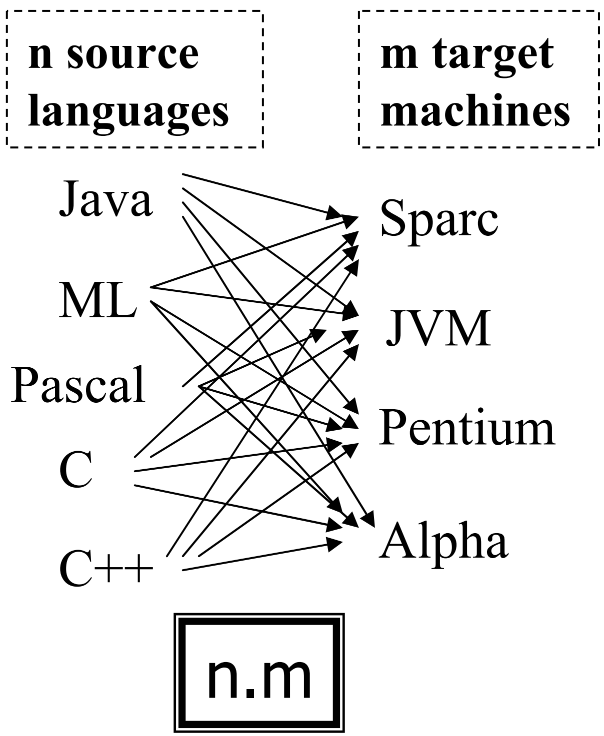
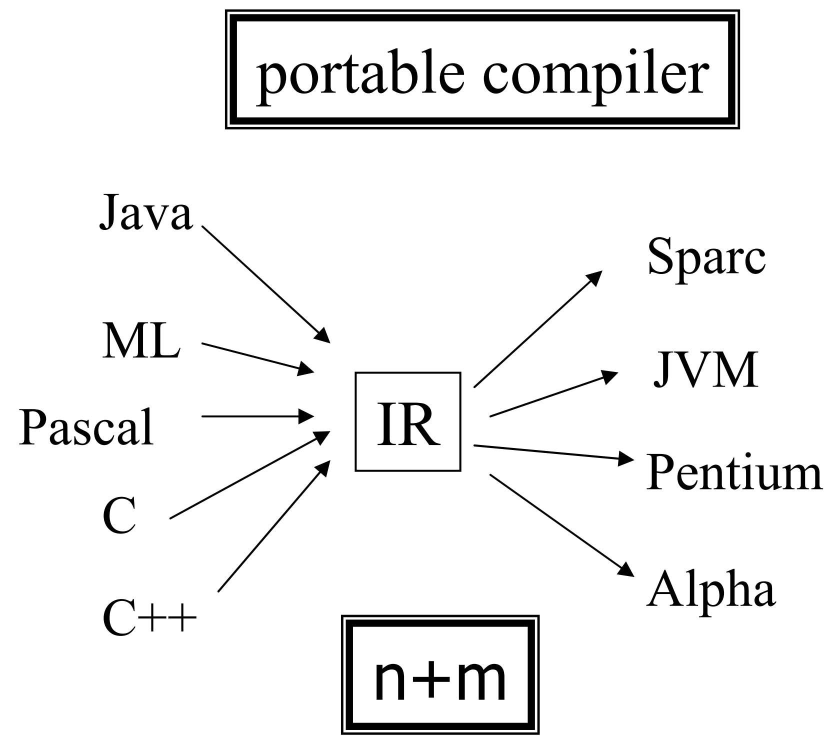
Session Plan

- Session 7: ***Translation to intermediate representation***
  - intermediate representation
    - » why?
    - » definition of an IR using trees
  - example translations
  - see book for while-loops, for-loops, functions, declarations

Intermediate representation (1)

- semantic analysis:
  - converts abstract syntax to abstract machine code (an intermediate rep)
- why an intermediate representation…?





IR: intermediate representation (2)

- IR: an abstract machine language
  - express target machine operations, but without machine-specific details
  - source-language independent
- compiler
  - front end: lexical analysis, parsing, semantic analysis
  - back end:
    - » optimisation of IR (rewrite IR so as to improve execution speed)
    - » translation to real machine language
    - » (in case of Java, to *another* abstract machine language JVM)
- many IRs
  - Appel uses simple expression trees

IR: intermediate representations

- any good representation
  - must be convenient for semantic analysis phase to produce
  - must be convenient to translate to real (or virtual) machine language for target machines
  - must have simple meaning for each construct that leads to simple operations on the IR to rewrite parts of it for optimisation etc
- in any IR
  - individual components describe simple operations on the abstract machine represented by the IR instructions
  - each element of the complex abstract syntax is translated into a set of simple IR abstract machine instructions
  - groups of IR instructions will be grouped and regrouped to form real machine instructions

What does the IR abstract machine have?

- ie what do the trees represent/operate on?

What does the IR abstract machine have?

- ie what do the trees represent/operate on?
  - see lecture discussion

integer constants

memory

registers [temporaries in the translation - infinite number in IR]

instruction set

sequential execution

labels and jumps

IR: tree expression operators

CONST(i)	integer constant
NAME(n)	symbolic constant (an assembly lang label)
TEMP(t)	abstract register...infinite number!!
BINOP(o,e_1,e_2)	PLUS, MINUS, MUL, DIV, AND, OR, XOR, LSHIFT, RSHIFT, ARSHIFT
MEM(e)	contents of *wordSize* bytes starting at addr e (means “store” if left child of move, else “fetch”)
CALL(f,l)	procedure call, applies f to list l
ESEQ(s,e)	eval s for side-effects, eval e for result

IR: statements

perform side effects
& control flow

MOVE(TEMP t,e)	eval e & move into temp t
MOVE(MEM(e_1,k),e_2)	eval e_1 ($\Rightarrow$ addr a); eval e_2 & store results into k bytes of mem starting at a
EXP(e)	eval e & discard the result
JUMP(e,labs)	transfer control to addr e; labs specifies list of all poss locations e can eval to
CJUMP(o,e_1,e_2,t,f)	eval e_1 then e_2, compare result with relational op o; if true jump to t else to f
SEQ(s_1,s_2)	???
LABEL(n)	defines const n to be current machine code address

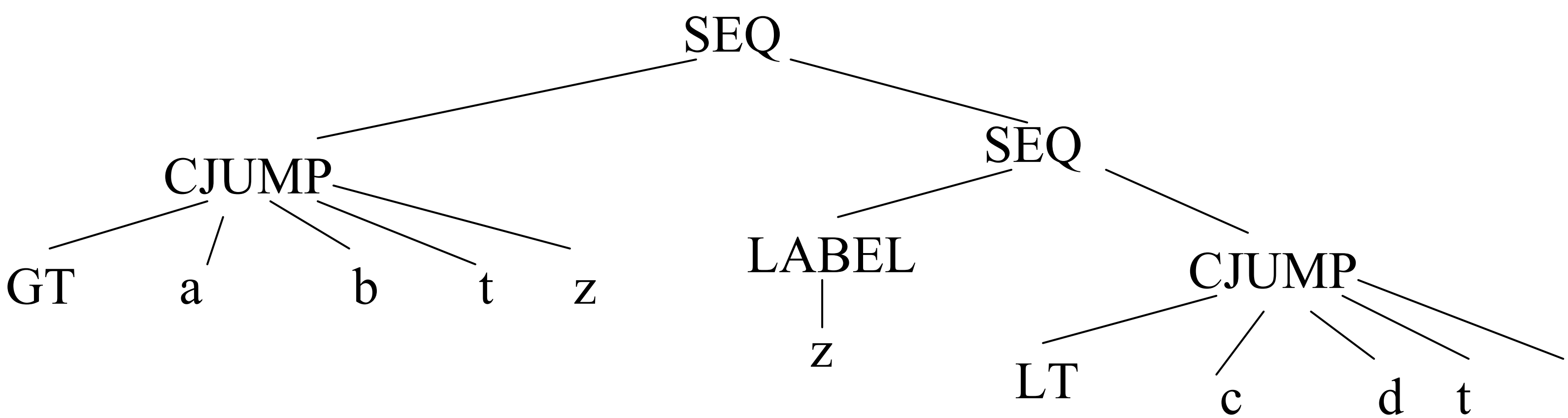
No provision for procedure and & function defs - just body of each function

Example translation

a > b | c < d *translates to*

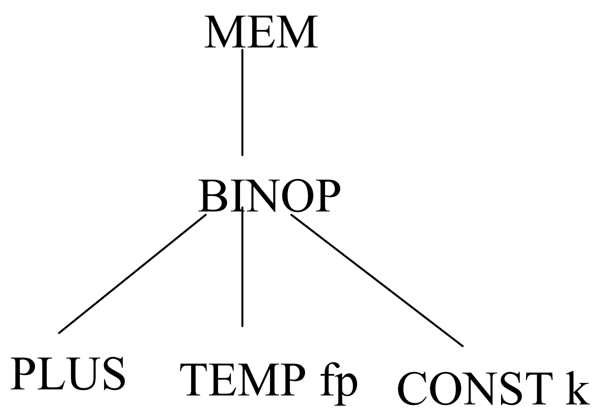
SEQ(CJUMP(GT,a,b,t,z),

SEQ(LABEL z,CJUMP(LT,c,d,t,f))), *with* t,f *labels*



Simple variables translation

- Simple variable *v* in current procedure or function stack frame
  - k: offset of *v* in frame
  - TEMP fp: frame pointer register

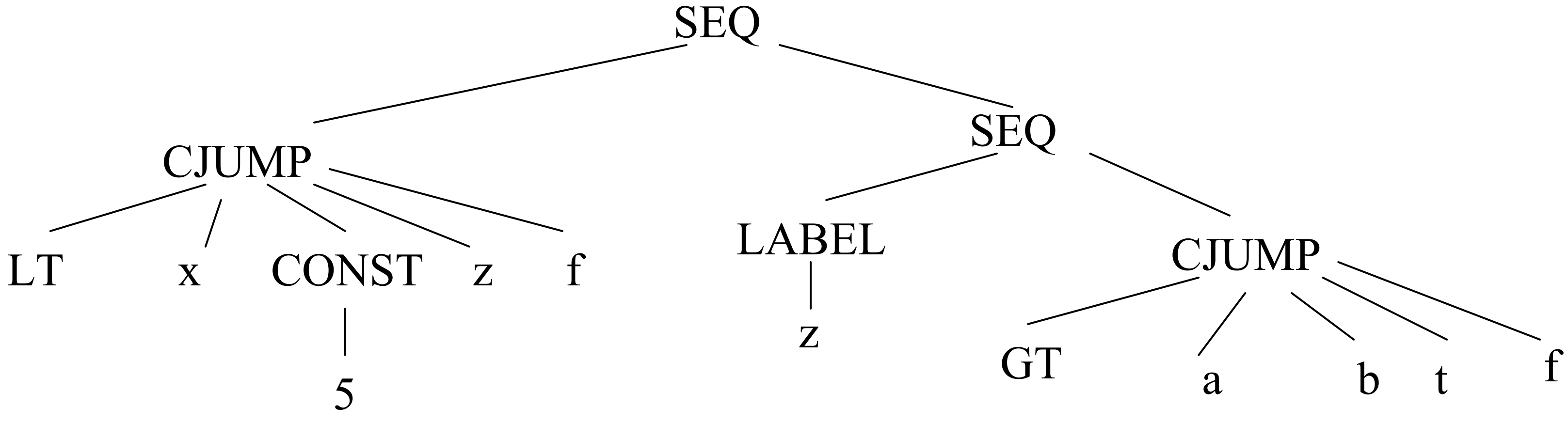


MEM(BINOP(PLUS,TEMP fp, CONST k))

Conditionals

- Use CJUMP
  - x < 5 translates to CJUMP(LT, x, CONST(5), t, f)
  - for labels t, f
- if x < 5 then a > b else 0

SEQ(CJUMP(LT,x,CONST(5),z,f),
 SEQ(LABEL z,
  CJUMP(GT,a,b,t [pick up val of a>b],f [pick up val 0])))



What you should do now

- See book for other translations
  - while-loops
  - for-loops
  - functions
  - declarations