SWEN430 - Compiler Engineering

Lecture 23 - Review

Lindsay Groves and David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

SWEN430 Exam

- **(Date)** Tuesday 19th October 2021
- **(Time)** 2:30pm – 4:30pm
- **(Location)** MCLT102

  The exam is **closed book** and split into **six** questions worth **20 marks**.

  All material covered in lectures, and assignments **examinable**.

  Note **memory models** was a new topic this year!

Exam Appendices

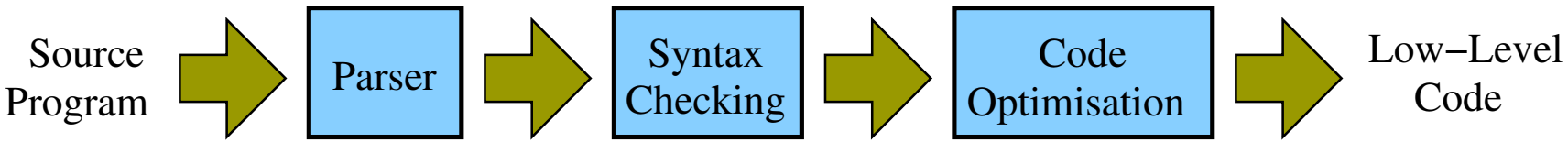
- **Appendix A: Java Bytecodes**. Provides some commonly used Java bytecodes and their effect on the stack.
- **Appendix B: x86_64 Machine Instructions**. Provides some commonly used `X86_64` instructions and their meaning.
- These appendices will be **available** in the exam!

NOTE: you should check them for any errors now.

Previous Exams

- **2019 (2 hours, 6 questions)** — set by Dave & Lindsay
- **2018 (2 hours, 6 questions)** — set by Lindsay
- **2017** — didn't run
- **2016 (3 hours, 6 questions)** — set by Dave & Lindsay
- **2015 (3 hours, 6 questions)** — set by Dave & Roman
- **2014 (3 hours, 6 questions)** — set by Dave & Alex

Lecture 1 — Introduction



- Compilers translate **source programs** into **low-level code**
- Compilers check for certain errors (e.g. type errors)
- Compilers optimise the program where possible
- Examples (all subject to active research and improvement):
  - Microsoft Visual C#/C++/F#/VB (translates into .NET IL)
  - GCC (e.g. translates C/C++ into x86)
  - GHC (translates Haskell into x86)
  - Javac (translates Java into Java bytecode)

Lecture 2 — WHILE

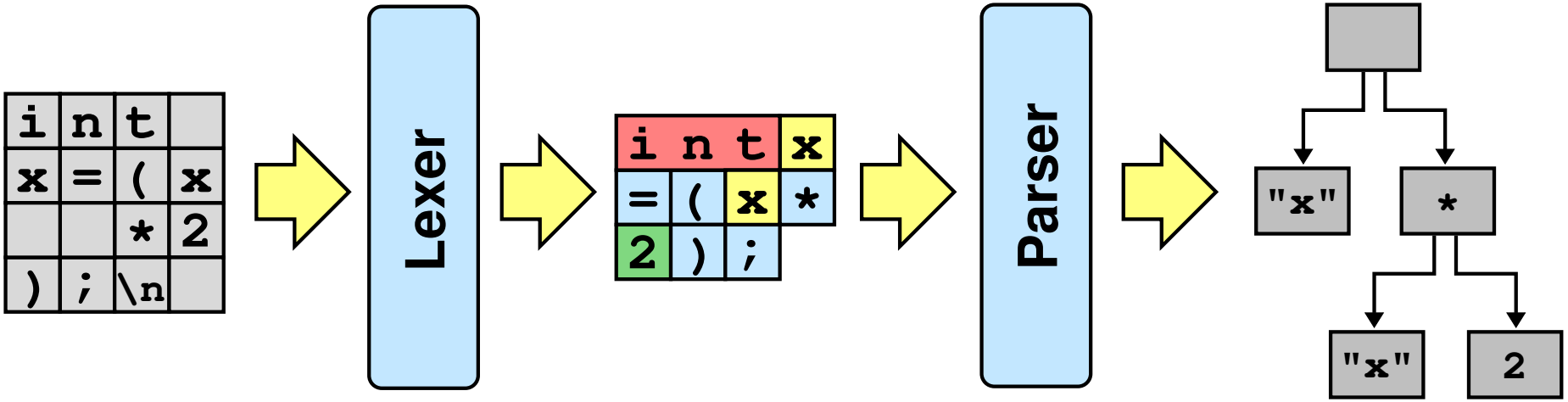
test.while

```
type Point is {int x, int y}

Point move(Point p, int dx, int dy) {
  return {x: p.x + dx, y: p.y + dy};
}
```

- A **simple** imperative language
- **Statements**: for, while, if, switch, ...
- **Expressions**: binary, unary, invocation, ...
- **Types**: bool, int, arrays, records, unions, ...

Lecture 3 — Parsing



- **Lexer.** Turns *characters* into *tokens*.

- **Parser.** Turns *tokens* into *Abstract Syntax Tree (AST)*.

Lecture 4 — Parsing

- Context-Free Grammar (CFG) is a set of rules of the form

$$N \longrightarrow \alpha_1 \mid \ldots \mid \alpha_2$$

Here, N is a *non-terminal*, and $\alpha_1, \ldots, \alpha_2$ are strings of *terminals* and non-terminals.

$$\begin{array}{lcl} E & \longrightarrow & N \mid V \mid \texttt{(}\, E \,\texttt{+}\, E \,\texttt{)} \\ N & \longrightarrow & \texttt{[0-9]}^{+} \\ V & \longrightarrow & \texttt{[a-zA-Z\_]}^{+}\texttt{[a-zA-Z0-9\_]}^{*} \end{array}$$

- Valid words in this language include:

```
1, (2 + 3), ((_x + 3) + 103282)
```

- Invalid words in this language include:

```
0x, 2 + x, ((_x + 3))
```

Lecture 5 — Operational Semantics

t	$::=$		(Terms)
	$\mid$	x	(Variables)
	$\mid$	v	(Values)
	$\mid$	$t\ t$	(App)
v	$::=$		(Values)
	$\mid$	$\lambda x.t$	(Function)
	$\mid$	c	(Integer)

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(R-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda x.t_1)\ v_1 \longrightarrow t_1[x \mapsto v_1]} \quad \text{(R-App3)}$$

- A **simple** language — *syntax* and *semantics* on one slide!
- Useful starting point for **formalising** programming languages
- Term $t_1[x \mapsto t_2]$ is t_1 with all occurences of x replaced with t_2

Lecture 6 — Semantics for $\lambda_{\mathbb{W}}$

- Semantics specified in **small step** style as transitions:

$$\langle \Sigma, \mathtt{e} \rangle \longrightarrow \langle \Sigma', \mathtt{e}' \rangle$$

  - Here, Σ represents the **call stack**
  - Call stack made up from one or more **stack frames** (or **activation records**), σ, separated by "::" (i.e. $\Sigma = \{\sigma_1 :: \sigma_2 :: \ldots :: \sigma_n\}$)
  - Each stack frame is a map from variable names to values
  - We will use notation $\Sigma(n)$ to look up the value stored in variable named *n* by checking the stack frames *from the top of the stack downwards*
  - We will use notation $\Sigma :: \{n \mapsto v\}$ to add a new mapping from variable named *n* to value *v* to the top of the stack
  - Finally, we will use notation $\Sigma[n \mapsto v]$ to replace a mapping already at the top of the stack, and if its missing, to add new mapping for *n* to the topmost stack frame.

Lecture 7 — Typing

$$\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$$

- Γ is the **typing environment**
  - A set of pairs $(\mathtt{v}, \mathtt{T})$ mapping variables to types
  - Records the declared type for every variable
- $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ is the **typing judgement**
  - In typing environment Γ, term $\mathtt{t}_1$ can be shown to have type $\mathtt{T}_1$

Lecture 8 — Typing $\lambda_{\mathtt{W}}$

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{e} : \mathtt{T}_2 \quad \mathtt{T}_2 \leq \mathtt{T}_1}{\Gamma \vdash \mathtt{x} = \mathtt{e} : \mathtt{void}} \text{ (T-Assign)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{return}\ \mathtt{e} : \mathtt{T}} \text{ (T-Return)}$$

$$\frac{\Gamma \vdash \mathtt{s}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{s}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{s}_1 \,;\, \mathtt{s}_2 : \mathtt{T}_2} \text{ (T-Sequence)}$$

- Example:

$$\{\mathtt{x} : \mathtt{int}\} \vdash \mathtt{x} = 1; \mathtt{return}\ \mathtt{x} : \mathtt{int}$$

- **NOTE:** Subtyping operator "$\leq$" discussed later!

Lecture 9 — Unreachable Code

- Consider the following Java method:

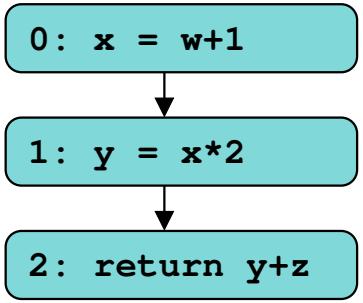
```
int f(int x) {
 if(x > 0) {
  return 1;
 } else {
  throw new NullPointerException();
 }
 x = x + 1;
 if(x > 100) { return 3; }
 return 2;
}
```

- Can this method ever return 2 or 3?
- Will this compile under `javac`?

Lecture 10 — Definite Assignment

- Consider following method, and CFG:

```
int f(int w) {
 int x,y,z;
 x = w + 1;
 y = x * 2;
 return y + z;
}
```



- The flow sets are:

$DEF_{IN}(0) = \{w\}$	$DEF_{AT}(0) = \{x\}$	$USES(0) = \{w\}$	$DEF_{OUT}(0) = \{w, x\}$
$DEF_{IN}(1) = \{w, x\}$	$DEF_{AT}(1) = \{y\}$	$USES(1) = \{x\}$	$DEF_{OUT}(1) = \{w, x, y\}$
$DEF_{IN}(2) = \{w, x, y\}$	$DEF_{AT}(2) = \emptyset$	$USES(2) = \{y, z\}$	$DEF_{OUT}(2) = \{w, x, y\}$

- Note that $DEF_{IN}(0) = \{w\}$ since w is parameter
- Java gives an error because $USES(2) - DEF_{IN}(2) = \{z\}$
- In this example, $DEF_{IN}(i+1) = DEF_{OUT}(i)$

Lecture 11 — Java Bytecode

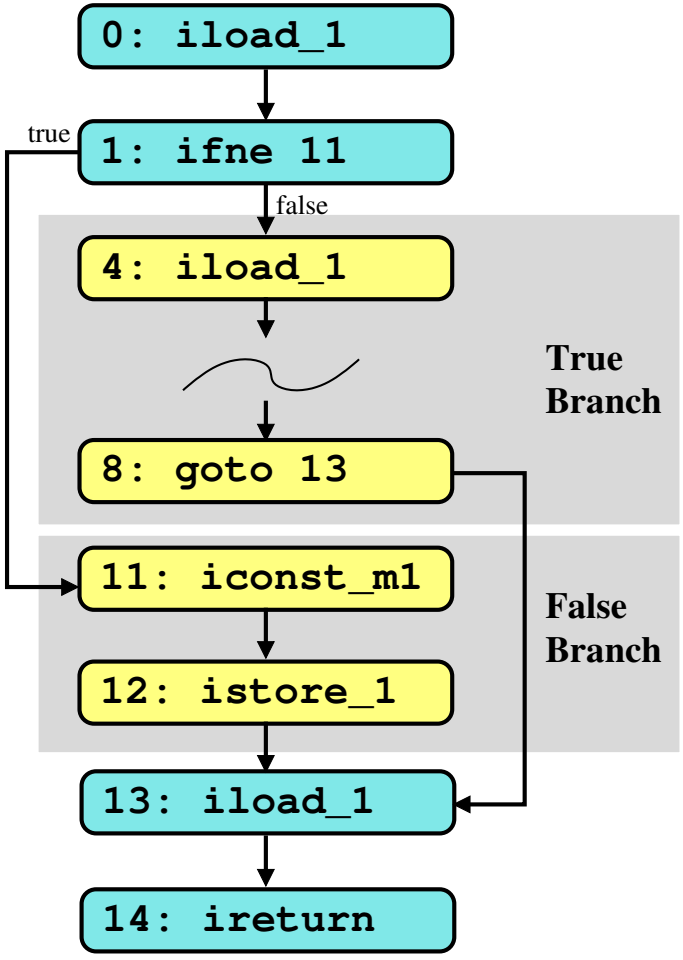
```
class Test {
 public int f(int x) {
  int y = x * 2;
  return y + x;
}}
```

```
public int f(int);
  Code:
   Stack=2, Locals=3
   0:   iload_1
   1:   iconst_2
   2:   imul
   3:   istore_2
   4:   iload_2
   5:   iload_1
   6:   iadd
   7:   ireturn
```

- A stack-based language, similar to machine code
- Can decompile Java programs with `javap`
- For details of bytecode instructions, see *JVM Specification*

Lecture 12 — Bytecode Generation

```
int f(int y) {
 if(y == 0) {
  y = y + 1;
 } else {
  y = -1;
 }
 return y;
}
```



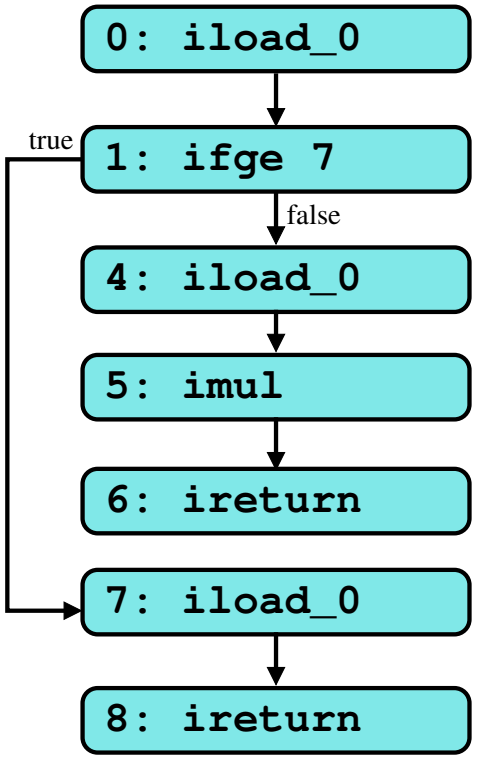
- The true branch **jumps over** the false branch!

Lecture 13 — Bytecode Generation

```
int f(java.lang.String[]);
Code:                              # diff # height
 0:   aload_1                      #  +1  #  1  #
 1:   ifnull  12                   #  -1  #  0  #
 4:   getstatic System.out         #  +1  #  1  #
 7:   ldc "Hello␣World"            #  +1  #  2  #
 9:   invokevirtual println:(Ljava/lang/String;)V
                                   #  -2  #  0  #
 12:  iconst_0                     #  +1  #  1  #
 13:  istore_2                     #  -1  #  0  #
 14:  iload_2                      #  +1  #  1  #
 15:  aload_1                      #  +1  #  2  #
 16:  arraylength                  #  0   #  2  #
 17:  iadd                         #  -1  #  1  #
 18:  istore_2                     #  -1  #  0  #
 19:  iload_2                      #  +1  #  1  #
 20:  ireturn                      #  -1  #  0  #
```

- Hence, the maximum stack height for this method is 2

Lecture 14 — Bytecode Verification




```
Exception in thread "main" java.lang.VerifyError:
(class: Test_1, method: abs signature: (I)I)
Unable to pop operand off an empty stack
```

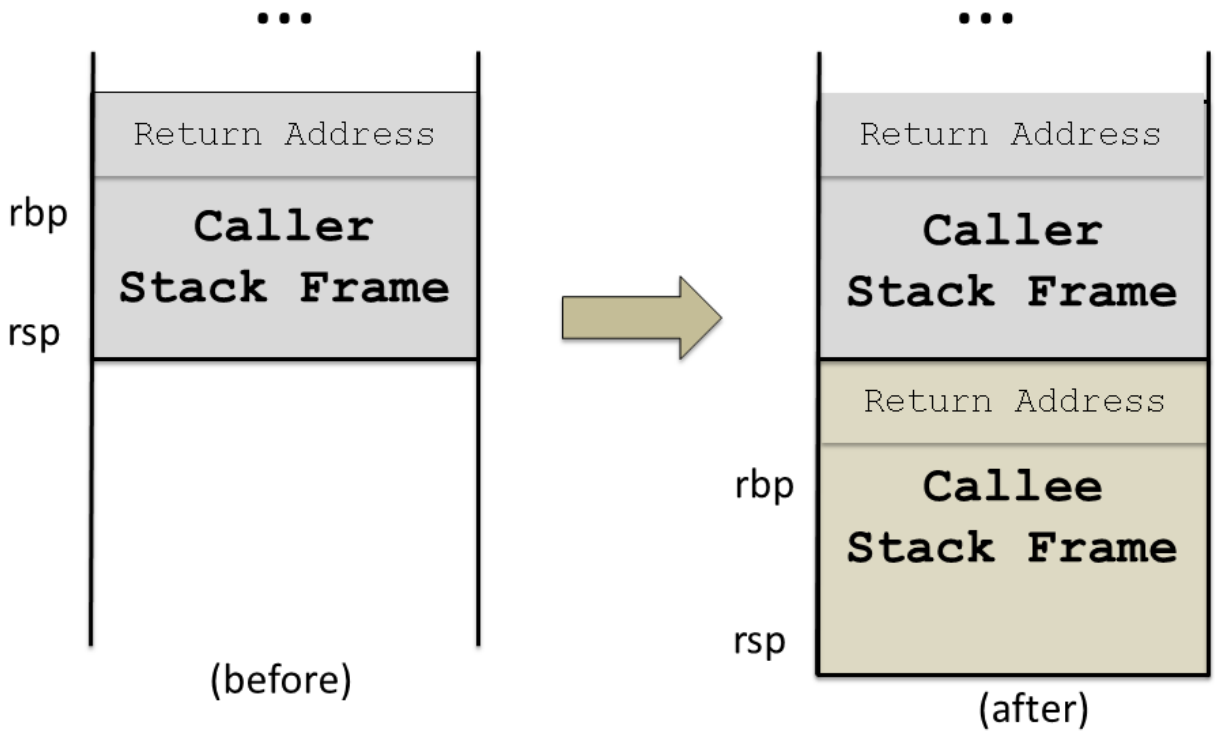
Lecture 15 — Code Generation

`movl $c, %eax`	Assign constant `c` to `eax` register	eax = c
`movl %eax, %edi`	Assign register `eax` to `edi` register	edi = eax
`addl $c, %eax`	Add constant `c` to `eax` register	eax += c
`addl %eax, %ebx`	Add `eax` register to `ebx` register	ebx += eax
`subl $c, %eax`	Substract constant `c` from `eax` register	eax -= c
`subl %eax, %ebx`	Subtract `eax` register from `ebx` register	ebx -= eax
`cmpl $0, %edx`	Compare constant `0` register against `edx` register	
`cmpl %eax, %edx`	Compare `eax` register against `edx` register	

- General form: **Instr** src, dst
- Similar range of instructions as found in JVM Bytecode
- However, x86 is a **register-based** machine code

Lecture 16 — Code Generation

`call proc`	call procedure `proc` (`push %rip + 2` ; `jmp operand`)
`ret`	return from procedure (`pop %rip`)



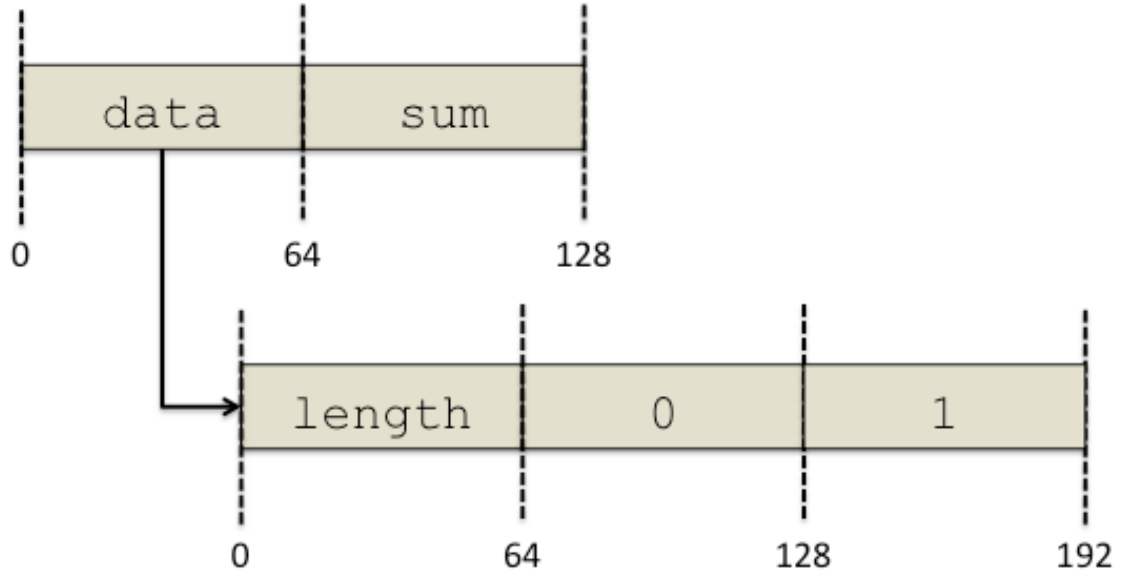
- Method calls implemented via **call** instruction
- This pushes address of **next instruction** then jumps to target

Lecture 17 — Data Representation

- For arrays, we **cannot predetermine** their width:

```
type SumList is {int[] data, int sum}
```

- Instead, arrays are **dynamically allocated**:



- Enclosing record has fixed width with array represented as **pointer**

- This makes dealing with **value semantics** quite tricky!

Lecture 17 — Code Optimisation

> *"**Peephole optimization** is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the* **peephole** *or window"* –*Wikipedia*

```
...
pushq %rax
popq  %rax
...
```

```
...
movq %eax, %eax
...
```

```
   ...
   jmp lab
lab:
   ...
```

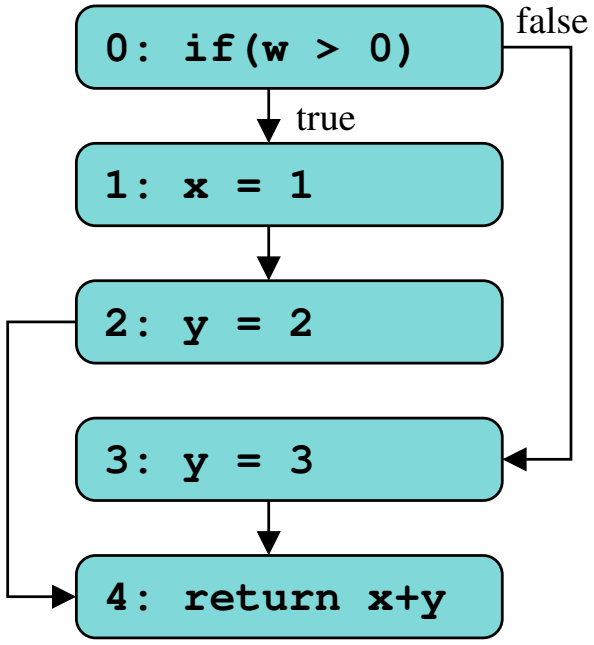
- Identify **patterns** in peephole, ignoring other instructions
- Typically only for relatively **simple** optimisations

Lecture 19 — Register Allocation

```
r0 := r1 + r2
r4 := r0 + 1
r4 := r4 * 2
syscall
```

- Micro-processors have finite number of *registers*
- If we cannot allocate all program variables to registers, then must *spill* one or more to main memory.
- Fewer clock cycles required to fetch from register than cache, or main memory
- Reducing number of registers used increases performance

Lecture 20 — Register Allocation



Node	0	1	2	3	4
0	$\emptyset$	$\{w\}$			
1	$\emptyset$	$\emptyset$			
2	$\emptyset$	$\emptyset$			
3	$\emptyset$	$\emptyset$			
4	$\emptyset$	$\{x, y\}$			

(State after one iteration)

Lecture 21 — Hardware Memory Models

Sequential Consistency

“ ... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” – Lamport’79.

Thread 1	Thread 2
`x = 1;`	`r1 = y;`
`y = 1;`	`r2 = x;`

- Can this program ever see `r1=1` and `r2=0` at the same time?

Lecture 22 — Software Memory Models

```
int x;
volatile int done;
```

Thread 1	Thread 2
`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

- In Java 5 (or later), this program cannot print `0`!
- This is because `volatile` is *synchronizing*.