==
PHASES OF COMPILATION

Compilers among the oldest and best understood complex programs
    date to late 1950s
    embody several lovely formalisms

Phase = large-scale step in the compilation process.

```
character stream
                                   scanner (lexical analysis)
token stream
                                   parser (syntax analysis)
parse tree
(concrete syntax tree)
                                   semantic analysis and
                                   intermediate code generation       FE
abstract syntax tree (AST) or                                      --------
other intermediate form (IF)
                                   machine-independent                ME
                                   code improvement
                                   (may actually be many phases)
modified intermediate form                                         --------
                                   target code generation             BE
target language
(e.g., assembly)
                                   machine-specific optimization
                                   (may also be multiple phases)
better target code

------
symbol table
```

Pass = set of phases that finish before the next pass starts.
Typically implemented as a separate program.
    historically, reduced the compiler's memory footprint
    today, serve to support _compiler families_
    N + M +1 separately developed passes instead of N * M +1
        (+1 for the "middle end" [the big part])

Phases within a pass may not be clearly differentiated.
    Most compilers, for example, do not build an explicit parse tree.

Automatic tools
leverage the formalisms on which the compilation process is based

```
scanner                -- definitely handy during language development
                          but may be hand-re-written for production use
parser                 -- big conceptual & organizational win
                          but may also be abandoned (e.g., in gcc)
                          to produce better error messages
attribute evaluator    -- not used all that much in practice, but
                          can be conceptually useful, and has been
                          applied to syntax-directed editors,
                          incremental compilation, and language
                          research
data-flow engine       -- very useful in the middle end: captures
                          incremental discovery of code properties --
                          e.g., which registers contain values that
                          might be needed after a subroutine call, and
                          ought to be saved on the stack
affine math framework  -- great for loop optimizations, characterization
                          of possible values array indices
code generator         -- great for portability; fairly widely used
```

More on the various phases:

All phases rely on the SYMBOL TABLE
  - keeps track of all the identifiers and what compiler knows about them
  - may be retained (in some form) for later use --
        by debugger, garbage collector, reflection mechanism, etc.

SCANNING divides the program into "tokens"
    smallest meaningful pieces of a program
    saves time by reducing the number of pieces the parser has to process
        (and scanning is faster than parsing)

Also typically
    removes comments
    saves text of strings, identifiers, numbers in the symbol table
    evaluates numeric constants (maybe)
    tags tokens with file/line/column, for good diagnostics in later phases

Consider an (extremely simple) language to describe the input to a
hand-held calculator. Tokens for such a language might include:

```
    id      = letter ( letter | digit ) *
                [ except "read" and "write" ]
    literal = digit digit *
    ":=", "+", "-", "*", "/", "(", ")"
    $$ [end of input]
```

(These are REGULAR EXPRESSIONS.)

PARSING discovers the "context free" structure of the program.
That's the structure (set of rules) that can be described with a
CONTEXT FREE GRAMMAR (CFG).

Continuing the calculator example, suppose
  - All variables are integers.
  - There are no declarations.
  - The only statements are assignments, input, and output.
  - Expressions get to use the four arithmetic operators and parentheses.
  - Operators are left associative, with the usual precedence.
  - There are no unary operators.

Here's a grammar, in EBNF (extended Backus-Naur form):

```
    <pgm>           ->  <statement list> $$
    <stmt list>     ->  <stmt list> <stmt> | ε
    <stmt>          ->  id := <expr> | read id | write <expr>
    <expr>          ->  <term> | <expr> <add op> <term>
    <term>          ->  <factor> | <term> <mult op> <factor>
    <factor>        ->  ( <expr> ) | id | literal
    <add op>        ->  + | -
    <mult op>       ->  * | /
```

The initial, "augmenting" production is for the parser's convenience --
    $$ is generated by the scanner; it isn't part of the user's program.

Note that there is an infinite number of grammars for any given language.
This is just one.

[An aside: You may recall from 173 that the "extra" levels of this
  grammar (expr v. term v. factor), and the choice of ordering within
  productions, serves to produce parse trees that make it easier to see
  the precedence and associativity of operators; more on that in Chap 2.

  Also: this grammar happens to belong to one of two main classes of
  grammars that are easily parsed. It's from a different class than the
  one you may have worked with in CSC 173. I'm using it because it's
  arguably more intuitive. More on this in the next lecture.]

Mini theory lesson:

    Scanners and parsers are *recognizers* for regular and context-free
    "languages," respectively.
      - useful for describing the language

    Regular expressions and context-free grammars are *generators* for
    regular and context-free languages, respectively.
      - useful for telling if a given string is in the language

    Scanner and parser generators like lex and yacc, or antlr, transform a
    generator (RE, CFG) into a recognizer (scanner, parser).

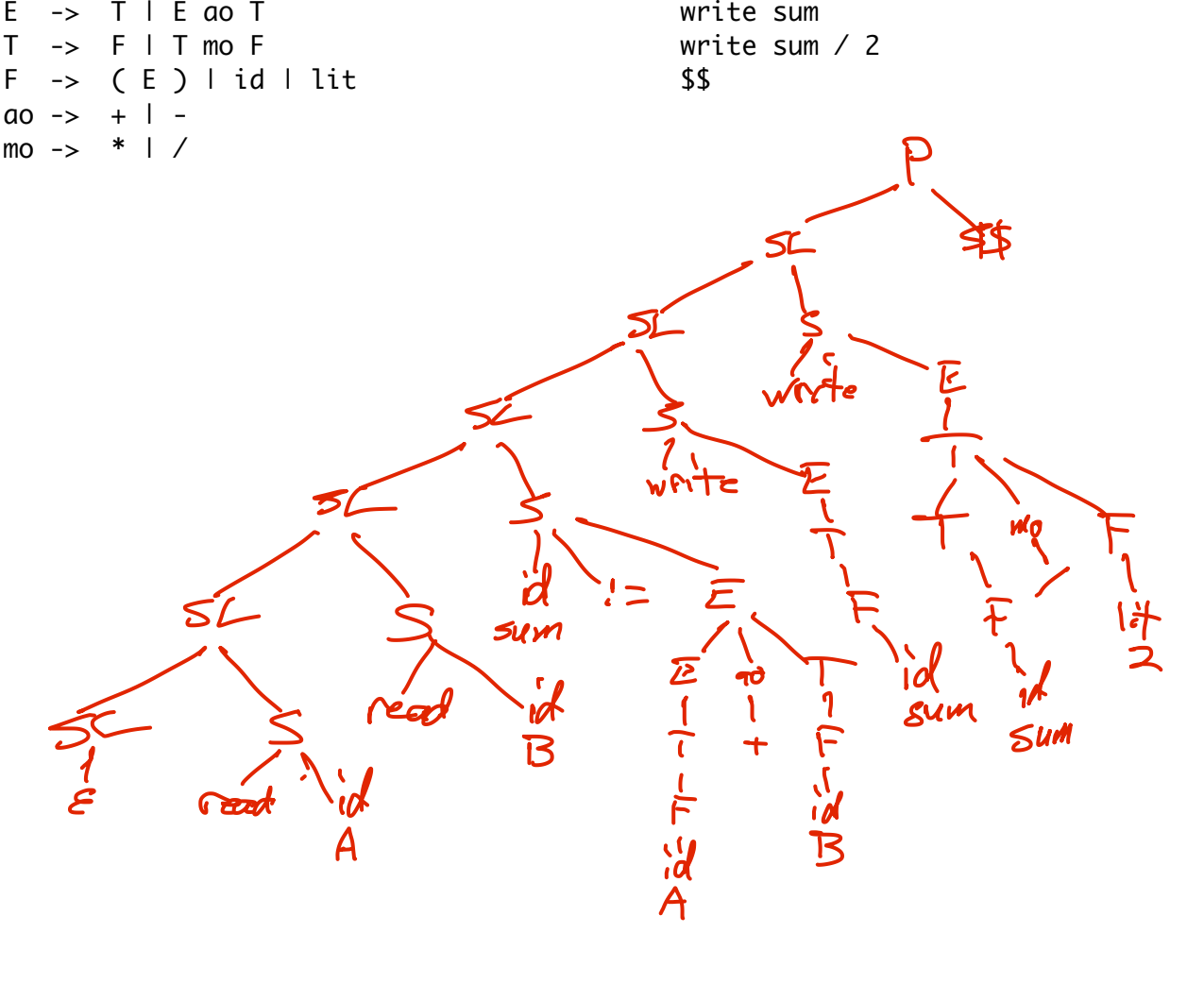
    [For those who've had CSC 280 or its equivalent, scanning is
    recognition of a regular language, e.g. via DFA; parsing is recognition
    of a context-free language, e.g. via PDA. If you don't know what that
    means, don't worry; we'll get to it.]

Using our grammar for the calculator language, consider the following
input program to print the sum and average of two numbers:

```
    read A
    read B
    sum := A + B
    write sum
    write sum / 2
    $$
```

50 characters in this program (including the spaces and line feeds)
Scanner turns them into 16 tokens (including the extra $$) and passes
    these on to the parser
The parser will discover the structure of the program and
build a PARSE TREE:

```
    P  ->  SL $$                              read A
    SL ->  SL S | ε                           read B
    S  ->  id := E | read id | write E        sum := A + B
    E  ->  T | E ao T                         write sum
    T  ->  F | T mo F                         write sum / 2
    F  ->  ( E ) | id | lit                   $$
    ao ->  + | -
    mo ->  * | /
```



SEMANTIC ANALYSIS is the discovery of "meaning" in the program.
[More accurately, it maps the program to something like math or a
formally specified abstract machine, to which humans already assign
meaning.]

The semantic analyzer enforces all the rules that can be enforced at
compile time (before the program runs), but which the couldn't be
expressed in the CFG. These are STATIC semantics.

Other rules (e.g. array subscript out of bounds) can't (in general) be
enforced until run time. Those are DYNAMIC semantics -- enforced (if at all)
by code that the compiler adds to your program, to execute at run time.

Examples of (typically) static semantic rules
  - identifiers must be declared before use
  - operands need to have matching types
  - subroutines need to be passed the right number and types of parameters
  - functions must contain return statements
  - labels on the arms of a switch (case) statement must be disjoint
  - and so on

Semantic analysis for the calculator language is essentially
non-existent -- little that CAN go wrong.
Since there are no branches in our control flow, however, we can check
to make sure no variable is used before it is given a value, and maybe
warn programmers if a variable is given a value that is never used.

[This is not possible in a more general language, unless you impose
  restrictions on merging code paths, as Java and C# do. A good
  compiler may catch some errors, even if it can't catch all of them.]

```
    read A              read A              read A
    write B             read B              read B
                        write A             C := A + B
                                            C := 5
```

Semantics analysis is often done together with INTERMEDIATE CODE
GENERATION, in a single phase.

A parse tree reflects the structure of a program according to a CFG.
    Sometimes called a "concrete syntax tree"
    Typically has lots of extraneous detail --
        e.g., expr, term, and factor in our calculator example

Before enforcing semantic rules, we typically want to create a more
convenient structure -- the ABSTRACT SYNTAX TREE (AST).

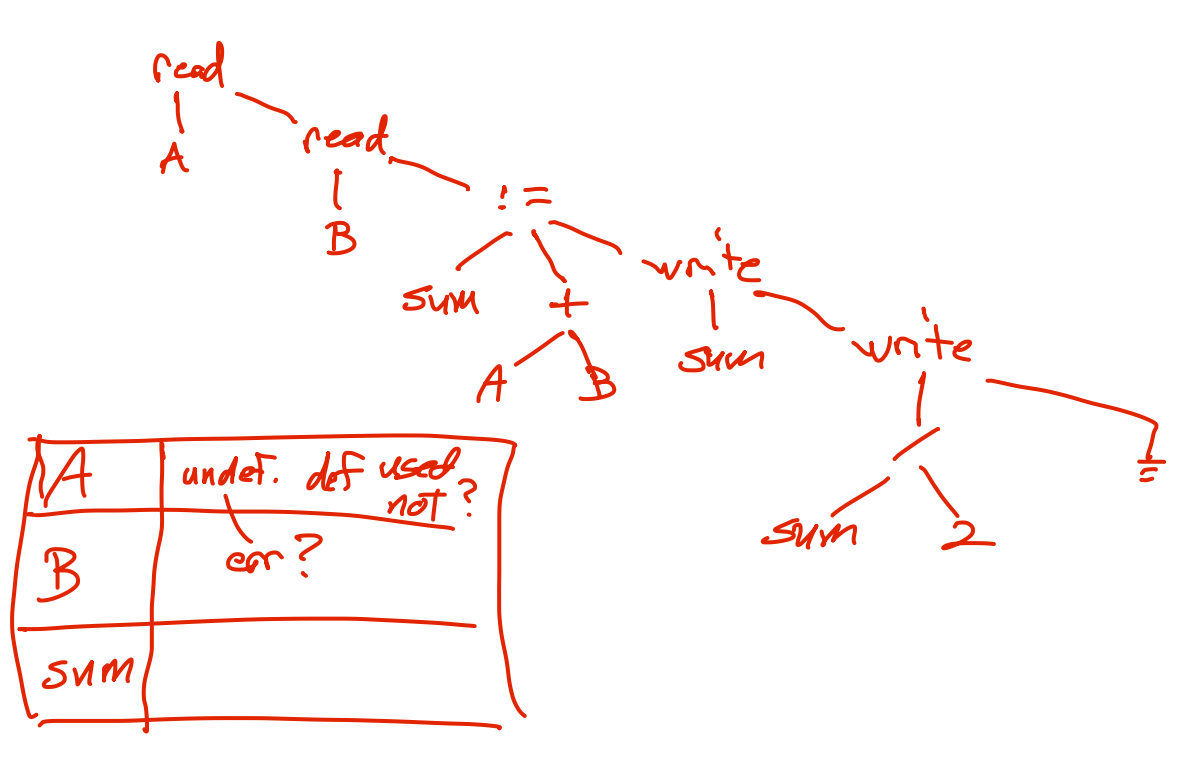
For brevity, I'll say "parse tree" instead of "concrete syntax tree"
and "syntax tree" instead of "abstract syntax tree" or AST.

In practice, construction of the AST is often interleaved with parsing,
so we don't actually have to build the parse tree.

The semantic analyzer typically works by walking the AST and labeling
(ANNOTATING) nodes. Labels might include
  - pointers into the symbol table
  - types of expressions
  - accumulated error messages
  - many others

The syntax tree for our sum-and-average program might look like this:

```
    read A
    read B
    sum := A + B
    write sum
    write sum / 2
    $$
```



If we traverse this tree left-to-right (given the calculator's simple
linear control flow), we can keep track in the symbol table of which
variables have been given a value, and which values have been used.

  - When we see an identifier, we look it up in the symbol table.
  - If it isn't there already, we add it.
  - For each symbol, we keep track of whether it has (a) no value,
    (b) an unused value, or (c) a used value.
  - For unused values, we may also keep track of whether an error message
    has been generated (to avoid redundant messages).
  - Initially, every variable has no value.
  - Whenever we give a symbol a value we check to see if it already has
    an unused value.
      - If so, we print a warning message.
      - In either case, we note that it now has an unused value.
  - Whenever we try to use a symbol's value we check to see if it
    currently has no value.
      - If so (and perhaps if we haven't already complained),
        we print an error message.
      - Otherwise, we note that it now has a used value.
  - At the end of the program, we scan the whole symbol table to see if
    anything has an unused value. If so, we print a warning message.

Again, most compilers for "real" languages would not track values this
way: conditions, loops, and subroutines preclude it. They _would_ do
things like type checking.

The Scanner, Parser, and Semantic Analyzer together are the FRONT END of
the compiler -- the language-dependent part. The same front end would
be used by an interpreter.

Next is the "middle end" -- MACHINE-INDEPENDENT CODE IMPROVEMENT
    a.k.a. OPTIMIZATION
Usually comprises multiple phases -- often dozens of them.
Each takes an intermediate-code program and produces another that does
the same thing faster, or in less space.

Such phases are often optional: they increase compilation time, but
produce better code.

Code improvement is the bulk of a modern compiler, but we won't have
time for much coverage this semester.
Take 2/455 to learn more, or read Chap 17 on the PLP CS.

Optimization phases often proceed through several progressively "lower"
(more machine-like) intermediate forms.
LLVM, gcc, and many other compiers have three main levels
    (each of which may have many sub-levels)

    high level -- abstract syntax tree
    medium level -- often some sort of CONTROL FLOW GRAPH with
        idealized assembly code within straight-line BASIC BLOCKS
    low level -- typically the assembly code of the target machine,
        or something very close

The typical phase traverses the current IF, adding annotations and
perhaps producing a "lower" IF.

(An interpreter, of course, uses a traversal to "run" your program.)

Annotations created in the middle end might include
    which recently computed values are still "live"
        (may be needed later in the program)
    which functions may call a given function
    which variables a pointer may refer to
    which variables are changed in the body of a loop
    how many times a loop is likely to run
    which expessions are evaluated in a given body of code
        (useful for finding redundancies)
    what ranges of values might be held in a given variable
    which values can actually be determined at compile time
    and many many more

All of these can help the compiler create a revised IF that is likely to
produce a faster or smaller program.

The back end of the compiler starts with TARGET CODE GENERATION.
This phase typically produces assembly language or (sometimes) machine
language. It is driven by one or more additional traversals of the IF
produced by the middle end.

Among other things, the target code generator must decide how to use the
resources of the target machine.
    layout of memory
    registers to reserve for special purposes
    calling conventions and layout of the stack
    etc.

Annotations in this step might include
    sizes of variables
    locations of variables in memory (absolute, or offset in stack frame)
    names and locations of temporary variables created to hold intermediate
        results of complicated computations
    which variables are temporarily held in which registers
    statistics on the range of case statement labels
        (to drive a look-up strategy)

In our calculator example, the simple sum-and-average program might be
translated into the following (very naive!) code for the x86:

```
        .data
    A:              .long   0
    B:              .long   0
    sum:            .long   0
        .text
    __start:
         call    input
         movl    %eax, A
         call    input
         movl    %eax, B
         movl    A, %eax
         movl    B, %ebx
         addl    %ebx, %eax
         movl    %eax, C
         movl    C, %eax
         push    %eax
         call    output_int
         addl    $4, %esp
         movl    C, %eax
         movl    $2, %ebx
         cltd
         idivl   %ebx
         push    %eax
         call    output_int
         addl    $4, %esp
         leave
         ret
```

This is obviously not the best code for our program.
You can see where it came from, though.

At the very least, a real compiler would want to track which values are
in registers so it can avoid all the redundant loads and stores.

The final phase is MACHINE-SPECIFIC CODE IMPROVEMENT. This serves
mainly to take advantage of special features of the hardware and to
identify idioms that can be replaced with something simpler.
    As a very simple example, consider multiplication by 0 or 1.
    It's often easier to fix such things in the optimizer than to
        generate the better version in the first place.

The calculator language is too simple to really illustrate this.

Some remaining units this semester will focus on compiler (and
interpreter) implementation. These will be interleaved with units that
focus on language design. Framework presented here will hopefully
provide useful context.