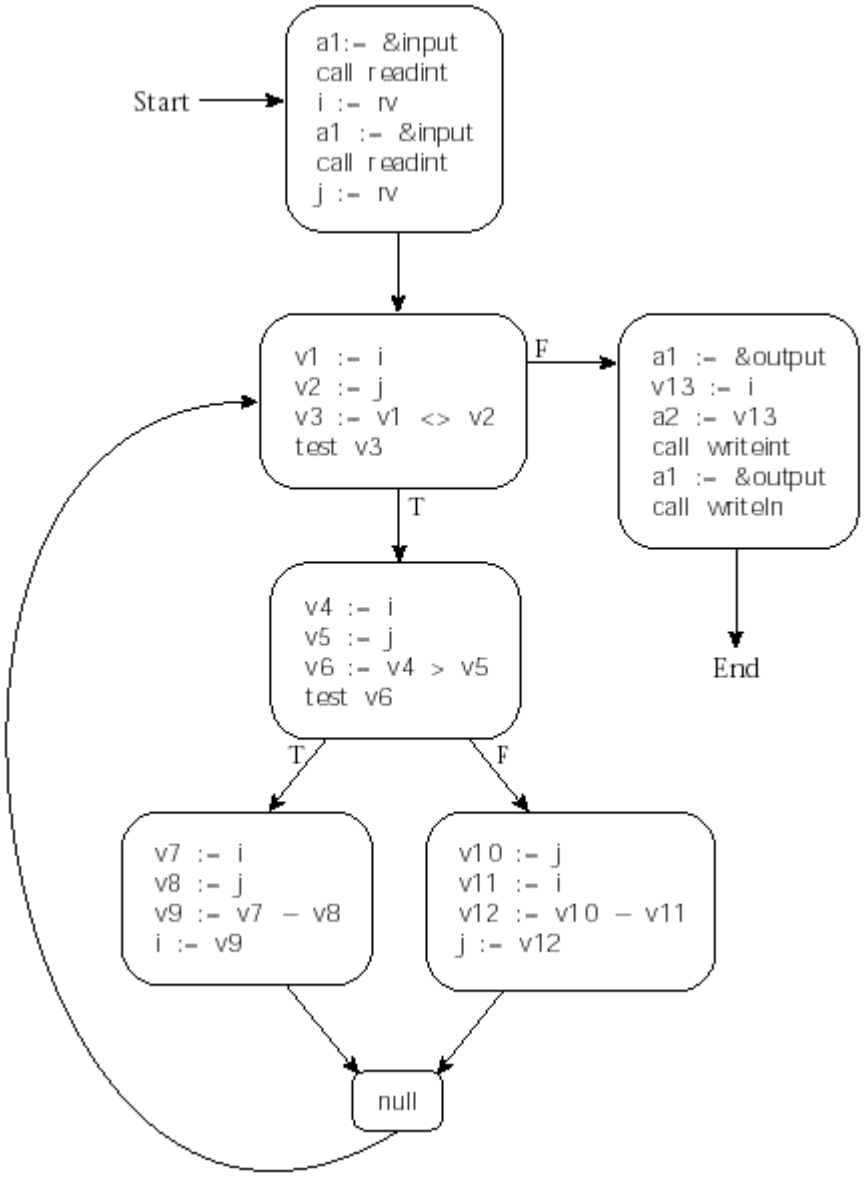
Parse Tree and Symbol Table



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

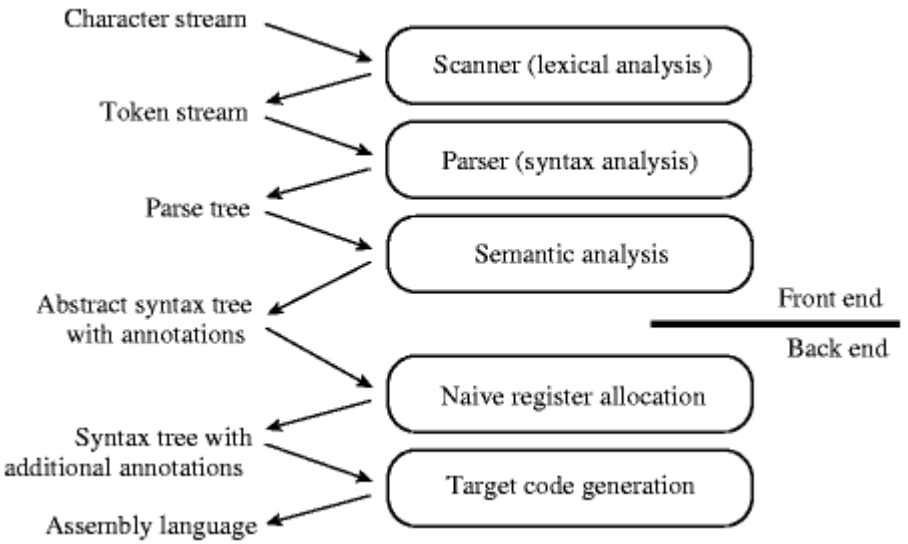
Intermediate Code

- Parse tree is converted to a control flow graph
- Nodes of control flow graph are basic blocks and contain a pseudo-assembly language
- Control can enter a basic block **only** from the beginning and exit **only** from the end



Non-optimizing Compilers

- Control flow graph is converted to target code for the target machine
- Target code is another pseudo-assembly language
- Control flow made explicit by:
  - Labeling the beginning of basic blocks
  - Converting control flow edges to branch, call, and return instructions
- Virtual registers replaced by real registers

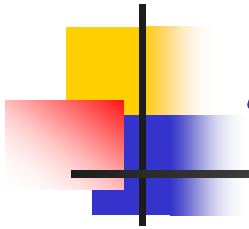


Target Code

- Target code is almost assembly code
  - Control flow is explicit
  - Code references only real register names
  - Directives for reserving storage are present
- Target code is simple to translate to assembly code

```
        -- first few lines generated during symbol table traversal
        .data           -- begin static data
        .word i         -- reserve one word to hold i
        .word j         -- reserve one word to hold j
        .text           -- begin text (code)
        -- remaining lines accumulated into program.code
main:
        a1 := &input    -- "input" and "output" are file control blocks
                        -- located in a library, to be found by the linker
        call readint    -- "readint", "writeint", and "writeln" are library subroutines
        i := rv
        a1 := &input
        call readint
        j := rv
        goto L1
L2:     r1 := i         -- body of while loop
        r2 := j
        r1 := r1 > r2
        if r1 goto L3
        r1 := j         -- "else" part
        r2 := i
        r1 := r1 - r2
        j := r1
        goto L4
L3:     r1 := i         -- "then" part
        r2 := j
        r1 := r1 - r2
        i := r1
L4:
L1:     r1 := i         -- test terminating condition
        r2 := j
        r1 := r1 <> r2
        if r1 goto L2
        a1 := &output
        r1 := i
        a2 := r1
        call writeint
        a1 := &output
        call writeln
        goto exit       -- return to operating system
```

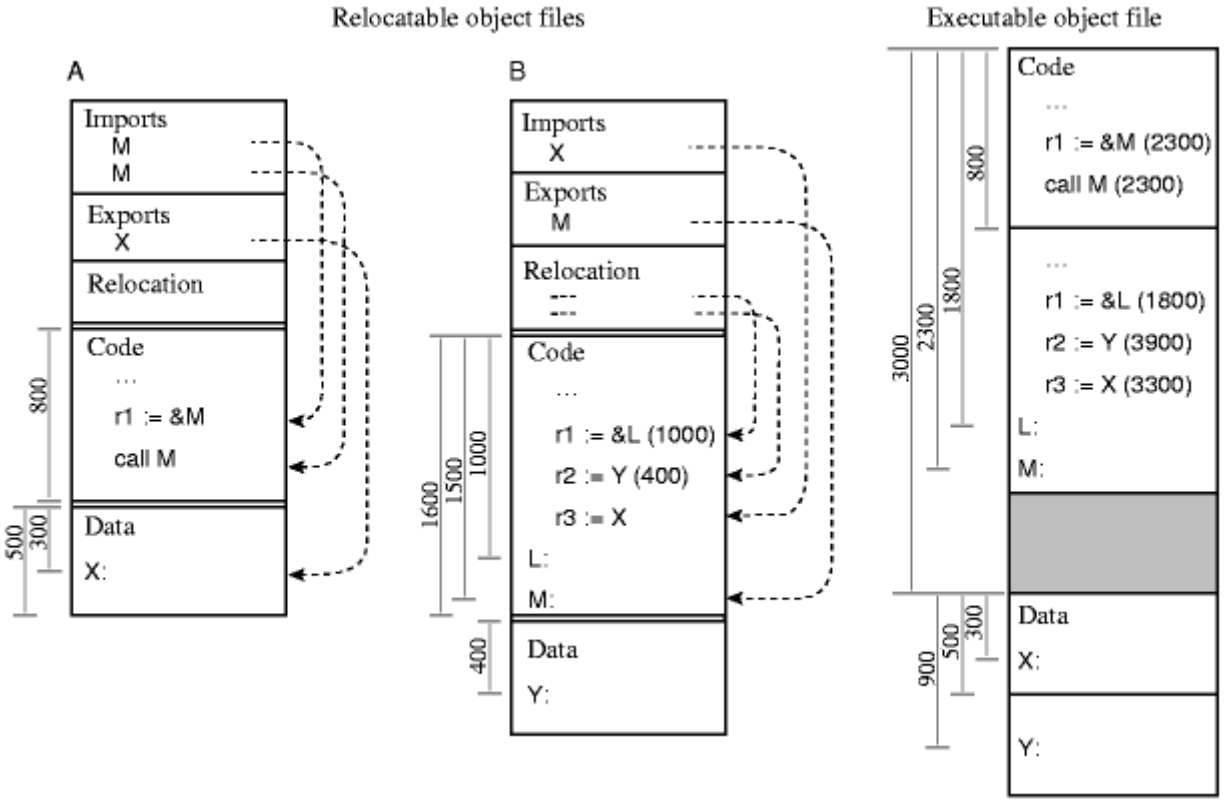
From Target Code to Assembly Code



- Usually simple:
  - r10 := r8 + r9 -> add $10, $8, $9
  - r10 := r8 + 0x12 -> addi $10, $8, 0x12
- Sometimes expand to a sequence of instructions:
  - r14 := 0x12345abc ->
    - lui $14, 0x1234
    - ori $14, 0x5abc

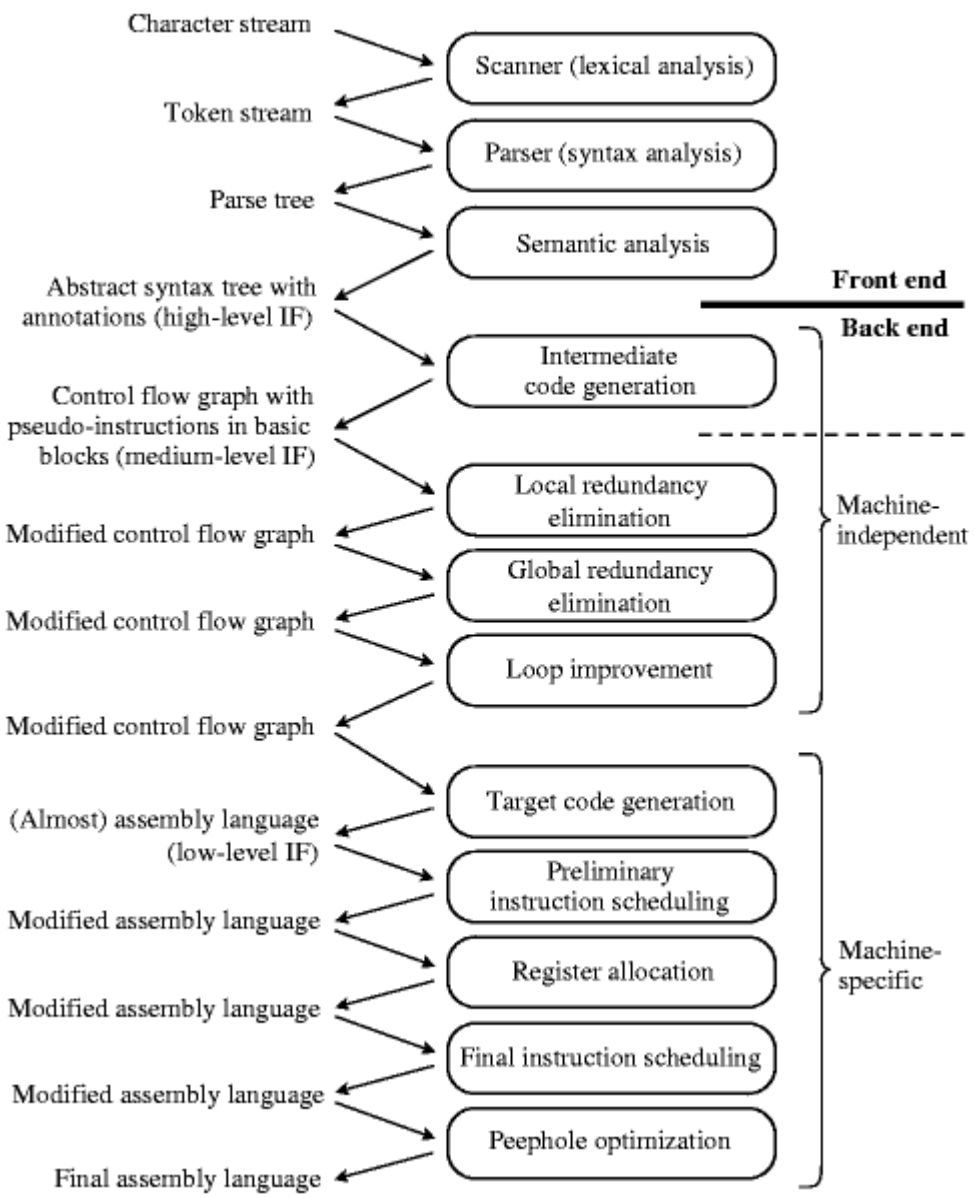
Linking

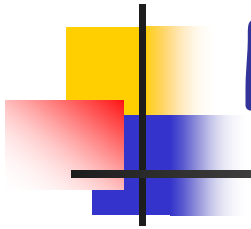
- Linking combines multiple object files produced by the assembler into a single executable file runable by the operating system



Optimizing Compilers

- Optimization is a complex process
- Improves generated code quality, at the cost of additional compilation time
- Optimizers are difficult to write, and some optimizations may not improve the final program

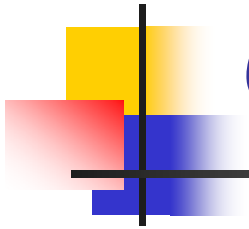




Peephole Optimization

- Look at the target code, a few instructions at a time, and try to make simple improvements
- Attempts to find short, "sub-optimal" sequences of instructions and replaces these sequences with a "better" sequence
- Sub-optimal sequences are specified as patterns
  - Mostly heuristic—no way to prove that replacing a "sub-optimal" pattern will actually improve the final program
- Simple and fairly effective

Types of Peephole Optimization



- Redundant load/store elimination

```
r2 := r1 + 5                    r2 := r1 + 5
i := r2          becomes        i := r2
r3 := i                         r4 := r2 x 3
r4 := r3 x 3
```

- Constant folding

```
r2 := 3 x 2      becomes        r2 := 6
```

- Common subexpression elimination

```
r2 := r1 x r5                   r4 := r1 x r5
r2 := r2 + r3    becomes        r2 := r4 + r3
r3 := r1 x r5                   r3 := r4
```

Types of Peephole Optimization

- Constant propogation

```
r2 := 4                          r2 := 4
r3 := r1 + r2    becomes         r3 := r1 + 4    and    r3 := r1 + 4
r2 := ...                        r2 := ...              r2 := ...
```

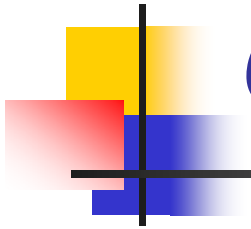
and also...

```
r2 := 4
r3 := r1 + r2    becomes         r3 := r1 + 4    and
r3 := *r3                        r3 := *r3              r3 := *(r1+4)
```

- Copy propogation

```
r2 := r1                         r2 := r1
r3 := r1 + r2    becomes         r3 := r1 + r1   and    r3 := r1 + r1
r2 := 5                          r2 := 5                r2 := 5
```

Types of Peephole Optimization

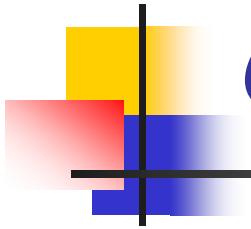


- Strength reduction

  r1 := r2 x 2
  r1 := r2 / 2
  becomes
  r1 := r2 + r2 or r1 := r2 << 1
  r1 := r2 >> 1

- Elimination of useless instructions

  r1 := r1 + 0
  r1 := r1 - 0
  r1 := r1 * 1
  becomes

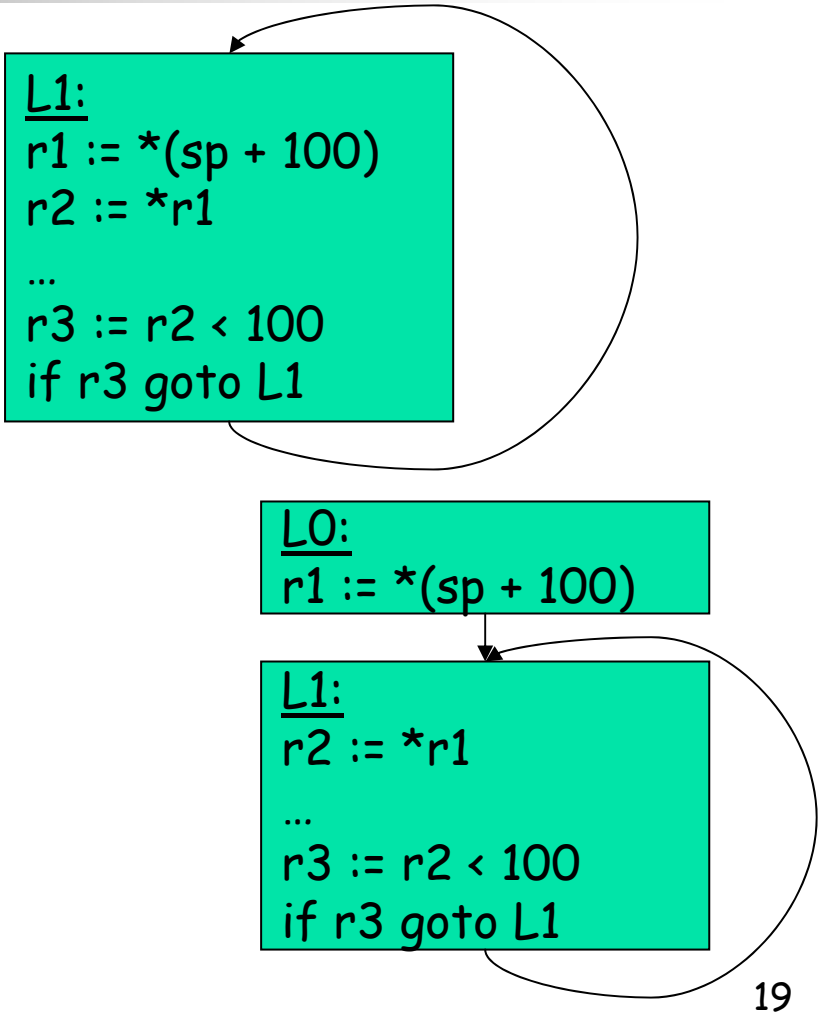


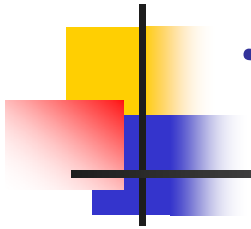
Complex Optimizations

- Require the optimizer to "understand" how data flows through registers and memory
  - Determined using dataflow analysis
    - Example: Find the set of variables which "flow" through a basic block on their way to some other block which uses them (live variables)
  - Important for optimizations which insert, delete, or move instructions which modify data
  - Original program dataflow cannot be changed!
- Require the optimizer to "understand" the structure of the control flow graph
  - Determined using control flow analysis
  - Important when optimizing the performance of loops or "loop-like" structures in the control flow graph

Complex Optimization Example

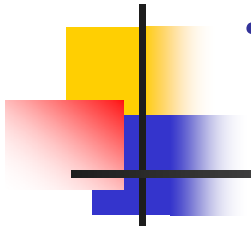
- Loop Invariant Code Motion
  - Move computations whose values remain the same (are invariant) during all iterations of the loop outside of the loop





What Have You Learned In the Last Eight Lectures?

- Programming languages are abstraction mechanisms which:
  - provide a framework for solving problems and creating new abstractions
  - shield the programmer from the low-level details of the target machine (assembly language, linking, etc.)
- Compilers are complex programs which turn high-level language programs into assembly language for subsequent execution
- Compiler writers manage the complexity of the compilation process by:
  - Breaking compiler into distinct phases
  - Using theory to build compiler components from specification



That's All

- Good luck on the final exam!
- Ask questions about the exam now or by e-mail…
- Remember:
  - Deadline for registration of final exam at Munopag: Monday July 21, 2003!
  - Final: Room HS1 in the Mathematischen Institut, Tuesday July 22, 14:00-16:00