

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)

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

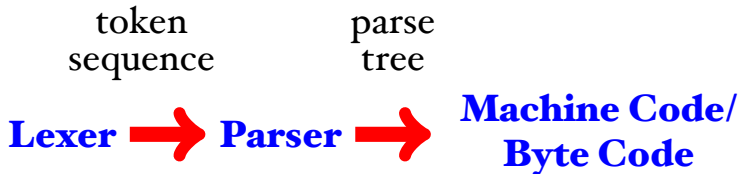
What is your reaction?

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



000000000:	4D 5A 90 00 03 00 00 00	04 00 00 00 FF FF 00 00	MZ7 ♥ ♦ ♣ ♠
000000010:	B8 00 00 00 00 00 00 00	40 00 00 00 00 00 00 00	• e
000000020:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
000000030:	00 00 00 00 00 00 00 00	00 00 00 00 00 01 00 00	☐
000000040:	0E 1F BA 0E 00 B4 09 CD	21 B8 01 4C CD 21 54 68	ITP=H 'GIT, GLI!Th
000000050:	69 73 20 70 72 6F 67 72	61 6D 20 63 61 6E 6E 6F	is program canno
000000060:	74 20 62 65 20 72 75 6E	20 69 6E 20 44 4F 53 20	t be run in DOS
000000070:	6D 6F 64 65 2E 0D 0D 00	24 00 00 00 00 00 00 00	mode.FJG\$
000000080:	EB 00 42 37 AF 61 2C 64	AF 61 2C 64 AF 61 2C 64	è B7_a,d_a,d_a,d
000000090:	D4 7D 20 64 AD 61 2C 64	C0 7E 27 64 AC 61 2C 64	O) d-a,d0^'d-a,d
0000000A0:	2C 7D 22 64 A7 61 2C 64	C0 7E 26 64 A4 61 2C 64	>"d@a,d0^'d@a,d
0000000B0:	C0 7E 28 64 AD 61 2C 64	21 69 73 64 AE 61 2C 64	0^'(d-a,d!isdra,d
0000000C0:	AF 61 2D 64 0E 61 2C 64	2C 69 71 64 06 61 2C 64	_a-d/a,d,iqd a,d
0000000D0:	99 47 27 64 92 61 2C 64	88 A7 51 64 AE 61 2C 64	TG'd'a,d^EQdra,d
0000000E0:	99 47 26 64 AC 61 2C 64	68 67 20 64 AE 61 2C 64	TG@d'a,dhgydra,d
0000000F0:	52 69 63 68 AF 61 2C 64	00 00 00 00 00 00 00 00	Rich_a,d
000000100:	00 00 00 00 00 00 00 00	50 45 00 00 4C 01 04 00	PE LG*
000000110:	B3 ED 87 49 00 00 00 00	00 00 00 00 E0 00 0F 01	3i+I à *Q
000000120:	00 01 06 00 00 C4 01 00	00 04 00 00 00 00 00 00	80* A0 * A0 e
000000130:	96 A7 01 00 00 10 00 00	00 E0 01 00 00 00 40 00	-A0 A0 e
000000140:	00 10 00 00 00 02 00 00	04 00 00 00 00 00 00 00	▶ ◀ ♦
000000150:	04 00 00 00 00 00 00 00	00 00 02 00 00 04 00 00	♦ ◀ ◀ ◀
000000160:	00 00 00 00 03 00 00 00	00 00 10 00 00 10 00 00	▶ ▶ ▶ ▶
000000170:	00 00 10 00 00 10 00 00	00 00 00 00 10 00 00 00	▶ ▶ ▶ ▶
000000180:	00 00 00 00 00 00 00 00	28 2D 02 00 78 00 00 00	<-0 x
000000190:	00 90 02 00 10 03 00 00	00 00 00 00 00 00 00 00	?0 ▶
0000001A0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	

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
```

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  /* Fibonacci numbers implemented in
2     the WHILE language */
3
4  write "Input a number ";
5  read n;
6  x := 0;    // start values
7  y := 1;
8  while n > 0 do {
9     temp := y;
10    y := x + y;
11    x := temp;
12    n := n - 1 // decrement counter
13 };
14 write "Result ";
15 write y
```


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)

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

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

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

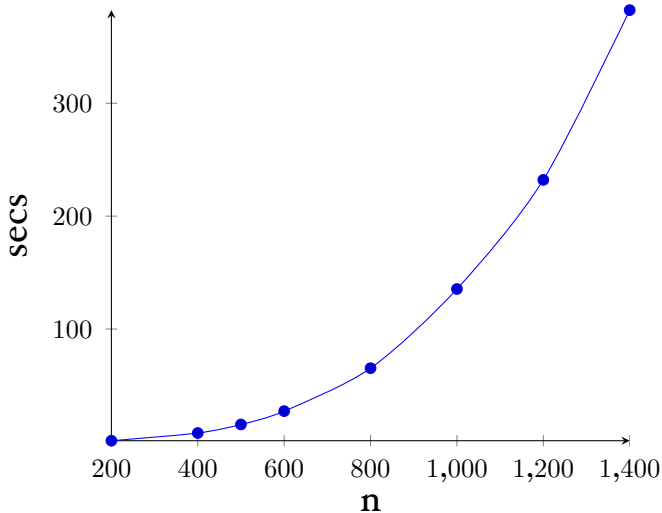
$$\begin{aligned} \text{eval}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=} \\ &\text{if } \text{eval}(b, E) \\ &\text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\ &\text{else } E \end{aligned}$$

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

Test Program

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

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

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

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

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

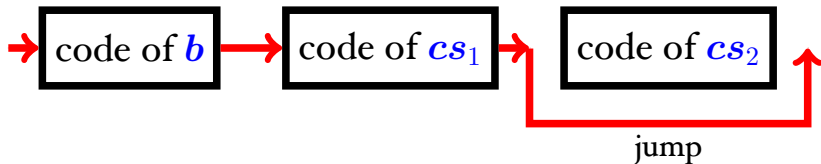
code of cs_1

code of cs_2

Compiling Ifs

if b then cs_1 else cs_2

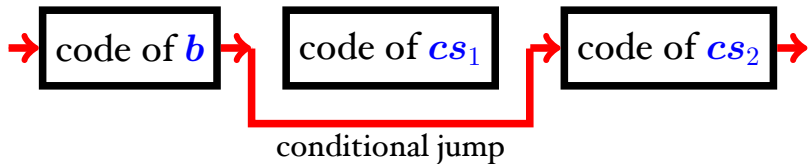
Case **True**:



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 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 r  
    ldc r  
    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

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

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

@ goto l_{ifend}

@ l_{ifelse} :

@ is_2

@ l_{ifend} :, E'')

Compiling Whiles

while *b* do *cs*

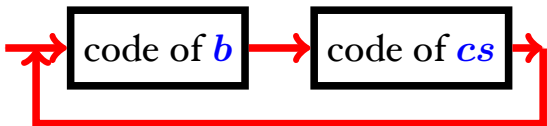
code of *b*

code of *cs*

Compiling Whiles

while *b* do *cs*

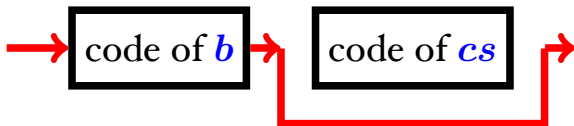
Case **True**:



Compiling Whiles

while *b* do *cs*

Case **False**:



Compiling Whiles

while b do cs

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

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)

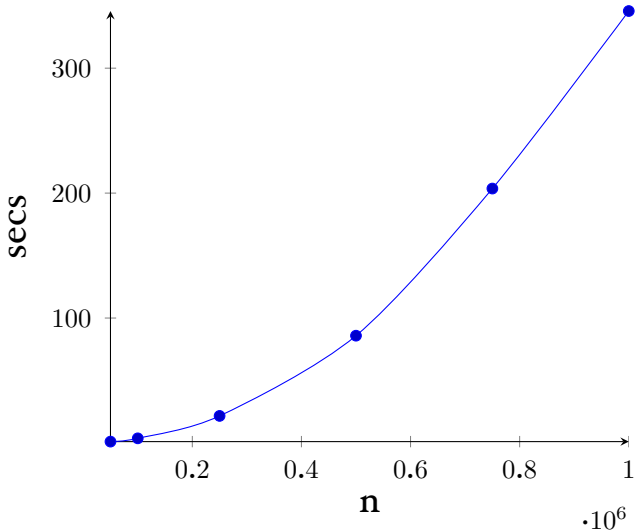
```
  return  
.end method
```

Next Compiler Phases

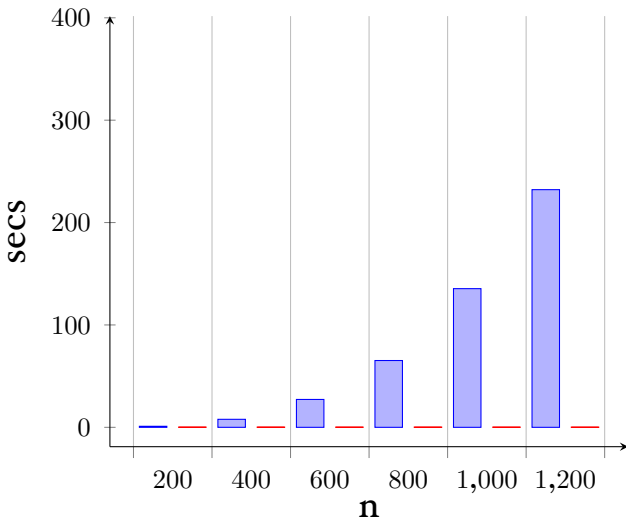
- assembly \Rightarrow byte code (class file)
- labels \Rightarrow absolute or relative jumps

- javap is a disassembler for class files

Compiled Code



Compiler vs. Interpreter



What Next

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

- GCC, LLVM, tracing JITs