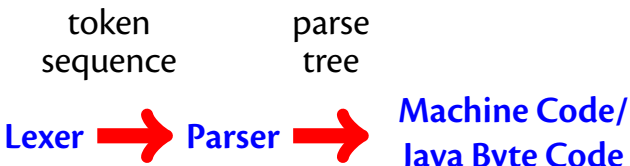
Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk

Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Bird's Eye View



CW3

Atomic parsers for tokens

$$\texttt{T\_Num(123)} :: \textit{rest} \;\Rightarrow\; \{(\texttt{T\_Num(123)}, \textit{rest})\}$$

- you consume one or more token from the input (stream)
- `T_NUM(1), T_OP(+), T_NUM(2)`

- a good starting point would be `comb2.scala`
- in case CW2 did not work, use `toks.scala` as input to the parser

JVM Code

Jasmin
Krakatau
ASM lib

```
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
```

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

Fibonacci Numbers

??

Interpreter

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

Interpreter (2)

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

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

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

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

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

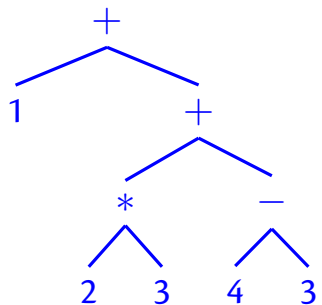
Test Program

??

```
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
```

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 $\Rightarrow$ 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

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

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

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

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

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

$$\mathbf{x} := \mathbf{5} + \mathbf{y} * \mathbf{2}$$

- 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

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

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

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

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

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

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

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

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

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

Compiling Assignments

$x := x + 1$

```
iload n_x
ldc 1
iadd
istore n_x
```

where n_x is the index corresponding to the variable x

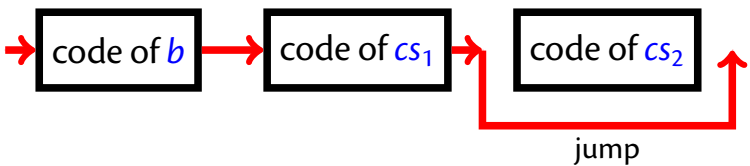
Compiling Ifs

if b then cs_1 else cs_2

Compiling Ifs

if b then cs_1 else cs_2

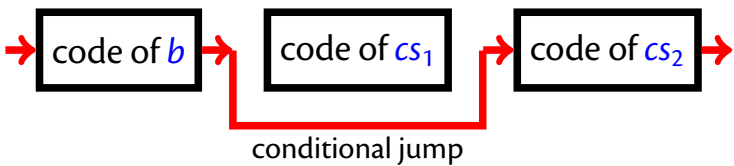
Case **True**:



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:
```

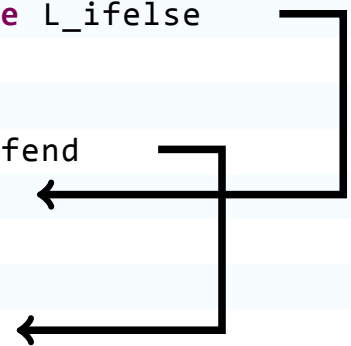
labels must be unique

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:
```



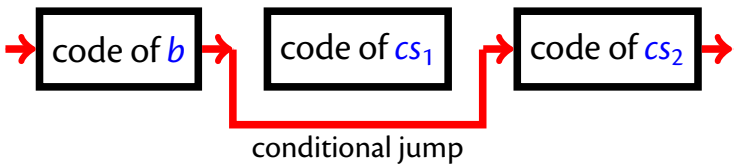
Compiling BExps

$a_1 = a_2$

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

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

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

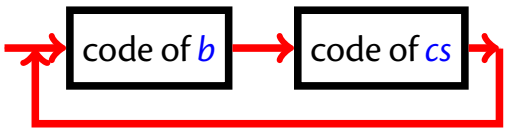
Compiling Whiles

while *b* do *cs*

Compiling Whiles

while *b* do *cs*

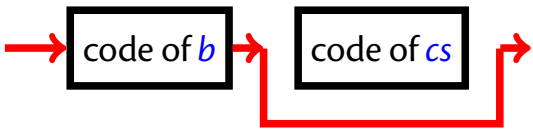
Case **True**:



Compiling Whiles

while *b* do *cs*

Case **False**:



Compiling Whiles

while b do cs

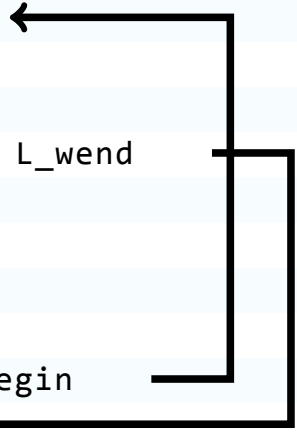
$$
\begin{array}{l}
\text{compile}(\text{while } b \text{ do } cs, E) \stackrel{\text{def}}{=} \\
\quad l_{wbegin} \text{ (fresh label)} \\
\quad l_{wend} \text{ (fresh label)} \\
\quad (is, E') = \text{compile}(cs_1, E) \\
\quad (l_{wbegin} : \\
\quad\quad @\ \text{compile}(b, E, l_{wend}) \\
\quad\quad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

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

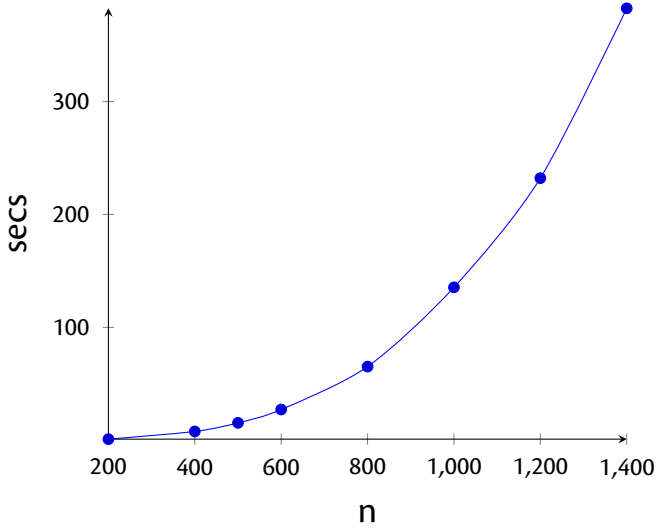
```
.class public XXX.XXX
.super java/lang/Object

.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

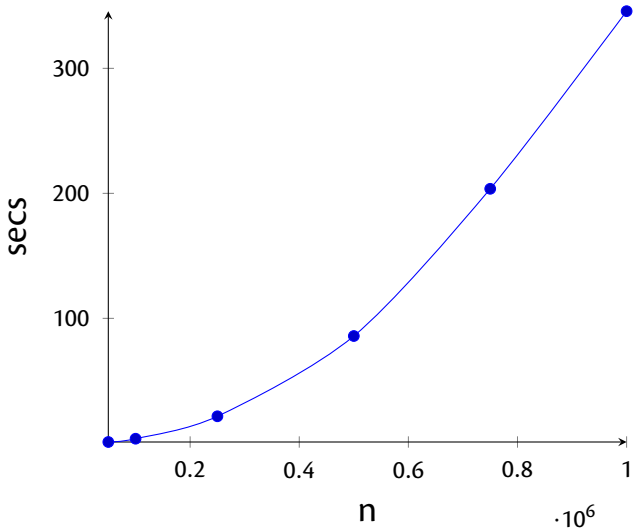
- `javap` is a disassembler for class files
- jasmin and krakatau are assemblers for jvm code

Recall: Interpreted Code

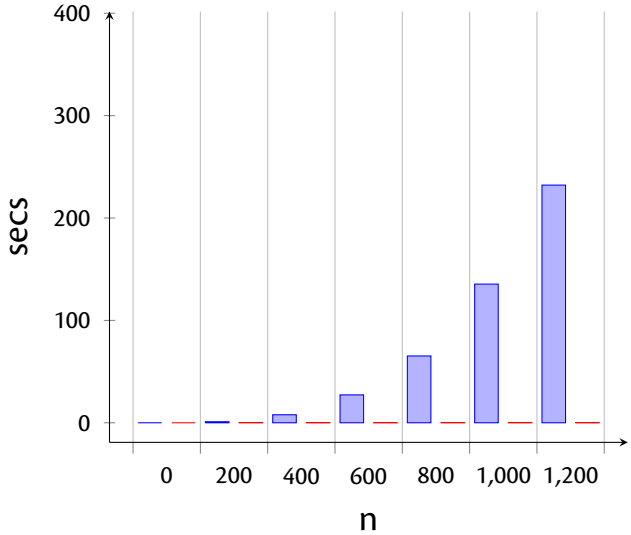


Loop program

Compiled Code



Compiler vs. Interpreter



A "Compiler" for BF*** to C

`>`	⇒	`ptr++`
`<`	⇒	`ptr--`
`+`	⇒	`(*ptr)++`
`-`	⇒	`(*ptr)--`
`.`	⇒	`putchar(*ptr)`
`,`	⇒	`*ptr = getchar()`
`[`	⇒	`while(*ptr){`
`]`	⇒	`}`
	⇒	ignore everything else

```
char field[30000]
char *ptr = &field[15000]
```

BF***

we need some big array, say `arr` and 7 (8) instructions:

- `>` move `ptr++`
- `<` move `ptr--`
- `+` add `arr[ptr]++`
- `-` subtract `arr[ptr]--`
- `.` print out `arr[ptr]` as ASCII
- `[` if `arr[ptr] == 0` jump just after the corresponding `]`; otherwise `ptr++`
- `]` if `arr[ptr] != 0` jump just after the corresponding `[`; otherwise `ptr++`

Arrays in While

- `new arr[15000]`
- `x := 3 + arr[3 + y]`
- `arr[42 * n] := ...`

New Arrays

```
new arr[number]
```

```
ldc number
newarray int
astore loc_var
```

Array Update

```
arr[...] :=
```

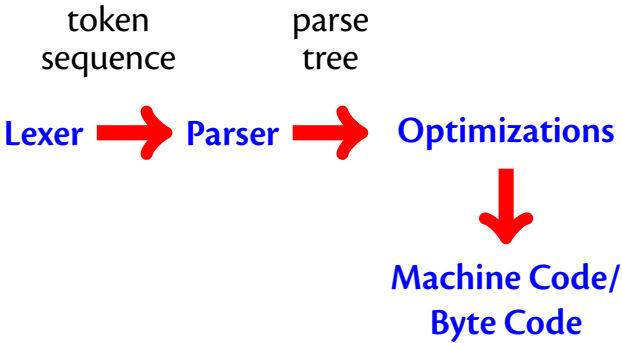
```
aload loc_var
index_aexp
value_aexp
iastore
```

Array Lookup in AExp

```
...arr[...]...
```

```
aload loc_var
index_aexp
iaload
```

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