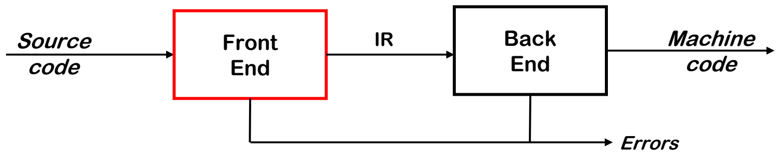
Intermediate representation



The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler
  - > Previous slides described a formal model of semantic analysis
  - > Now look at various models of building IR

Intermediate representations

Advantages
- compiler can make multiple passes over program
- break the compiler into manageable pieces
- support multiple languages and architectures
- using multiple front & back ends
- enables machine-independent optimization

Desirable properties
- easy & inexpensive to generate and manipulate
- contains sufficient information

Examples
- abstract syntax tree (AST)
- directed acyclic graph (DAG)
- control flow graph (CFG)
- three address code
- stack code

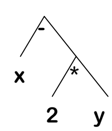
Intermediate representations

Broadly speaking, IRs fall into three categories:

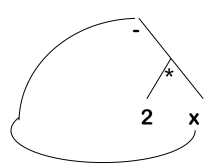
- Structural
  - structural IRs are graphically oriented
  - examples: trees, directed acyclic graphs
  - heavily used in source to source translators
  - nodes, edges tend to be large
- Linear
  - pseudo-code for some abstract machine
  - large variation in level of abstraction
  - simple, compact data structures
  - easier to rearrange
- Hybrids
  - combination of graphs and linear code
  - attempt to take best of each
  - examples: control-flow graph

Example IRs

- An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed. (Already discussed) For example: x-2*y ➔



- For ease of manipulation, can use a linearized (operator) form of the tree. x 2 y * - in postfix form.
- A Directed acyclic graph (DAG) is an AST with a unique node for each value. E.g., x-2*x ➔



- Control flow graph (CFG) – Standard flowchart of program

Three address code

- Three address code generally allow statements of the form with a single operator and, at most, three names :
  x = y op z
- Complex expressions like X-2*y
  are simplified to
  t1 = 2 * y
  t2 = x – t1
- Advantages
  - compact form (direct naming)
  - names for intermediate values
- Register transfer language (RTL)
  - only load/store instructions access memory, all other operands are registers
  - version of three address code for RISC
  - Typical statement types
    - assignments --- x = y op z, x = op y, x = y[i]
    - branches --- goto L, if x relop y goto L
    - procedure calls --- param x, call p
    - address and pointer assignments
- Can represent three address code using *quadruples:* x-2*y. Directly executes as RISC code.

1.	Load	t1	y	
2.	Loadi	t2	2	
3.	Mult	t3	t2	t1
4.	Load	t4	x	
5.	Sub	t5	t4	t2

Stack machine code

- Can simplify IR by assuming implicit stack: z = x - 2 * y becomes
  push z
  push x
  push 2
  push y
  multiply
  subtract
  store
  (What is different about multiply and subtract operator and store operator?)
- Advantages
  - compact form
  - introduced names are implicit, not explicit
  - simple to generate and execute code
- Disadvantages
  - processors operate on registers, not stacks
  - difficult to reuse values on stack

Intermediate representations

- But this isn't the whole story
- Symbol table:
  - identifiers, procedures
  - size, type, location
  - lexical nesting depth
- Constant table:
  - representation, type
  - storage class, offset(s)
- Storage map:
  - storage layout
  - overlap information
  - (virtual) register assignments