Automata and Formal Languages (9)

Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)
Slides: KEATS (also home work is there)

Imagine the following situation: You talk to somebody and you find out that she/he has implemented a compiler.

What is your reaction? Check all that apply.

Imagine the following situation: You talk to somebody and you find out that she/he has implemented a compiler.

What is your reaction? Check all that apply.

- ☐ You think she/he is God
- ☐ Überhacker
- ☐ superhuman
- ☐ wizard
- ☐ supremo

While-Language

$$
\begin{array}{lcl}
Stmt & \rightarrow & \text{skip} \\
 & | & Id := AExp \\
 & | & \text{if } BExp \text{ then } Block \text{ else } Block \\
 & | & \text{while } BExp \text{ do } Block \\
 & | & \text{write } Id \\
Stmts & \rightarrow & Stmt \,;\, Stmts \\
 & | & Stmt \\
Block & \rightarrow & \{Stmts\} \\
 & | & Stmt \\
AExp & \rightarrow & \ldots \\
BExp & \rightarrow & \ldots
\end{array}
$$

Fibonacci Numbers

```
1  /* Fibonacci Program
2     input: n
3     output: fib_res */
4
5  n := 90;
6  minus1 := 0;
7  minus2 := 1;
8  temp := 0;
9  while n > 0 do {
10         temp := minus2;
11         minus2 := minus1 + minus2;
12         minus1 := temp;
13         n := n - 1
14 };
15 fib_res := minus2;
16 write fib_res
```

Interpreter

$$
\begin{array}{lcl}
\mathsf{eval}(n, E) & \stackrel{\text{def}}{=} & n \\
\mathsf{eval}(x, E) & \stackrel{\text{def}}{=} & E(x) \quad \text{lookup } x \text{ in } E \\
\mathsf{eval}(a_1 + a_2, E) & \stackrel{\text{def}}{=} & \mathsf{eval}(a_1, E) + \mathsf{eval}(a_2, E) \\
\mathsf{eval}(a_1 - a_2, E) & \stackrel{\text{def}}{=} & \mathsf{eval}(a_1, E) - \mathsf{eval}(a_2, E) \\
\mathsf{eval}(a_1 * a_2, E) & \stackrel{\text{def}}{=} & \mathsf{eval}(a_1, E) * \mathsf{eval}(a_2, E) \\
 & & \\
\mathsf{eval}(a_1 = a_2, E) & \stackrel{\text{def}}{=} & \mathsf{eval}(a_1, E) = \mathsf{eval}(a_2, E) \\
\mathsf{eval}(a_1 \mathrel{!}= a_2, E) & \stackrel{\text{def}}{=} & \neg(\mathsf{eval}(a_1, E) = \mathsf{eval}(a_2, E)) \\
\mathsf{eval}(a_1 < a_2, E) & \stackrel{\text{def}}{=} & \mathsf{eval}(a_1, E) < \mathsf{eval}(a_2, E)
\end{array}
$$

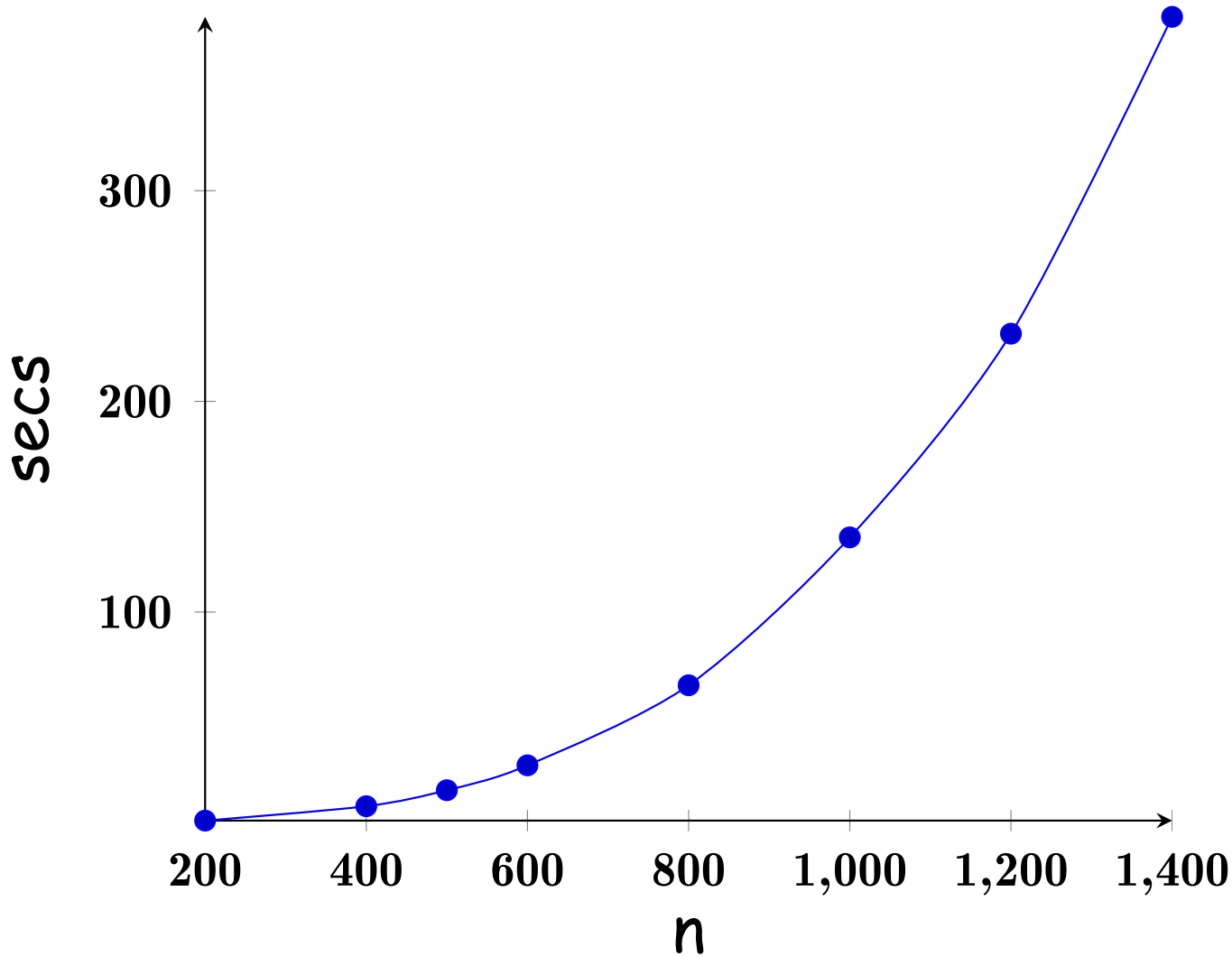
Interpreter (2)

$$\begin{array}{lcl}
\text{eval}(\text{skip}, E) & \stackrel{\text{def}}{=} & E \\
\text{eval}(x := a, E) & \stackrel{\text{def}}{=} & E(x \mapsto \text{eval}(a, E)) \\
\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad \text{if eval}(b, E) \text{ then eval}(cs_1, E) \text{ else eval}(cs_2, E)} \\
\text{eval}(\text{while } b \text{ do } cs, E) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad \text{if eval}(b, E) \text{ then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \text{ else } E} \\
\text{eval}(\text{write } x, E) & \stackrel{\text{def}}{=} & \{\ \text{println}(E(x))\ ;\ E\ \}
\end{array}$$

Test Program

```
1  start := 1;
2  x := start;
3  y := start;
4  z := start;
5  while 0 < x do {
6   while 0 < y do {
7    while 0 < z do {
8      z := z - 1
9    };
10   z := start;
11   y := y - 1
12  };
13  y := start;
14  x := x - 1
15 };
16 write x;
17 write y;
18 write z
```

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)
- languages compiled to the JVM: Scala, Clojure...
- is garbage collected $\Rightarrow$ no buffer overflows

Compiling AExps

1 + 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

$$\begin{array}{lcl}
\text{compile}(n) & \stackrel{\text{def}}{=} & \text{ldc } n \\
\text{compile}(a_1 + a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{iadd}} \\
\text{compile}(a_1 - a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{isub}} \\
\text{compile}(a_1 * a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{imul}}
\end{array}$$

Compiling AExps

$$\begin{array}{lcl}
\text{compile}(n) & \stackrel{\text{def}}{=} & \text{ldc } n \\
\text{compile}(a_1 + a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{iadd}} \\
\text{compile}(a_1 - a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{isub}} \\
\text{compile}(a_1 * a_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{imul}}
\end{array}$$

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 $number$
- store: istore $number$

Variables

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

- lookup: iload $number$
- store: istore $number$

during compilation we have to maintain a map between our identifiers and the Java bytecode numbers

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

Compiling AExps

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

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

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

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

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

Compiling AExps

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

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

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

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

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

Compiling Statements

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

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

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

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

Compiling AExps

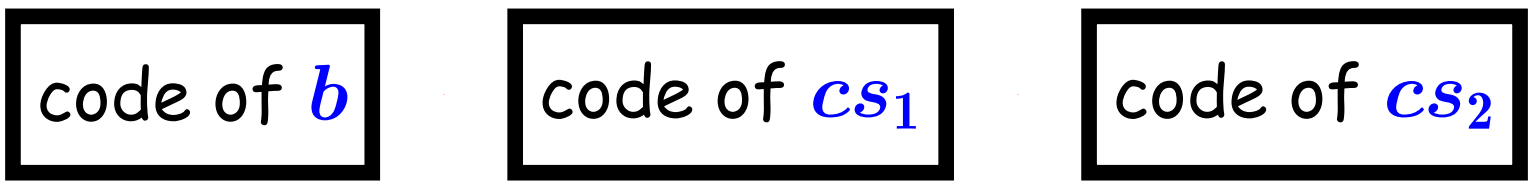
$x := x + 1$

iload n_x
ldc 1
iadd
istore n_x

where n_x is the number corresponding to the variable x

Compiling Ifs

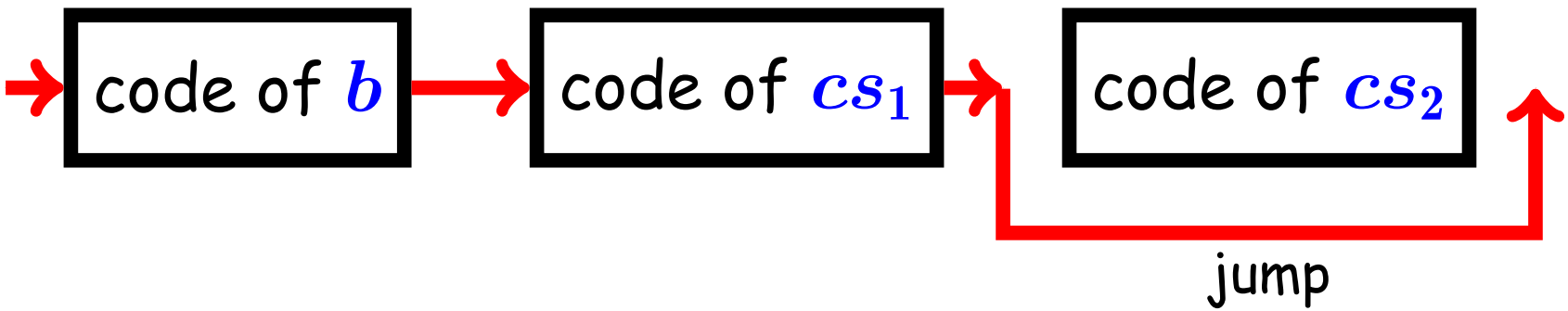
if b else cs_1 then cs_2



Compiling Ifs

if b else cs_1 then cs_2

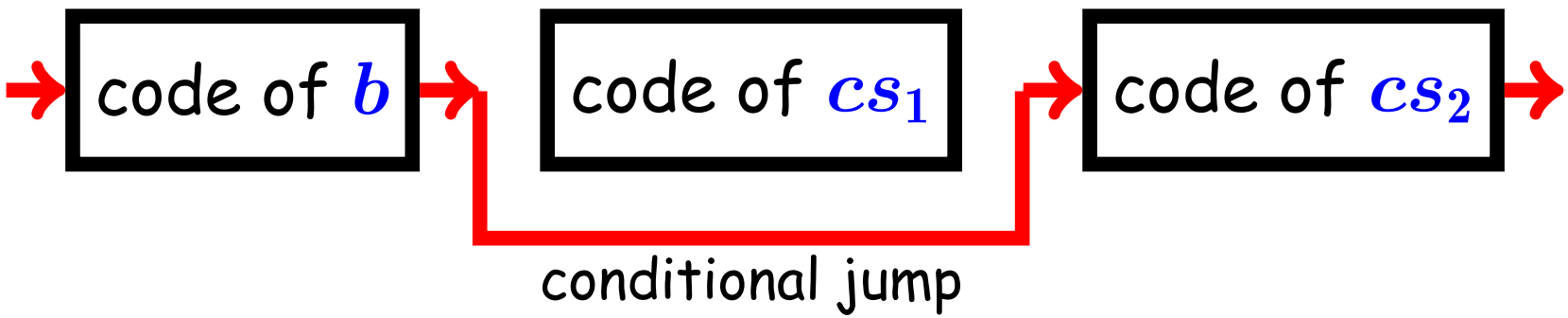
Case True:



Compiling Ifs

if b else cs_1 then 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 then another, then jump

...

labels must be unique

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 then another, then jump

...

L_1:
    if_icmpeq L_2
    iload 1
    ldc 1
    iadd
    if_icmpeq L_1
L_2:

labels must be unique

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 then another, then jump

...

L_1:
    if_icmpeq L_2
    iload 1
    ldc 1
    iadd
    if_icmpeq L_1
L_2:

labels must be unique

Compiling BExps

$$a_1 = a_2$$

$$\text{compile}(a_1 = a_2, E, lab) \stackrel{\text{def}}{=}$$
$$\text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2.E) \mathbin{@} \text{if\_icmpne } lab$$

Compiling Ifs

if b then cs_1 else cs_2

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

l_{ifelse} (fresh label)

l_{ifend} (fresh label)

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

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

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

$@\ is_1$

$@\ \text{goto}\ l_{ifend}$

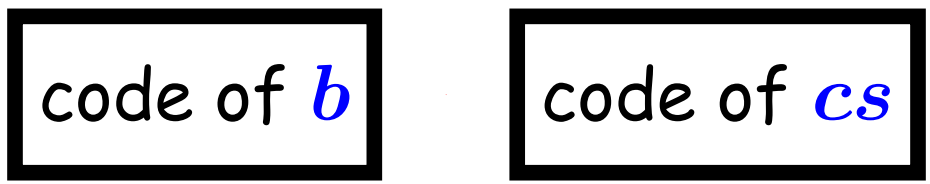
$@\ l_{ifelse}:$

$@\ is_2$

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

Compiling Whiles

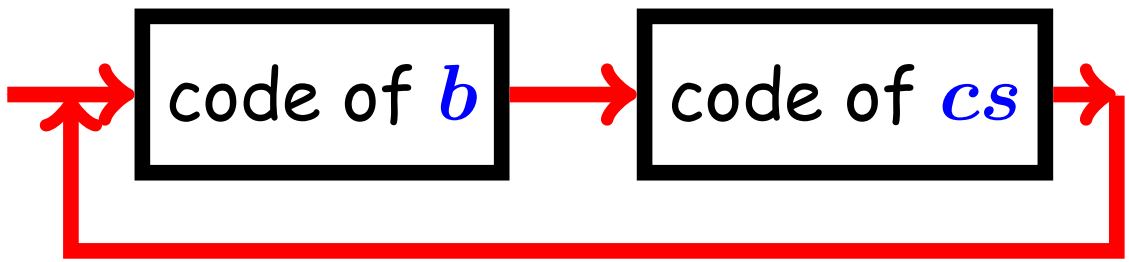
while b do cs



Compiling Whiles

while b do cs

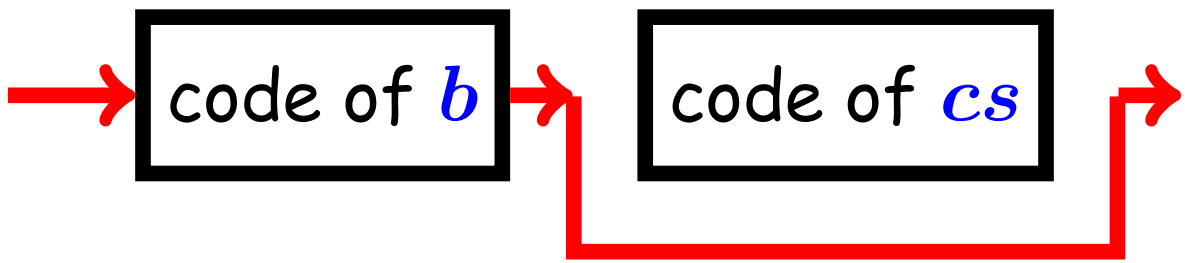
Case True:



Compiling Whiles

while b do cs

Case False:



Compiling Whiles

while b do cs

$$\begin{array}{l}\text{compile}(\text{while } b \text{ do } cs, E) \stackrel{\text{def}}{=} \\ \quad l_{wbegin} \text{ (fresh label)} \\ \quad l_{wend} \text{ (fresh label)} \\ \quad (is, E') = \text{compile}(cs_1, E) \\ \quad (l_{wbegin} : \\ \quad\ @\ \text{compile}(b, E, l_{wend}) \\ \quad\ @\ is \\ \quad\ @\ \text{goto } l_{wbegin} \\ \quad\ @\ l_{wend} :, E'')\end{array}$$

Compiling Writes

write x

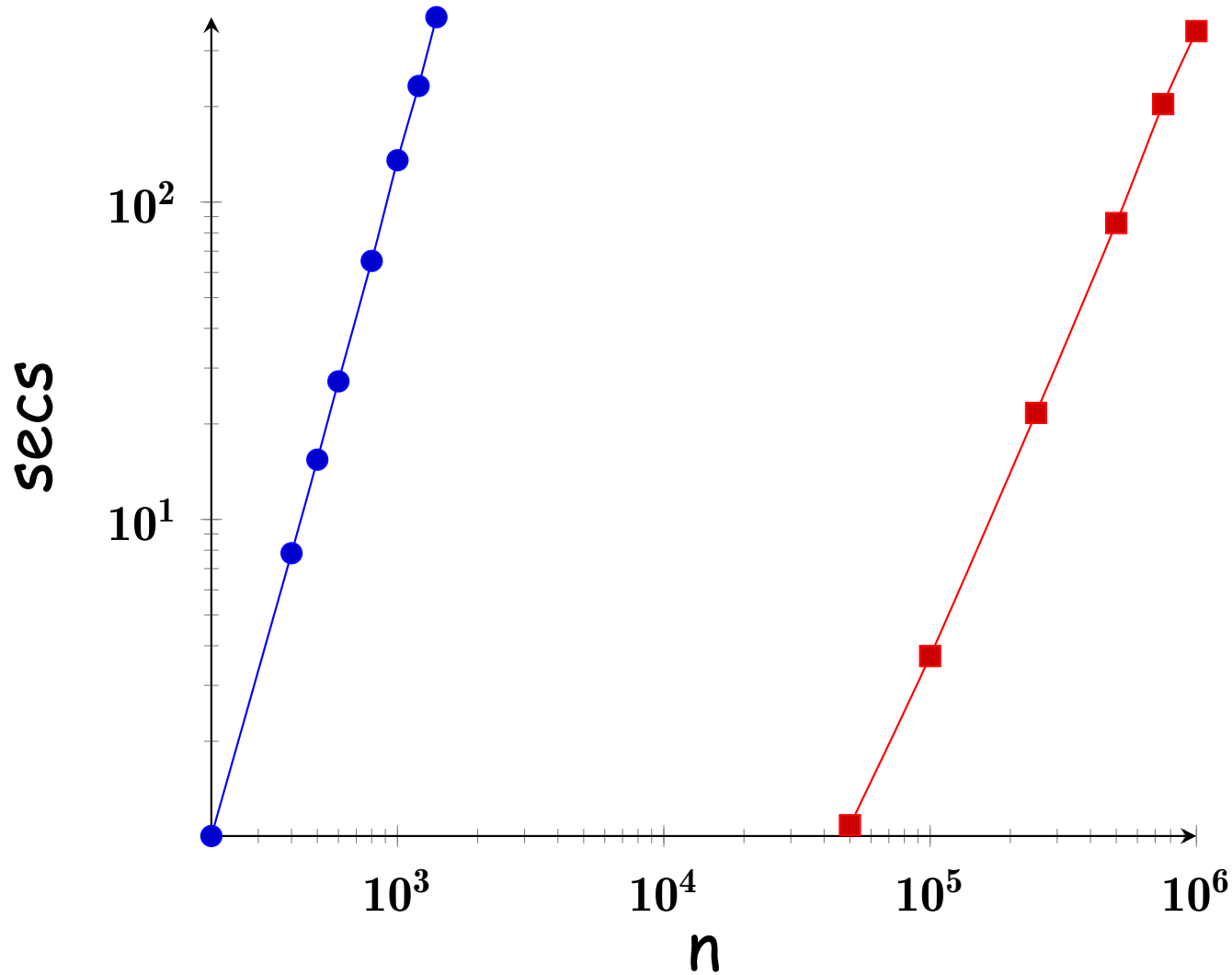
```
.method public static write(I)V
  .limit locals 5
  .limit stack 5
  iload 0
  getstatic java/lang/System/out Ljava/io/PrintStream;
  swap
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

iload $E(x)$
invokestatic write(I)V

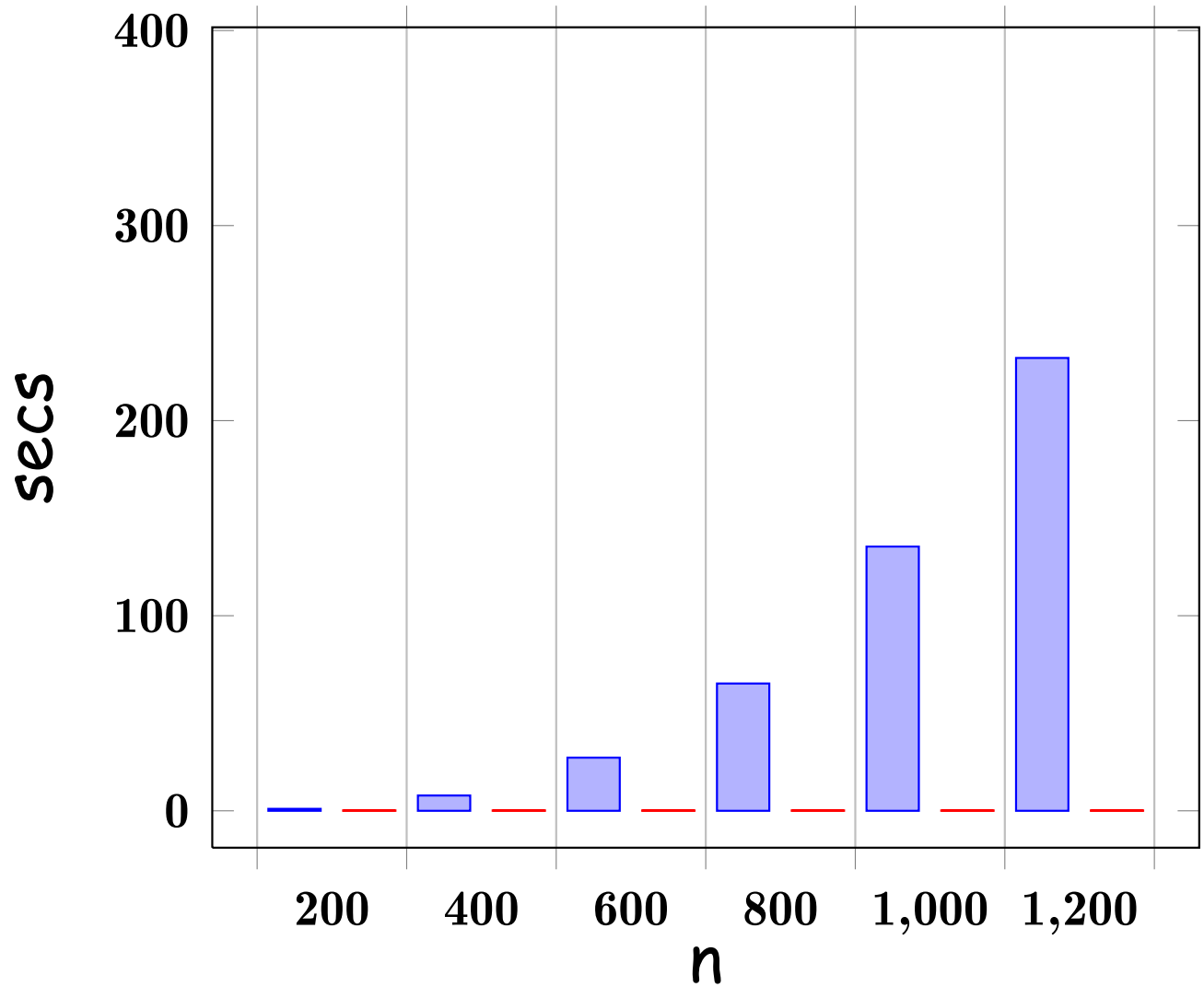
Next Compiler Phases

- assembly $\Rightarrow$ native code
- labels $\Rightarrow$ absolute or relative addresses

Compiled vs. Interpreted Code



Compiled vs. Interpreted Code



What Next

- register spilling
- dead code removal
- loop optimisations
- instruction selection
- type checking
- concurrency
- fuzzy testing
- verification

- GCC, LLVM, tracing JITs