Compilers and Formal Languages (7)

Email: christian.urban at kcl.ac.uk
Office: N7.07 (North Wing, Bush House)
Slides: KEATS (also home work is there)

Bird's Eye View

token sequence → parse tree

Lexer → **Parser** → **Machine Code/ Java Byte Code**

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

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
```

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

Fibonacci Numbers

```
write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

Interpreter

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

Interpreter (2)

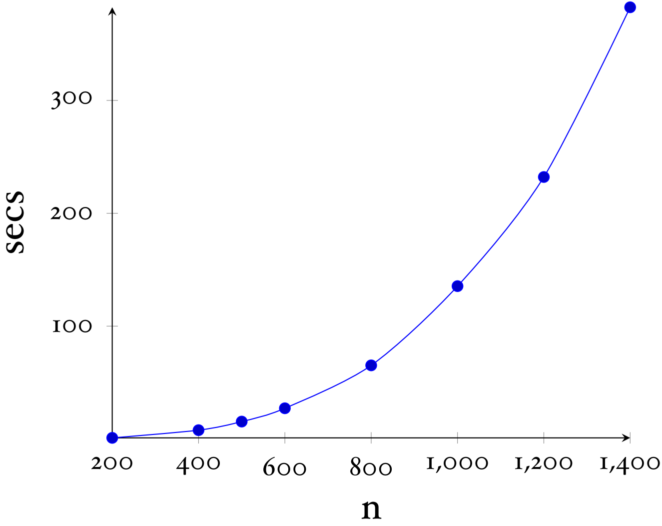
$$
\begin{aligned}
\text{eval}(\text{skip}, E) &\stackrel{\text{def}}{=} E\\
\text{eval}(x := a, E) &\stackrel{\text{def}}{=} E(x \mapsto \text{eval}(a, E))\\
\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) &\stackrel{\text{def}}{=}\\
&\quad \text{if eval}(b, E) \text{ then eval}(cs_1, E)\\
&\quad \phantom{\text{if eval}(b, E)} \text{ else eval}(cs_2, E)\\
\text{eval}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=}\\
&\quad \text{if eval}(b, E)\\
&\quad \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E))\\
&\quad \text{else } E\\
\text{eval}(\text{write } x, E) &\stackrel{\text{def}}{=} \{\ \text{println}(E(x))\ ;\ E\ \}
\end{aligned}
$$

Test Program

```
start := 1000;
x := start;
y := start;
z := start;
while 0 < x do {
 while 0 < y do {
  while 0 < z do { z := z - 1 };
  z := start;
  y := y - 1
 };
 y := start;
 x := x - 1
}
```

```
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

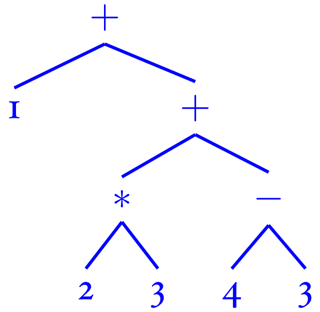


Java Virtual Machine

- 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

For example $1 + ((2 * 3) + (4 - 3))$:




```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

Traverse tree in post-order ⇒ code for stack-machine

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

$$\begin{aligned}
\text{compile}(n) &\stackrel{\text{def}}{=} \text{ldc } n\\
\text{compile}(a_1 + a_2) &\stackrel{\text{def}}{=}\\
&\quad \text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{iadd}\\
\text{compile}(a_1 - a_2) &\stackrel{\text{def}}{=}\\
&\quad \text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{isub}\\
\text{compile}(a_1 * a_2) &\stackrel{\text{def}}{=}\\
&\quad \text{compile}(a_1) \mathbin{@} \text{compile}(a_2) \mathbin{@} \text{imul}
\end{aligned}$$

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

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

Compiling AExps

$$\begin{aligned}
\text{compile}(n, E) &\stackrel{\text{def}}{=} \text{ldc } n \\
\text{compile}(a_1 + a_2, E) &\stackrel{\text{def}}{=} \\
&\quad \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{iadd} \\
\text{compile}(a_1 - a_2, E) &\stackrel{\text{def}}{=} \\
&\quad \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{isub} \\
\text{compile}(a_1 * a_2, E) &\stackrel{\text{def}}{=} \\
&\quad \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{imul} \\
\text{compile}(x, E) &\stackrel{\text{def}}{=} \text{iload } E(x)
\end{aligned}$$

Mathematical Functions

Compilation of some mathematical functions:

```
Aop("+", a1, a2)  ⇒  ...iadd
Aop("-", a1, a2)  ⇒  ...isub
Aop("*", a1, a2)  ⇒  ...imul
Aop("/", a1, a2)  ⇒  ...idiv
Aop("%", a1, a2)  ⇒  ...irem
```

Compiling Statements

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

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

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

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

Compiling Assignments

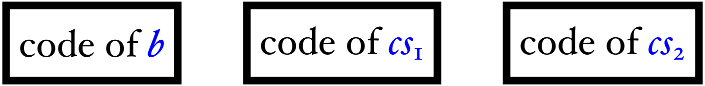
$x := x + 1$

iload n_x
ldc 1
iadd
istore n_x

where n_x is the index corresponding to the variable x

Compiling Ifs

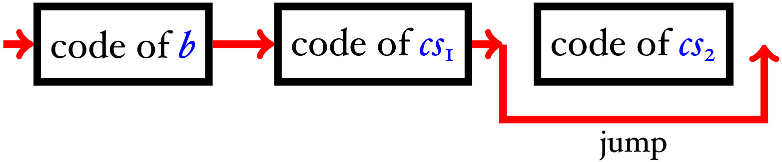
if b then cs_1 else cs_2



Compiling Ifs

if b then cs_1 else cs_2

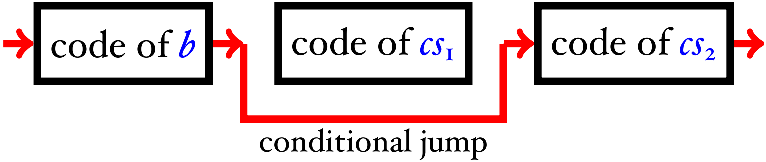
Case **True**:



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 then 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 then 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 then another, then jump

...

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

labels must be unique

Compiling Ifs

For example

```
if 1 = 1 then x := 2 else y := 3
```

```
    ldc 1
    ldc 1
    if_icmpne L_ifelse
    ldc 2
    istore 0
    goto L_ifend
L_ifelse:
    ldc 3
    istore 1
L_ifend:
```

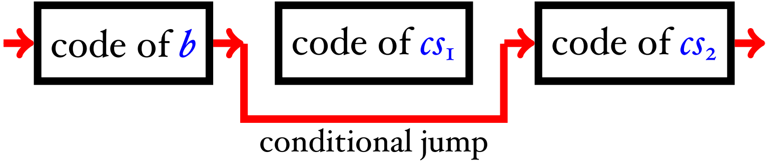
Compiling BExps

$a_1 = a_2$

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

Boolean Expressions

Compilation of boolean expressions:




```
Bop(”==”, a1, a2)  ⇒  ...if_icmpne...
Bop(”!=”, a1, a2)  ⇒  ...if_icmpeq...
Bop(”<”, a1, a2)   ⇒  ...if_icmpge...
Bop(”<=”, a1, a2)  ⇒  ...if_icmpgt...
```

Compiling Ifs

if b then cs_1 else cs_2

$$
\begin{array}{l}
\text{compile}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) \quad \stackrel{\text{def}}{=} \\
\quad l_{ifelse} \quad \text{(fresh label)} \\
\quad l_{ifend} \quad \text{(fresh label)} \\
\quad (is_1, E') = \text{compile}(cs_1, E) \\
\quad (is_2, E'') = \text{compile}(cs_2, E') \\
\quad (\text{compile}(b, E, l_{ifelse}) \\
\quad\; @\ is_1 \\
\quad\; @\ \text{goto } l_{ifend} \\
\quad\; @\ l_{ifelse} : \\
\quad\; @\ is_2 \\
\quad\; @\ l_{ifend} :, E'')
\end{array}
$$

Compiling Whiles

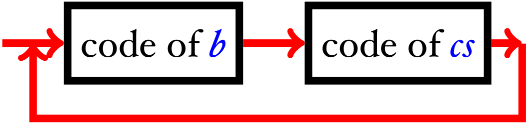
while *b* do *cs*

code of *b*   code of *cs*

Compiling Whiles

while *b* do *cs*

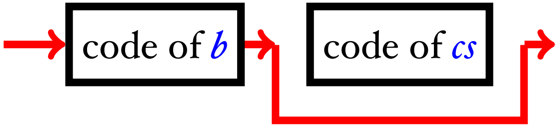
Case **True**:



Compiling Whiles

while b do cs

Case **False**:



Compiling Whiles

while b do cs

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

Compiling Whiles

For example

```
while x <= 10 do x := x + 1
```

```
L_wbegin:
    iload 0
    ldc 10
    if_icmpgt L_wend
    iload 0
    ldc 1
    iadd
    istore 0
    goto L_wbegin
L_wend:
```

Compiling Writes

```
.method public static write(I)V
  .limit locals 1
  .limit stack 2
  getstatic java/lang/System/out
                              Ljava/io/PrintStream;
  iload 0
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

```
iload E(x)
invokestatic XXX/XXX/write(I)V
```

Compiling Main

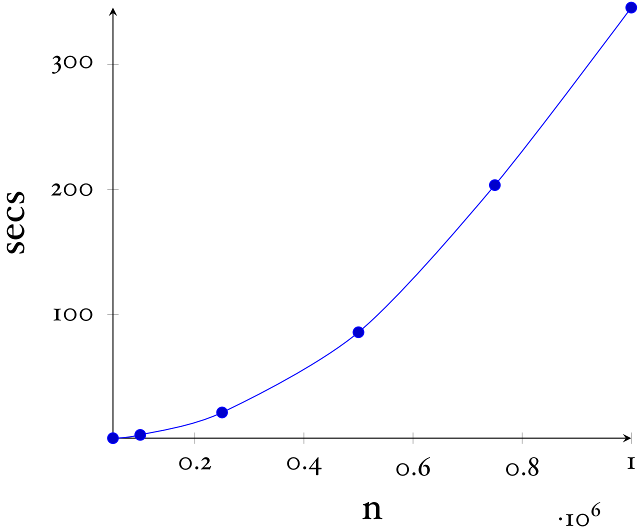
```
.class public XXX.XXX
.super java/lang/Object

.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

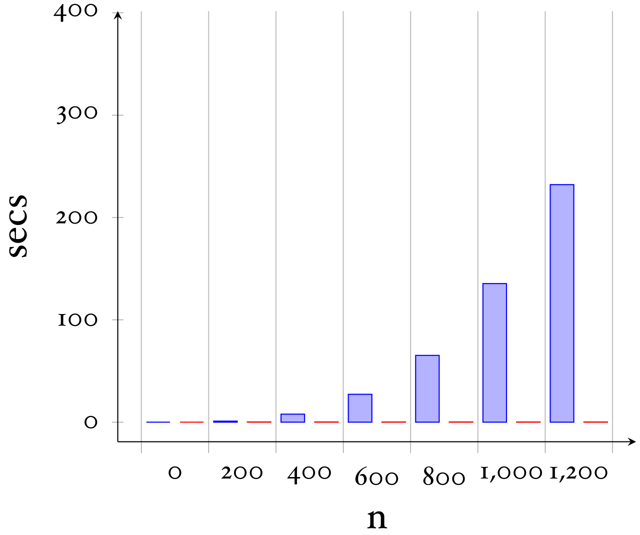
- assembly $\Rightarrow$ byte code (class file)
- labels $\Rightarrow$ absolute or relative jumps

- `javap` is a disassembler for class files

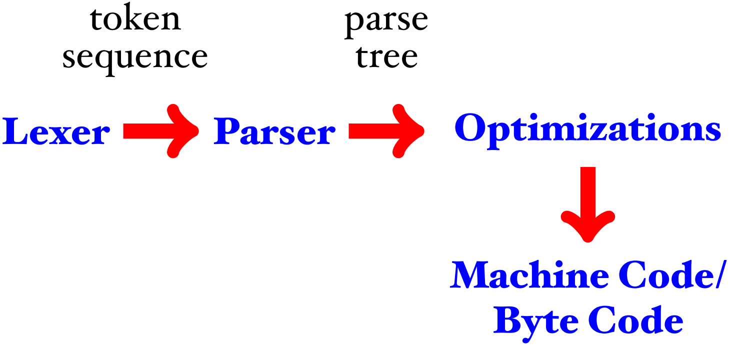
Compiled Code



Compiler vs. Interpreter



Backend


token sequence
parse tree
Lexer
Parser
Optimizations
Machine Code/ Byte Code

What is Next

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

- GCC, LLVM, tracing JITs

Coursework: MkEps

$$\begin{array}{lcl}
mkeps([c_1 c_2 \ldots c_n]) & \stackrel{\text{def}}{=} & undefined \\
mkeps(r^*) & \stackrel{\text{def}}{=} & Stars\;[] \\
mkeps(r^{\{n\}}) & \stackrel{\text{def}}{=} & Stars\;(mkeps(r))^n \\
mkeps(r^{\{n..\}}) & \stackrel{\text{def}}{=} & Stars\;(mkeps(r))^n \\
mkeps(r^{\{..n\}}) & \stackrel{\text{def}}{=} & Stars\;[] \\
mkeps(r^{\{n..m\}}) & \stackrel{\text{def}}{=} & Stars\;(mkeps(r))^n \\
mkeps(r^+) & \stackrel{\text{def}}{=} & mkeps(r^{\{1..\}}) \\
mkeps(r^?) & \stackrel{\text{def}}{=} & mkeps(r^{\{..1\}})
\end{array}$$

Coursework: Inj

$$
\begin{array}{lcl}
inj\,([c_1 c_2 \ldots c_n])\;c\;Empty & \stackrel{\text{def}}{=} & Chr\;c \\
inj\,(r^*)\;c\;Seq\;v\;(Stars\;vs) & \stackrel{\text{def}}{=} & Stars\;(inj\;r\;c\;v :: vs) \\
inj\,(r^{\{n\}})\;c\;Seq\;v\;(Stars\;vs) & \stackrel{\text{def}}{=} & Stars\;(inj\;r\;c\;v :: vs) \\
inj\,(r^{\{n..\}})\;c\;Seq\;v\;(Stars\;vs) & \stackrel{\text{def}}{=} & Stars\;(inj\;r\;c\;v :: vs) \\
inj\,(r^{\{..n\}})\;c\;Seq\;v\;(Stars\;vs) & \stackrel{\text{def}}{=} & Stars\;(inj\;r\;c\;v :: vs) \\
inj\,(r^{\{n..m\}})\;c\;Seq\;v\;(Stars\;vs) & \stackrel{\text{def}}{=} & Stars\;(inj\;r\;c\;v :: vs) \\
\\
inj\,(r^+)\;c\;v & \stackrel{\text{def}}{=} & inj\,(r^{\{1..\}})\;c\;v \\
inj\,(r^?)\;c\;v & \stackrel{\text{def}}{=} & inj\,(r^{\{..1\}})\;c\;v
\end{array}
$$