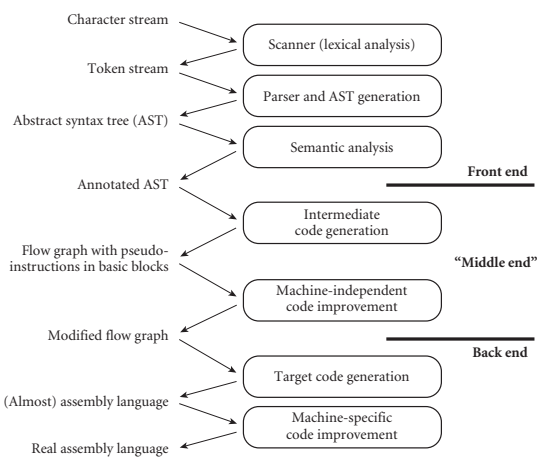
Code generation (intermediate and target)
23 Sept 2020

==========

Reprise compiler phases:




Note that this differs slightly from the version shown in the Chapter 1 lectures. Specifically, I've

- rolled AST generation into the parser
- separated semantic analysis from (medium level) IF generation
- put that IF generation in the "middle end"

This more accurately reflects the likely structure of a modern compiler.

It's common for a compiler to have more than one intermediate form/representation (IF/IR). These are sometimes differentiated by "level," or degree of abstractness:

high-level
  typically an AST
medium-level
  often a ***control flow graph***
    ***basic blocks*** as nodes
    jumps as edges
low-level
  usually instructions for an idealized machine
  perhaps the same notation that's used w/in basic blocks above

NB: there are no hard boundaries between these levels.

One family of IFs deserves separate mention: ***stack-based IFs***
  may be medium or low-level
  not used in most compilers, but important in special cases
    particularly where size is an issue
  examples include JBC, CIL, 1970s pcode

  example from the book: Heron's formula:

```
A = sqrt [s(s–a)(s–b)(s–c)]
where s = (a+b+c)/2
```

```
stack–based:        3–address pseudo–assembly
   push a              r2 := a
   push b              r3 := b
   push c              r4 := c
   add                 r1 := r2 + r3
   add                 r1 := r1 + r4
   push 2              r1 := r1 / 2       –– s
   divide
   pop s
   push s
   push s              r2 := r1 – r2      –– s–a
   push a
   subtract
   push s              r3 := r1 – r3      –– s–b
   push b
   subtract
   push s              r4 := r1 – r4      –– s–c
   push c
   subtract
   multiply            r3 := r3 * r4
   multiply            r2 := r2 * r3
   multiply            r1 := r1 * r2
   push sqrt           call sqrt
   call
```

  time-space tradeoff
    stack code is denser
      lots of instructions, but tiny
    v. speed
      can't optimize for register set and pipeline performance

    The JBC or CIL version of the stack-based code will use a single byte for every instruction except the second-to-last, which will take 3 bytes. That's 23 instructions in 25 bytes.

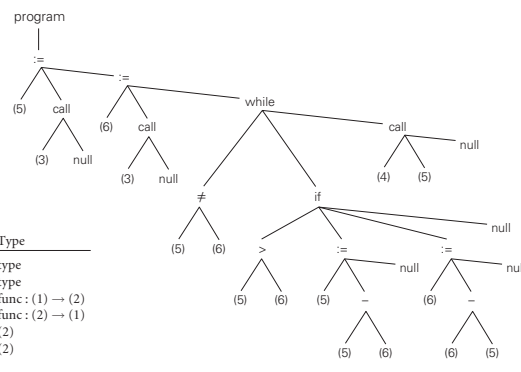
    The 3-address code keeps a, b, c, and s in registers, and uses only 13 instructions. Typically, however, most will be 4 bytes long (the last will be 8). That's 13 instructions in 56 bytes.

---

Consider the GCD example from the Chap. 1 of the book.
Source (in C):

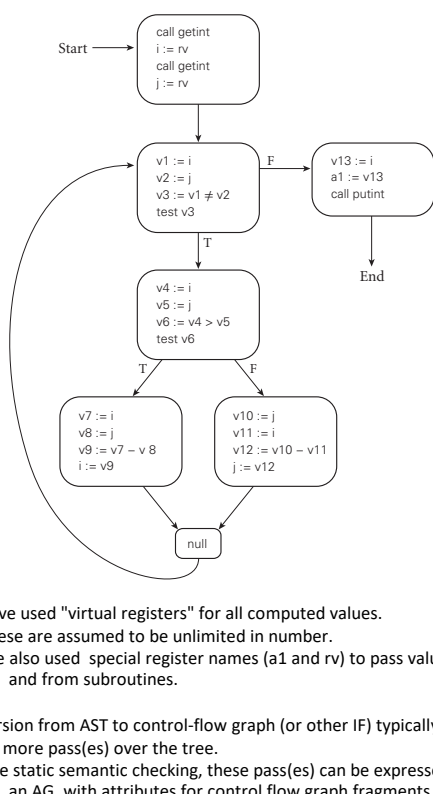
```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
      if (i > j) i = i – j;
      else j = j – i;
   }
   putint(i);
}
```

AST (we know how to generate this now):




| Index | Symbol | Type |
|---|---|---|
| 1 | void | type |
| 2 | int | type |
| 3 | getint | func : (1) → (2) |
| 4 | putint | func : (2) → (1) |
| 5 | i | (2) |
| 6 | j | (2) |

Control Flow Graph is "straightforward" to generate from the AST:




Here I've used "virtual registers" for all computed values.
  These are assumed to be unlimited in number.
  I've also used special register names (a1 and rv) to pass values to and from subroutines.

Conversion from AST to control-flow graph (or other IF) typically uses one or more pass(es) over the tree.
  Like static semantic checking, these pass(es) can be expressed with an AG, with attributes for control flow graph fragments.
  More commonly, it's just hand-written code.

The control-flow graph may see many changes during code improvement. We may split and merge basic blocks; add and delete blocks; change the code inside blocks; move code from one block to another; etc.

Much of the decision making is driven by ***data flow analysis***, which discovers properties of blocks that depend on other blocks. E.g.,

- which virtual registers are ***live*** (contain values that may be needed in the future at the end of a given block?
- which values are known to be ***available*** (contained in some virtual register) at the start of a given block?

Like the algorithm that builds predict sets for a top-down parser, the data flow "engine" begins with "obvious" facts and iterates until it can't learn anything more (and we can prove the answer has converged).

Conversion to low-level IF can be as simple as picking an order for the basic blocks of the control flow graph:

```
      call getint
      i := rv
      call getint
      j := rv

L1:   v1 := i
      v2 := j
      v3 := v1 != v2
      test v3
      if false goto L2

      v4 := i
      v5 := j
      v6 := v4 > v5
      test v6
      if false goto L3

      v7 := i
      v8 := j
      v9 := v7 – v8
      i := v9
      goto L4

L3:   v10 := j
      v11 := i
      v12 := v10 – v11
      j := v12

L4:   goto L1

L5:   v13 := i
      a1 := v13
      call putint
      halt
```

This is unlikely to give you the best code for a given target instruction set; more on this below.

Key tasks of target code generation

  ***instruction selection***
    This seems like it ought to be straightforward, but it can be tricky
      more than one way to do things on many machines
      multiply by 2 v. add to self v. left shift one bit
    messy addressing modes
    side effects (e.g., on condition codes or scratch registers)
  Common to make a simple choice,
    then follow up w/ machine-dependent code improvement
  Both simple choice and improvement may be based on automated pattern matching (code generator generator)

  ***instruction scheduling***
    order in which to execute logically independent instructions
    e.g.

```
r2 += r3 * r4   \
r1 := a         / swap these!
r2 += r1
```

  ***register allocation***
    what should be kept in registers when?
      NP hard in the general case -- equivalent to minimal graph coloring
      typical modern compilers use a heuristic solution to the coloring problem

  instruction scheduling and register allocation interact in complicated ways
    if you reorder instructions, the number of registers needed may change
      (have to hang onto a temporary value across the creation of some other temporary value)
    if you run out of registers, you have to ***spill*** them,
      which changes the set of instructions
      and the new instructions are loads and stores, for which scheduling is particularly important
    real compiler might
     - schedule instructions assuming unlimited registers
     - allocate registers, spilling as necessary
       (this is the NP hard graph coloring problem)
       typical modern compilers use a heuristic solution
     - reschedule to fill new load delays, so long as it doesn't mess up register allocation
    more on this in Chap 17 (not covered this semester)

FWIW, with aggressive (machine independent *and* machine independent) code improvement, even something as simple as the GCD program can produce surprisingly clever code.
The following is from LLVM -O3, hand translated from x86-64 assembly back into pseudocode for readability.

```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
      if (i > j) i = i – j;
      else j = j – i;
   }
   putint(i);
}
```

```
    call getint
    r1 := rv                 // r1 holds i
    call getint              // rv holds j
    compare rv, r1           // compare j to i
    goto L2 if equal
    r2 := 0
    compare r1, rv
L1:
    r3 := 0
    r3 := rv if less than    // "conditional move"
                             // based on most recent comparison
    r4 := r1                 // i
    r4 := r2 if less than
    r1 –:= r3                // i –= (i < j ? j : 0)
    rv –:= r4                // j –= (i < j ? 0 : i)
    compare r1, rv
    goto L1 if not equal
L2:
    a1 := r1                 // i
    call putint
```

The inner loop here is only 8 instructions long, compared to 20 in our naive linearized control flow graph.