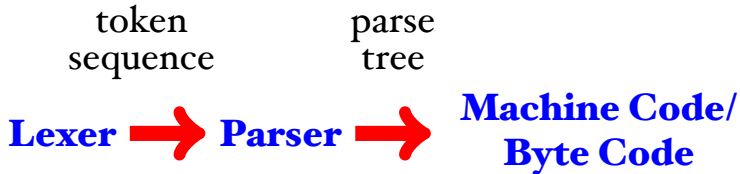


# Automata and Formal Languages (8)

Email: christian.urban at kcl.ac.uk  
Office: SI.27 (1st floor Strand Building)  
Slides: KEATS (also home work is there)

# Bird's Eye View



# JVM Code

```
ldc 1000
istore 0
iload 0
istore 1
iload 0
istore 2
iload 0
istore 3

Loop_begin_0:

ldc 0
iload 1
if_icmpge Loop_end_1

Loop_begin_2:

ldc 0
iload 2
if_icmpge Loop_end_3

if_icmpge Loop_end_5
iload 3
ldc 1
isub
istore 3
goto Loop_begin_4

Loop_end_5:

iload 0
istore 3
iload 2
ldc 1
isub
istore 2
goto Loop_begin_2

Loop_end_3:

iload 0
istore 2
```

*Stmt* → skip  
|  
*Id* := *AExp*  
|  
if *BExp* then *Block* else *Block*  
|  
while *BExp* do *Block*  
|  
read *Id*  
|  
write *Id*  
|  
write *String*

*Stmts* → *Stmt* ; *Stmts*  
|  
*Stmt*

*Block* → { *Stmts* }  
|  
*Stmt*

*AExp* → ...

*BExp* → ...

# Fibonacci Numbers

```
1  write "Fib";
2  read n;
3  minus1 := 0;
4  minus2 := 1;
5  while n > 0 do {
6      temp := minus2;
7      minus2 := minus1 + minus2;
8      minus1 := temp;
9      n := n - 1
10 };
11 write "Result";
12 write minus2
```

# Interpreter

$\text{eval}(n, E)$	$\stackrel{\text{def}}{=} n$
$\text{eval}(x, E)$	$\stackrel{\text{def}}{=} E(x) \quad \text{lookup } x \text{ in } E$
$\text{eval}(a_1 + a_2, E)$	$\stackrel{\text{def}}{=} \text{eval}(a_1, E) + \text{eval}(a_2, E)$
$\text{eval}(a_1 - a_2, E)$	$\stackrel{\text{def}}{=} \text{eval}(a_1, E) - \text{eval}(a_2, E)$
$\text{eval}(a_1 * a_2, E)$	$\stackrel{\text{def}}{=} \text{eval}(a_1, E) * \text{eval}(a_2, E)$
$\text{eval}(a_1 = a_2, E)$	$\stackrel{\text{def}}{=} \text{eval}(a_1, E) = \text{eval}(a_2, E)$
$\text{eval}(a_1 \neq a_2, E)$	$\stackrel{\text{def}}{=} \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E))$
$\text{eval}(a_1 < a_2, E)$	$\stackrel{\text{def}}{=} \text{eval}(a_1, E) < \text{eval}(a_2, E)$

# Interpreter (2)

$$\text{eval}(\text{skip}, E) \stackrel{\text{def}}{=} E$$

$$\text{eval}(x := a, E) \stackrel{\text{def}}{=} E(x \mapsto \text{eval}(a, E))$$

$$\begin{aligned} \text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) &\stackrel{\text{def}}{=} \\ &\text{if } \text{eval}(b, E) \text{ then } \text{eval}(cs_1, E) \\ &\text{else } \text{eval}(cs_2, E) \end{aligned}$$

$$\begin{aligned} \text{eval}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=} \\ &\text{if } \text{eval}(b, E) \\ &\text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\ &\text{else } E \end{aligned}$$

$$\text{eval}(\text{write } x, E) \stackrel{\text{def}}{=} \{ \text{println}(E(x)) ; E \}$$

# Test Program

```
1  start := 1000;
2  x := start;
3  y := start;
4  z := start;
5  while 0 < x do {
6    while 0 < y do {
7      while 0 < z do { z := z - 1 };
8      z := start;
9      y := y - 1
10   };
11  y := start;
12  x := x - 1
13 }
```



```
ldc 1000
istore 0
iload 0
istore 1
iload 0
istore 2
iload 0
istore 3
```

```
Loop_begin_0:
```

```
ldc 0
iload 1
if_icmpge Loop_end_1
```

```
Loop_begin_2:
```

```
ldc 0
iload 2
if_icmpge Loop_end_3
```

```
Loop_begin_4:
```

```
ldc 0
iload 3
```

```
if_icmpge Loop_end_5
iload 3
ldc 1
isub
istore 3
goto Loop_begin_4
```

```
Loop_end_5:
```

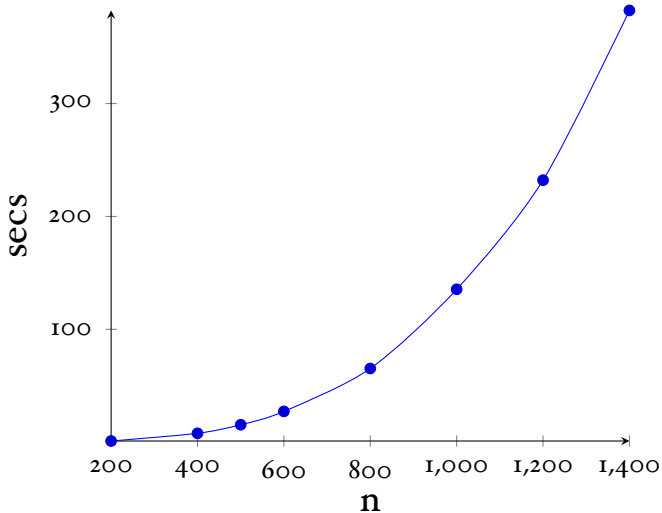
```
iload 0
istore 3
iload 2
ldc 1
isub
istore 2
goto Loop_begin_2
```

```
Loop_end_3:
```

```
iload 0
istore 2
iload 1
ldc 1
isub
istore 1
goto Loop_begin_0
```

```
Loop_end_1:
```

# Interpreted Code



# Java Virtual Machine

- introduced in 1995
- is a stack-based VM (like Postscript, CLR of .Net)
- contains a JIT compiler
- many languages take advantage of JVM's infrastructure (JRE)
- is garbage collected  $\Rightarrow$  no buffer overflows
- some languages compiled to the JVM: Scala, Clojure...

# Compiling AExps

I + 2

```
ldc 1  
ldc 2  
iadd
```

# Compiling AExps

$1 + 2 + 3$

ldc 1

ldc 2

iadd

ldc 3

iadd

# Compiling AExps

$1 + (2 + 3)$

ldc 1

ldc 2

ldc 3

iadd

iadd

# Compiling AExps

$1 + (2 + 3)$

ldc 1

ldc 2

ldc 3

iadd

iadd

dadd, fadd, ladd, ...

# Compiling AExps

$\text{compile}(n) \stackrel{\text{def}}{=} \text{ldc } n$

$\text{compile}(a_1 + a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{iadd}$

$\text{compile}(a_1 - a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{isub}$

$\text{compile}(a_1 * a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{imul}$



# Compiling AExps

$\text{compile}(n) \stackrel{\text{def}}{=} \text{ldc } n$

$\text{compile}(a_1 + a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{iadd}$

$\text{compile}(a_1 - a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{isub}$

$\text{compile}(a_1 * a_2) \stackrel{\text{def}}{=} \text{compile}(a_1) @ \text{compile}(a_2) @ \text{imul}$

# Compiling AExps

$1 + 2 * 3 + (4 - 3)$

ldc 1

ldc 2

ldc 3

imul

ldc 4

ldc 3

isub

iadd

iadd

# Variables

$$x := 5 + y * 2$$

# Variables

$$x := 5 + y * 2$$

- lookup: *iload index*
- store: *istore index*

# Variables

$$x := 5 + y * 2$$

- lookup: *iload index*
- store: *istore index*

while compiling we have to maintain a map between our identifiers and the Java bytecode indices

$$\text{compile}(a, E)$$

# Compiling AExps

$\text{compile}(n, E) \stackrel{\text{def}}{=} \text{ldc } n$

$\text{compile}(a_1 + a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{iadd}$

$\text{compile}(a_1 - a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{isub}$

$\text{compile}(a_1 * a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{imul}$

$\text{compile}(x, E) \stackrel{\text{def}}{=} \text{iload } E(x)$

# Compiling AExps

$\text{compile}(n, E) \stackrel{\text{def}}{=} \text{ldc } n$

$\text{compile}(a_1 + a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{iadd}$

$\text{compile}(a_1 - a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{isub}$

$\text{compile}(a_1 * a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{imul}$

$\text{compile}(x, E) \stackrel{\text{def}}{=} \text{iload } E(x)$

# Compiling Statements

We return a list of instructions and an environment for the variables

$$\text{compile}(\text{skip}, E) \stackrel{\text{def}}{=} (Nl, E)$$

$$\text{compile}(x := a, E) \stackrel{\text{def}}{=} \\ (\text{compile}(a, E) @ \text{istore } index, E(x \mapsto index))$$

where *index* is  $E(x)$  if it is already defined, or if it is not then the largest index not yet seen



# Compiling AExps

$x := x + I$

iload  $n_x$

ldc I

iadd

istore  $n_x$

where  $n_x$  is the index corresponding to the variable  $x$

# Compiling Ifs

if  $b$  then  $cs_1$  else  $cs_2$

code of  $b$

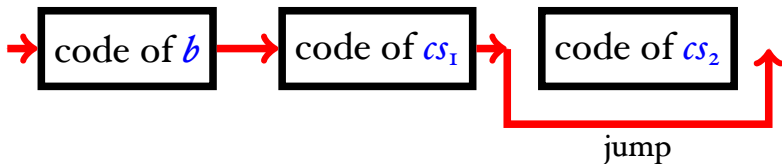
code of  $cs_1$

code of  $cs_2$

# Compiling Ifs

if  $b$  then  $cs_1$  else  $cs_2$

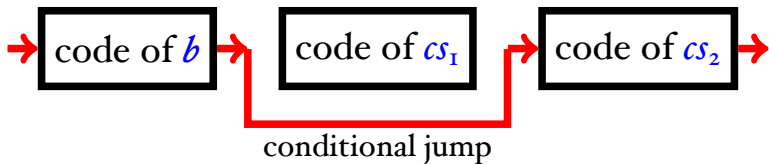
Case **True**:



# Compiling Ifs

if  $b$  then  $cs_1$  else  $cs_2$

Case **False**:



# Conditional Jumps

- `if_icmpeq label` if two ints are equal, then jump
- `if_icmpne label` if two ints aren't equal, then jump
- `if_icmpge label` if one int is greater or equal than another, then jump
- ...

# Conditional Jumps

- `if_icmpeq label` if two ints are equal, then jump
- `if_icmpne label` if two ints aren't equal, then jump
- `if_icmpge label` if one int is greater or equal than another, then jump
- ...

```
L1:  
    if_icmpeq L2  
    iload 1  
    ldc 1  
    iadd  
    if_icmpeq L1  
L2:
```

# Conditional Jumps

- `if_icmpeq` *label* if two ints are equal, then jump
- `if_icmpne` *label* if two ints aren't equal, then jump
- `if_icmpge` *label* if one int is greater or equal than another, then jump
- ...

$L_1$ :

`if_icmpeq`  $L_2$

`iload`  $r$

`ldc`  $r$

`iadd`

`if_icmpeq`  $L_1$

$L_2$ :

labels must  
be unique

# Compiling BExps

$$a_1 = a_2$$

$$\text{compile}(a_1 = a_2, E, lab) \stackrel{\text{def}}{=} \text{compile}(a_1, E) @ \text{compile}(a_2, E) @ \text{if_icmpne } lab$$



# Compiling Ifs

if  $b$  then  $cs_1$  else  $cs_2$

compile(if  $b$  then  $cs_1$  else  $cs_2, E$ )  $\stackrel{\text{def}}{=}$

$l_{ifelse}$  (fresh label)

$l_{ifend}$  (fresh label)

$(is_1, E') = \text{compile}(cs_1, E)$

$(is_2, E'') = \text{compile}(cs_2, E')$

$(\text{compile}(b, E, l_{ifelse})$

@  $is_1$

@ goto  $l_{ifend}$

@  $l_{ifelse}$  :

@  $is_2$

@  $l_{ifend}$  :,  $E''$ )

# Compiling Whiles

while *b* do *cs*

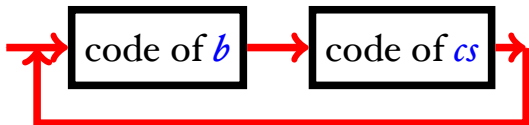
code of *b*

code of *cs*

# Compiling Whiles

while *b* do *cs*

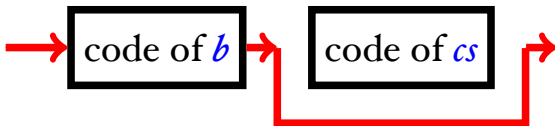
Case **True**:



# Compiling Whiles

while *b* do *cs*

Case **False**:



# Compiling Whiles

while  $b$  do  $cs$

$$\begin{aligned} \text{compile}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=} \\ &l_{wbegin} \text{ (fresh label)} \\ &l_{wend} \text{ (fresh label)} \\ &(is, E') = \text{compile}(cs_I, E) \\ &(l_{wbegin} : \\ &\quad @ \text{ compile}(b, E, l_{wend}) \\ &\quad @ is \\ &\quad @ \text{ goto } l_{wbegin} \\ &\quad @ l_{wend} :, E') \end{aligned}$$

# Compiling Writes

## write $x$

```
.method public static write(IV)      (library function)
  .limit locals 5
  .limit stack 5
  iload 0
  getstatic java/lang/System/out Ljava/io/PrintStream;
  swap
  invokevirtual java/io/PrintStream/println(IV)
  return
.end method
```

```
iload  $E(x)$ 
invokestatic write(IV)
```

```
.class public XXX.XXX  
.super java/lang/Object
```

```
.method public <init>()V  
  aload_0  
  invokenonvirtual java/lang/Object/<init>()V  
  return  
.end method
```

```
.method public static main([Ljava/lang/String;)V  
  .limit locals 200  
  .limit stack 200
```

(here comes the compiled code)

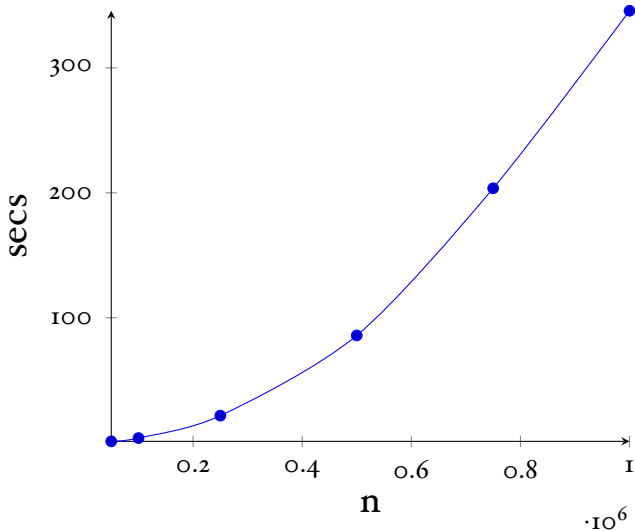
```
  return  
.end method
```

# Next Compiler Phases

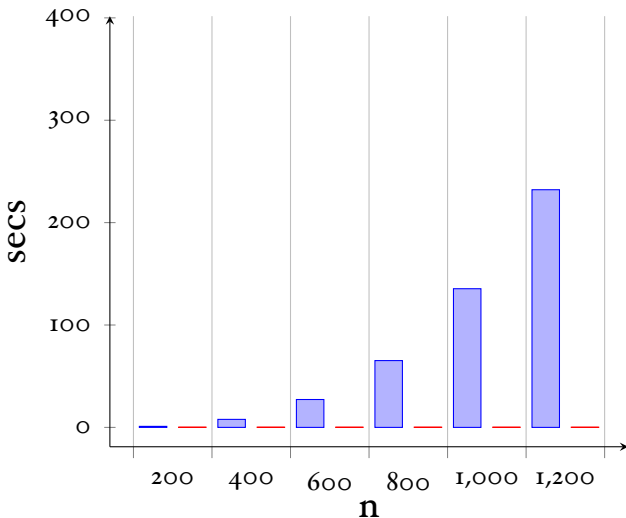
- assembly  $\Rightarrow$  byte code (class file)
- labels  $\Rightarrow$  absolute or relative jumps
  
- javap is a disassembler for class files



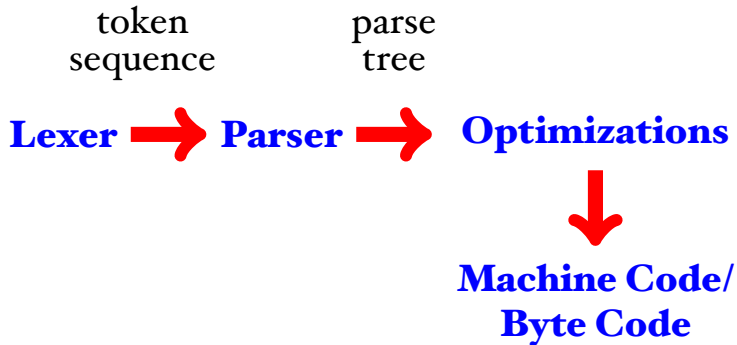
# Compiled Code



# Compiler vs. Interpreter



# Backend



# What Next

- register spilling
- dead code removal
- loop optimisations
- instruction selection
- type checking
- concurrency
- fuzzy testing
- verification
  
- GCC, LLVM, tracing JITs