# Compilers and Formal Languages (8)

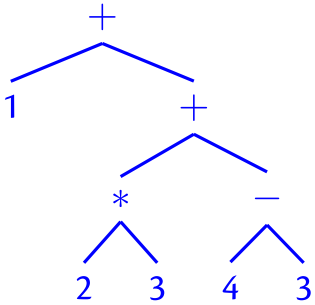

Email: christian.urban at kcl.ac.uk
Office: N7.07 (North Wing, Bush House)

Slides: KEATS (also home work is there)

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

# 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

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

# 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 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
```

# Functional Programming

```
def fib(n) = if n == 0 then 0
             else if n == 1 then 1
             else fib(n - 1) + fib(n - 2);

def fact(n) = if n == 0 then 1 else n * fact(n - 1);

def ack(m, n) = if m == 0 then n + 1
                else if n == 0 then ack(m - 1, 1)
                else ack(m - 1, ack(m, n - 1));

def gcd(a, b) = if b == 0 then a else gcd(b, a % b);
```

# Fun Grammar

$$
\begin{array}{lcl}
\textit{Exp} & ::= & \textit{Var} \mid \textit{Num} \mid \textit{Exp} + \textit{Exp} \\
 & \mid & \ldots \mid (\textit{Exp}) \mid \texttt{if}\ \textit{BExp}\ \texttt{then}\ \textit{Exp}\ \texttt{else}\ \textit{Exp} \\
 & \mid & \texttt{write}\ \textit{Exp} \mid \textit{Exp}\,;\,\textit{Exp} \\
 & \mid & \textit{FunName}\,(\textit{Exp}, \ldots, \textit{Exp}) \\
\textit{BExp} & ::= & \ldots \\
\textit{Def} & ::= & \texttt{def}\ \textit{FunName}\,(x_1, \ldots, x_n) = \textit{Exp} \\
\textit{Prog} & ::= & \textit{Def}\,;\,\textit{Prog} \mid \textit{Exp}\,;\,\textit{Prog} \mid \textit{Exp}
\end{array}
$$

# Abstract Syntax Trees

```scala
abstract class Exp
abstract class BExp
abstract class Decl

case class Var(s: String) extends Exp
case class Num(i: Int) extends Exp
case class Aop(o: String, a1: Exp, a2: Exp) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
case class Sequ(e1: Exp, e2: Exp) extends Exp
case class Call(name: String, args: List[Exp]) extends Exp

case class Bop(o: String, a1: Exp, a2: Exp) extends BExp

case class Def(name: String,
               args: List[String],
               body: Exp) extends Decl
case class Main(e: Exp) extends Decl
```

# Idea

Compile exp such that the result of the expression is on top of the stack.

- `write(1 + 2)`
- `1 + 2; 3 + 4`

# Sequences

Compiling `exp1 ; exp2`:

```
compile(exp1)
pop
compile(exp2)
```

# Write

Compiling call to `write(1+2)`:

```
compile(1+2)
dup
invokestatic XXX/XXX/write(I)V
```

needs the helper method

```
.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
```

# Function Definitions

```
.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
```

We will need for definitions, like

```
def fname (x1, ... , xn) = ...
```

```
.method public static fname (I...I)I
  .limit locals ??
  .limit stack ??
  ??
.end method
```

# Stack Estimation

$$
\begin{array}{lcl}
estimate(n) & \stackrel{\text{def}}{=} & 1\\
estimate(x) & \stackrel{\text{def}}{=} & 1\\
estimate(a_1\ aop\ a_2) & \stackrel{\text{def}}{=} & estimate(a_1) + estimate(a_2)\\
estimate(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & \stackrel{\text{def}}{=} & estimate(b) +\\
 & & \quad max(estimate(e_1), estimate(e_2))\\
estimate(\texttt{write}(e)) & \stackrel{\text{def}}{=} & estimate(e) + 1\\
estimate(e_1; e_2) & \stackrel{\text{def}}{=} & max(estimate(e_1), estimate(e_2))\\
estimate(f(e_1, ..., e_n)) & \stackrel{\text{def}}{=} & \sum_{i=1..n}\ estimate(e_i)\\
estimate(a_1\ bop\ a_2) & \stackrel{\text{def}}{=} & estimate(a_1) + estimate(a_2)
\end{array}
$$

# Successor Function

```
.method public static suc(I)I
.limit locals 1
.limit stack 2
  iload 0
  ldc 1
  iadd
  ireturn
.end method
```

```
def suc(x) = x + 1;
```

# Addition Function

```
.method public static add(II)I
.limit locals 2
.limit stack 5
  iload 0
  ldc 0
  if_icmpne If_else
  iload 1
  goto If_end
If_else:
  iload 0
  ldc 1
  isub
  iload 1
  invokestatic XXX/XXX/add(II)I
  invokestatic XXX/XXX/suc(I)I
If_end:
  ireturn
.end method
```

```
def add(x, y) =
    if x == 0 then y
    else suc(add(x - 1, y));
```

# Factorial

```
.method public static facT(II)I
.limit locals 2
.limit stack 6
  iload 0
  ldc 0^^I
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  invokestatic fact/fact/facT(II)I
If_end_3:
  ireturn
.end method
```

```
def facT(n, acc) =
  if n == 0 then acc
  else facT(n - 1, n * acc);
```

```
.method public static facT(II)I
.limit locals 2
.limit stack 6
facT_Start:
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  istore 1
  istore 0
  goto facT_Start
If_end_3:
  ireturn
```

```
def facT(n, acc) =
  if n == 0 then acc
  else facT(n - 1, n * acc);
```

# Tail Recursion

A call to `f(args)` is usually compiled as

```
args onto stack
invokestatic  .../f
```

# Tail Recursion

A call to `f(args)` is usually compiled as

```
args onto stack
invokestatic  .../f
```

A call is in tail position provided:

- `if Bexp then` `Exp` `else` `Exp`
- `Exp ;` `Exp`
- `Exp op Exp`

then a call `f(args)` can be compiled as

```
prepare environment
jump to start of function
```

# Tail Recursive Call

```
def compile_expT(a: Exp, env: Mem, name: String): Instrs =
  ...
  case Call(n, args) => if (name == n)
  {
    val stores = args.zipWithIndex.map
       { case (x, y) => "istore " + y.toString + "\n" }
    args.flatMap(a => compile_expT(a, env, "")) ++
    stores.reverse ++
    List ("goto " + n + "_Start\n")
  }
  else
  {
    val is = "I" * args.length
    args.flatMap(a => compile_expT(a, env, "")) ++
    List ("invokestatic XXX/XXX/" + n + "(" + is + ")I\n")
  }
```

# Dijkstra on Testing

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

What is good about compilers: the either seem to work, or go horribly wrong (most of the time).

# Proving Programs to be Correct

**Theorem:** There are infinitely many prime numbers.

**Proof** ...

similarly

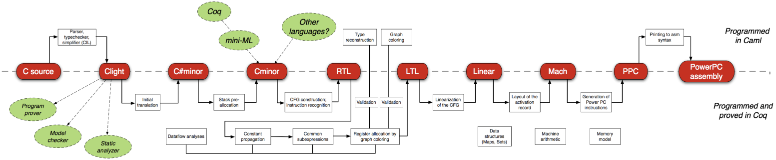
**Theorem:** The program is doing what it is supposed to be doing.

**Long, long proof** ...

This can be a gigantic proof. The only hope is to have help from the computer. 'Program' is here to be understood to be quite general (compiler, OS, ...).

# Can This Be Done?

- in 2008, verification of a small C-compiler
  - "if my input program has a certain behaviour, then the compiled machine code has the same behaviour"
  - is as good as `gcc -O1`, but much, much less buggy



# Fuzzy Testing C-Compilers

- tested GCC, LLVM and others by randomly generating C-programs
- found more than 300 bugs in GCC and also many in LLVM (some of them highest-level critical)

- about CompCert:

> "The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."