Automata and Formal Languages (8)

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?

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

Bird's Eye View

Lexer → (token sequence) → **Parser** → (parse tree) → **Machine Code/ Byte Code**


0000000000: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
0000000010: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000030: 00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00
0000000040: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
0000000050: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
0000000060: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
0000000070: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
0000000080: EB 00 42 37 AF 61 2C 64 AF 61 2C 64 AF 61 2C 64
0000000090: D4 7D 20 64 AD 61 2C 64 C0 7E 27 64 AC 61 2C 64
00000000A0: 2C 7D 22 64 A7 61 2C 64 C0 7E 26 64 A4 61 2C 64
00000000B0: C0 7E 28 64 AD 61 2C 64 21 69 73 64 AE 61 2C 64
00000000C0: AF 61 2D 64 0E 61 2C 64 2C 69 71 64 A6 61 2C 64
00000000D0: 99 47 27 64 92 61 2C 64 88 A7 51 64 AE 61 2C 64
00000000E0: 99 47 26 64 AC 61 2C 64 68 67 2A 64 AE 61 2C 64
00000000F0: 52 69 63 68 AF 61 2C 64 00 00 00 00 00 00 00 00
0000000100: 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00
0000000110: B3 ED 87 49 00 00 00 00 00 00 00 00 E0 00 0F 01
0000000120: 0B 01 06 00 00 C4 01 00 00 A4 00 00 00 00 00 00
0000000130: 96 A7 01 00 00 10 00 00 00 E0 01 00 00 00 40 00
0000000140: 00 10 00 00 00 02 00 00 04 00 00 00 00 00 00 00
0000000150: 04 00 00 00 00 00 00 00 00 A0 02 00 00 04 00 00
0000000160: 00 00 00 00 03 00 00 00 00 00 10 00 00 10 00 00
0000000170: 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00
0000000180: 00 00 00 00 00 00 00 00 28 2D 02 00 78 00 00 00
0000000190: 00 90 02 00 10 03 00 00 00 00 00 00 00 00 00 00
MZ?
is program canno
t be run in DOS
mode.
Rich
PE

Assembly Code

```
main:   subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sw      $fp, 16($sp)
        addiu   $fp, $sp, 28
        li      $v0, 4
        la      $a0, str
        syscall
        li      $a0, 10
        jal     fact
        addu    $a0, $v0, $zero
        li      $v0, 1
        syscall
        lw      $ra, 20($sp)
        lw      $fp, 16($sp)
        addiu   $sp, $sp, 32
        jr      $ra
```

```
fact:   subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sw      $fp, 16($sp)
        addiu   $fp, $sp, 28
        sw      $a0, 0($fp)
        lw      $v0, 0($fp)
        bgtz    $v0, L2
        li      $v0, 1
        j       L1
L2:     lw      $v1, 0($fp)
        subu    $v0, $v1, 1
        move    $a0, $v0
        jal     fact
        lw      $v1, 0($fp)
        mul     $v0, $v0, $v1
L1:     lw      $ra, 20($sp)
        lw      $fp, 16($sp)
        addiu   $sp, $sp, 32
        jr      $ra
```

Jasmin assembler for Java bytecode

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

Fibonacci Numbers

```
/* Fibonacci Program
   input: n */

write "Fib";
read n;    // n := 19;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

Interpreter

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

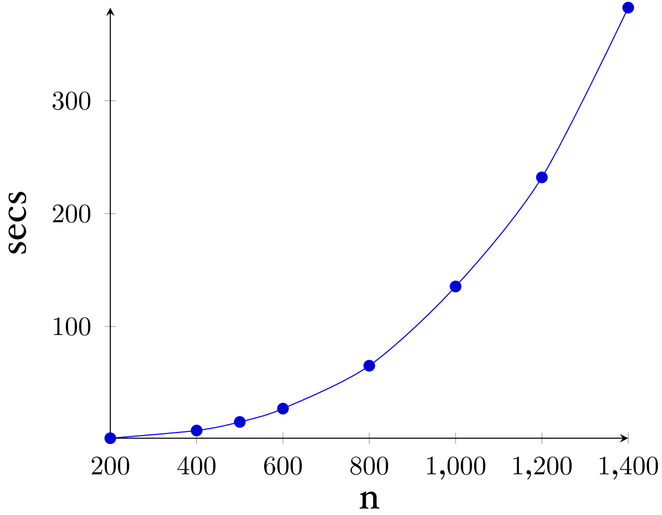
Interpreter (2)

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

Test Program

```
start := 1000;    // start value
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
}
```

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

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

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

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

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

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

Compiling AExps

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

Compiling AExps

1 + 2 * 3 + (4 - 3)

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

Variables

$$\boldsymbol{x} := 5 + \boldsymbol{y} \ast 2$$

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

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

Compiling AExps

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

Compiling Statements

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

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

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

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 index corresponding to the variable x

Compiling Ifs

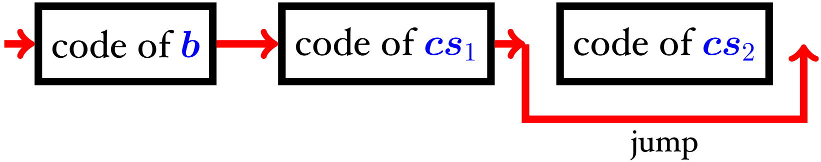
if b then cs_1 else cs_2

code of b	code of cs_1	code of cs_2

Compiling Ifs

if b then cs_1 else cs_2

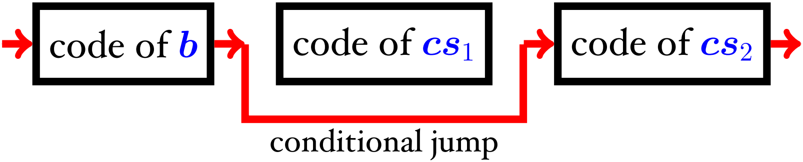
Case **True**:



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 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 BExps

$$a_1 = a_2$$

$$\begin{aligned} \text{compile}(a_1 = a_2, E, lab) &\stackrel{\text{def}}{=} \\ &\text{compile}(a_1, E) \;@\; \text{compile}(a_2, E) \;@\; \text{if\_icmpne}\; lab \end{aligned}$$

Compiling Ifs

if $\boldsymbol{b}$ then $\boldsymbol{cs}_1$ else $\boldsymbol{cs}_2$

$$
\begin{array}{l}
\text{compile}(\text{if } \boldsymbol{b} \text{ then } \boldsymbol{cs}_1 \text{ else } \boldsymbol{cs}_2, \boldsymbol{E}) \stackrel{\text{def}}{=} \\
\quad \boldsymbol{l_{ifelse}} \quad \text{(fresh label)} \\
\quad \boldsymbol{l_{ifend}} \quad \text{(fresh label)} \\
\quad (\boldsymbol{is}_1, \boldsymbol{E'}) = \text{compile}(\boldsymbol{cs}_1, \boldsymbol{E}) \\
\quad (\boldsymbol{is}_2, \boldsymbol{E''}) = \text{compile}(\boldsymbol{cs}_2, \boldsymbol{E'}) \\
\quad (\text{compile}(\boldsymbol{b}, \boldsymbol{E}, \boldsymbol{l_{ifelse}}) \\
\quad\ @\ \boldsymbol{is}_1 \\
\quad\ @\ \text{goto } \boldsymbol{l_{ifend}} \\
\quad\ @\ \boldsymbol{l_{ifelse}} : \\
\quad\ @\ \boldsymbol{is}_2 \\
\quad\ @\ \boldsymbol{l_{ifend}} :, \boldsymbol{E''})
\end{array}
$$

Compiling Whiles

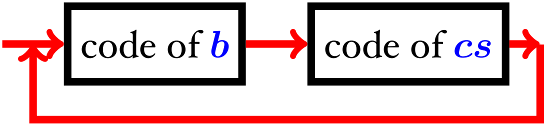
while b do cs

code of b | code of cs

Compiling Whiles

while b do cs

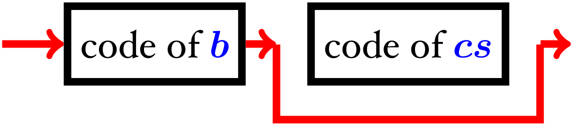
Case **True**:



Compiling Whiles

while b do cs

Case **False**:



Compiling Whiles

while b do cs

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

Compiling Writes

write x

```
.method public static write(I)V        (library function)
  .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

```
.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)

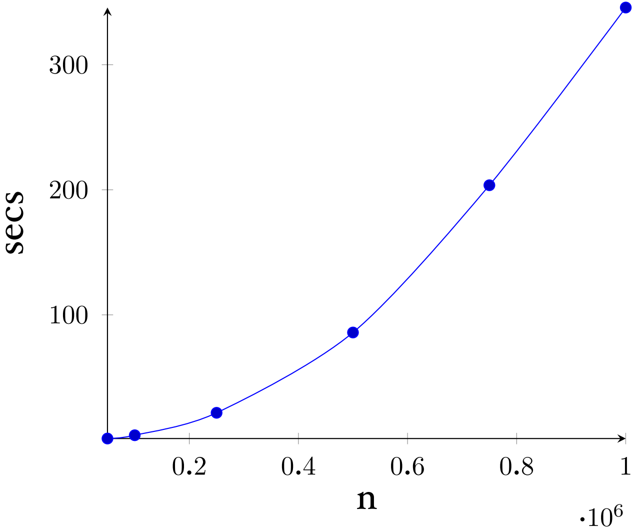
```
  return
.end method
```

Next Compiler Phases

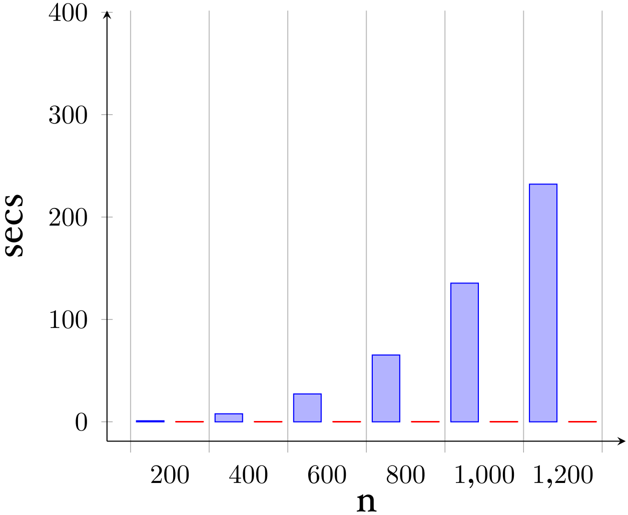
- assembly ⇒ byte code (class file)
- labels ⇒ absolute or relative jumps

- `javap` is a disassembler for class files

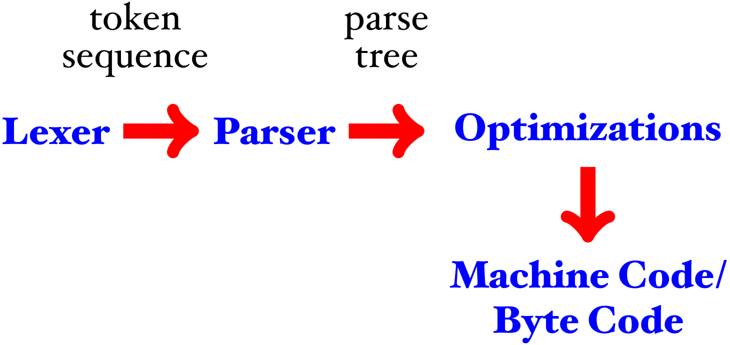
Compiled Code



Compiler vs. Interpreter



Backend


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

What Next

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

- GCC, LLVM, tracing JITs