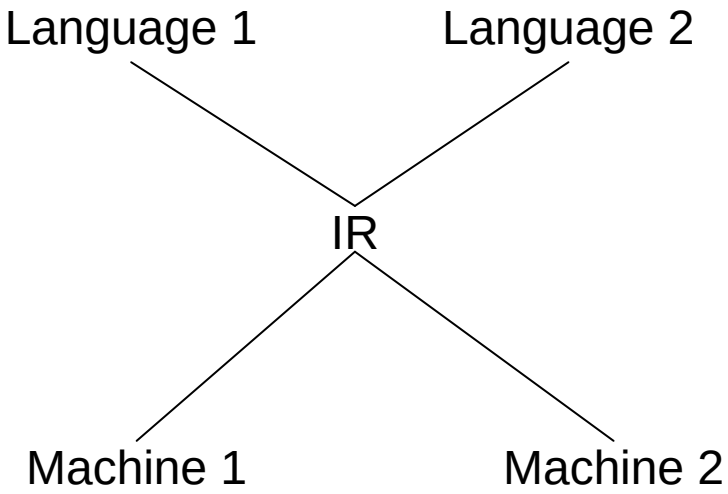
INTERMEDIATE CODE GENERATION
Intermediate representation trees

Why use intermediate representation?

- Allows different source languages and target architectures:



Why use intermediate representation? (contd.)

- Separating front end and back end simplifies compiler
- Allows optimization

How to design intermediate representation?

- Independent of specific language
- Independent of specific architecture
- Lower level than source language

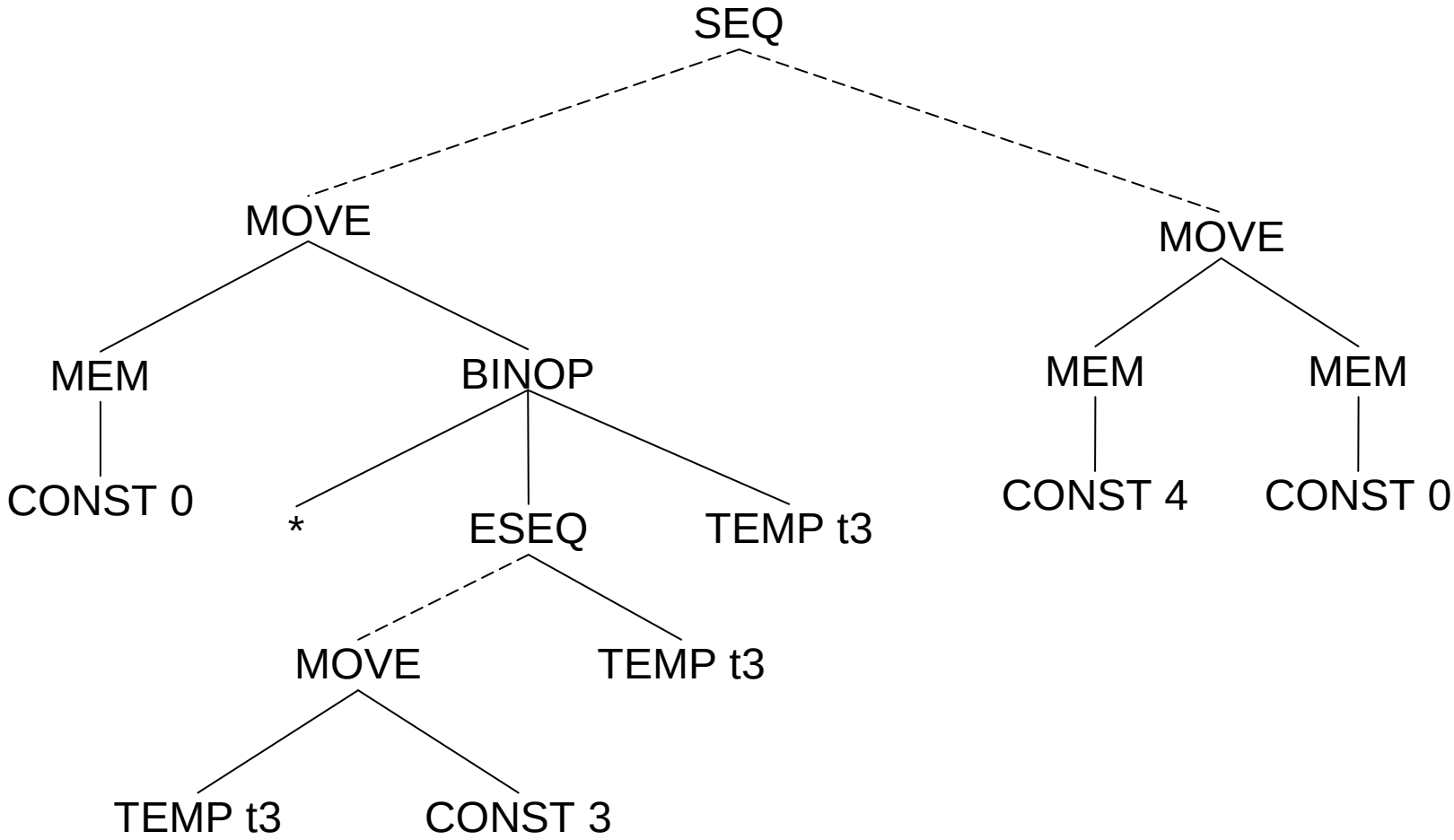
Tree representation

Similar to abstract syntax tree, except:

- Specifies variable location instead of variable:
  - temporary
  - *or* memory location
    - static
    - on stack
- Specifies types of operations
- Specifies size of data
- Control flow done by explicit jumps

Statement trees and expression trees

IR tree contains statement subtrees and expression subtrees.
Expression node may have a statement subtree, and vice versa.



Expression trees:

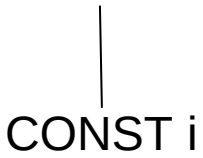
- Each node produces a value
- Value could be stored in a register, or hidden inside an instruction
- Order of evaluation is not significant

Expressions in tree representation

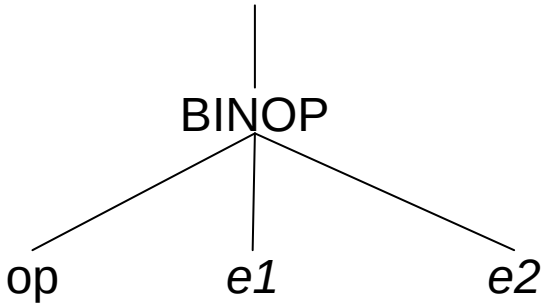
Expressions correspond to

- expressions in source program
- addresses in intermediate code

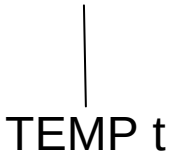
Constant `i`:



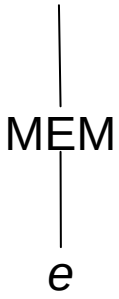
Expression *e1* `op` *e2*:



Temporary variable `t`:

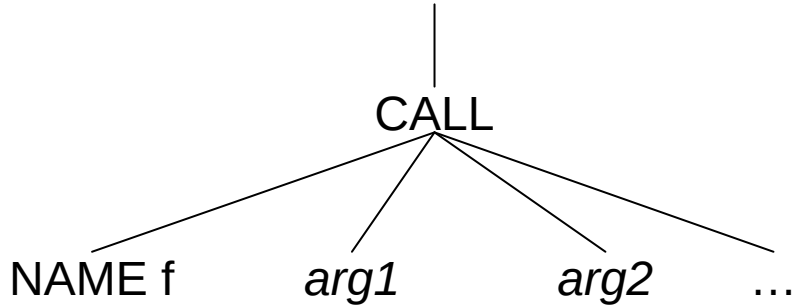


Variable in memory location *e*:

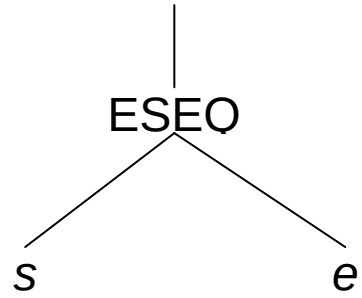


Symbolic constant (address) `n`:

Function call `f(`*`arg1, arg2,`* …`)`:



Expression *e* after executing statement *s*:



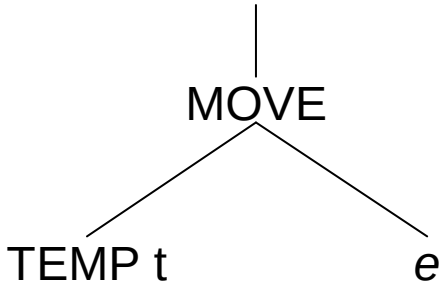
Statements in tree representation

Statements correspond to

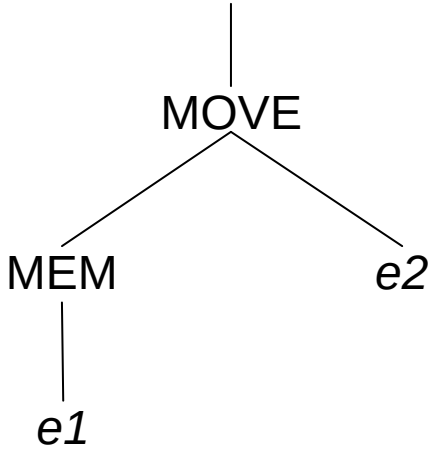
- statements in source program
- jumps and addresses in intermediate code

Evaluate and discard expression *e*:

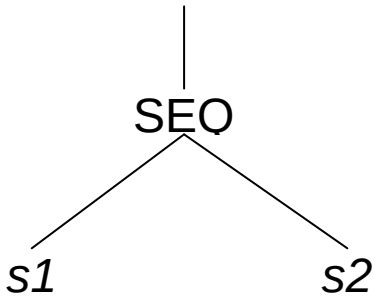
Assignment to temporary variable:



Assignment to variable in memory:

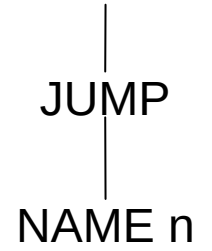


Sequence: statement *s1* followed by *s2*:



Address in intermediate code (jump destination):

Unconditional jump:



Conditional jump: `if (`*e1* `op` *e2*`) goto t else goto f`: