Notes for CSC 2/454, Jan. 21 and 23, 2025

CSC 2/454 Programming Language Design and Implementation
==========
Course Introduction

Language Design and Language Implementation go together
  implementor has to understand the language
  language designer has to understand implementation issues
  ** good programmer has to understand both

LOTS of programming languages
  Wikipedia's list has 674 entries as of Dec. 2024
    those are just the "notable" ones

Why so many?
  evolution—we've learned better ways of doing things over time
  diverse ideas about what is pleasant to use
  orientation toward special purposes (SQL)
  orientation toward special hardware (assembly, CUDA)
  market factors: desire to control, or avoid what others control
    (COBOL, PL/I, Ada, Swift, ...)

What makes a language successful?
  easy to learn (BASIC, Scheme, LOGO, Python)
  "powerful"—easy to express complicated things (if fluent)
    (C++, Common Lisp, Haskell, Perl, APL)
  easy to implement (BASIC, Forth)
  possible to compile to very good (fast/small) code (C, Fortran)
  exceptionally good at something important (PHP, Ruby on Rails, R, SQL)
  backing of a powerful sponsor (COBOL, Ada, Visual Basic, C#, Swift)
  wide dissemination at minimal cost (Pascal, Java, Python, Ruby)
  market lock-in (JavaScript)

What is a programming language *for*?

- abstraction of virtual machine—way of specifying what you want the hardware to do without getting down into the bits

    - languages from the implementor's point of view
- way of thinking—way of expressing algorithms
    - languages from the user's point of view

This course tries to balance both perspectives.

* Knuth: Computer Programming is the art of explaining to another human being what you want the computer to do.

This course should help you

- learn new languages more easily
- pick the right language for the task at hand (given a choice)
- choose among alternative ways to express things in a given language
- understand what a compiler does to your code
    - for performance and (sometimes) correctness debugging
- emulate useful features in languages that lack them
- use language & compiler technology in your own projects
    - almost every complex system has an input language
- prepare for 2/455 🙂

Key to all of this is understanding the *concepts behind* language design—thinking about languages not in terms of syntax but in terms of

- naming & binding (early? late?)
- data types and abstraction mechanisms
- control flow
- closures
- concurrency
- ...

Units on

- intro
- syntax
- names
- semantics
- control flow
- functional programming

- type systems
- subroutines
- concurrency
- composite types
- objects and scripting
- code generation and run-time systems

- (see the web site)

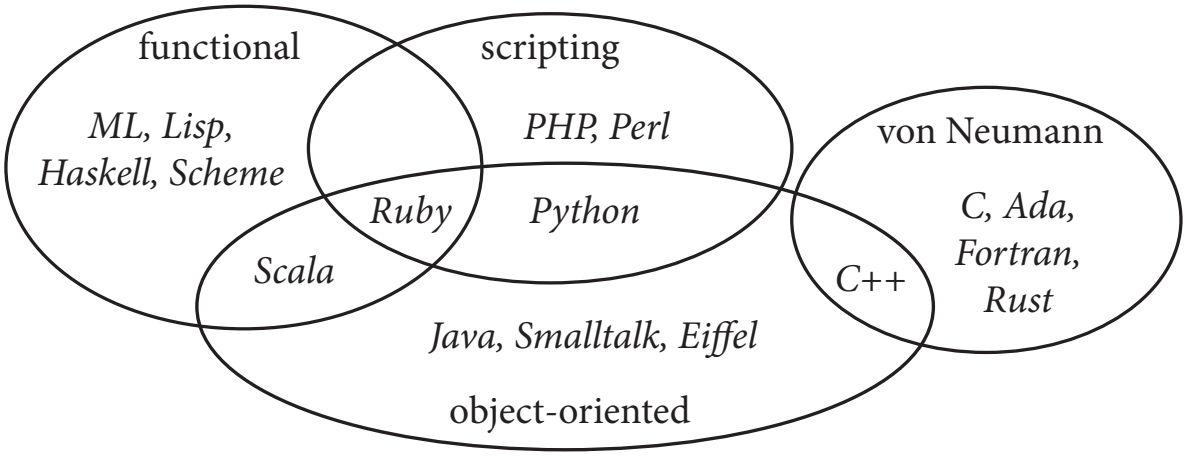
Traditional to group languages in terms of "paradigm"

imperative	
von Neumann	(Fortran, Ada, Pascal, Basic, C, Rust, ...)
object-oriented	(Smalltalk, Eiffel, C++, Java, C#, Swift, OCaml, ...)
scripting	(perl, Python, PHP, Ruby, Javascript, Matlab, R, ...)
declarative	
functional	(Scheme/Lisp, ML/OCaml/Haskell/F#)
logic, constraint-based	(Prolog, OPS5, spreadsheet, XSLT)

Not clear this ever really made sense: categories are not mutually exclusive, and have been getting less so over time. Today, probably best to talk about paradigms a language *supports* rather than "the" paradigm to which it belongs.

- Imperative languages emphasize computation by modifying variables. This allows you to do unbounded amounts of work in loops.
- Functional languages emphasize computation by creating, manipulating, and invoking functions. This allows you to do unbounded amounts of work via recursion.
- Object oriented languages emphasize structuring the code around abstract data types and their operations (methods).
- Scripting languages emphasize delayed decision making, programmer flexibility, pattern matching, and the ability to "glue" existing programs together.
- Logic languages emphasize the search for values that satisfy certain constraints. We'll touch on them a few times this semester, but they won't get as much emphasis as the others (sorry!)

So: paradigms sort of give us a Venn diagram:



(A few languages—logic, constraint-based, dataflow—lie in separate bubbles.)

Imperative languages have historically dominated—usually, today, with object oriented features. They'll get the bulk of our attention in this course.

BUT

- one unit and lots of scattered attention to functional languages
- lots of functional features making their way into mostly-imperative languages—Scala, Swift, Ruby, Python, Rust, ...
    - lambda expressions
    - functions as arguments and return values
    - list comprehensions
    - continuations

The imperative and functional paradigms tend to encourage different ways of thinking about algorithms. I'll be talking about this a lot, and encouraging you to think in both ways (because neither is better).

Will probably draw examples from about 40 languages this semester.
Will do projects in at least 6 of them
By the time we're done, you should be able to pick up a new language in a weekend (though becoming an expert will still take time).

==========

Course Administration

Still trying to figure out what works post-pandemic. Still believe in in-person instruction and in traditional textbooks. But providing lots of resources; pick what works for you.

NAVIGATION:

Course materials are available as a combination of open web and Blackboard. The home page is http://www.cs.rochester.edu/courses/254/spring2025/. Open up and browse. Pay particular attention to the course description, schedule, policies, and grading standards. The schedule is the hub of the course. It will guide you through all requirements. These notes are also on the web site.

Blackboard contains stuff that shouldn't really be public: announcements, discussion board, quizzes, "trivia" assignments, grades, recorded lectures. You should check both the announcements and the discussion board frequently.

PREREQUISITES: CSC 173 and 252, or equivalent.

Most of the students in the class are undergrads, but about 10% are grad students, who take it as 454 instead of 254. Grad students may be expected to do some extra work, and will be graded on a separate curve.

TEXTBOOK: *Programming Language Pragmatics*, 5th edition. Previous editions will not suffice. So-called "supplemental" sections (some of which I'll be assigning) are available online (see link on the course home page).

> *As of the the start of classes, Elsevier has not yet delivered copies of the 5th edition. Until they do, I'll be making preprints available in Blackboard on a chapter-by-chapter basis.*

CLASS MEETINGS (mandatory): Tuesday and Thursday, 3:25-4:40. I'll be taking attendance. Panopto recordings will be available in Blackboard as an optional study aid—not an alternative to attendance. Canned recordings from 2020 are also available in Blackboard.

OFFICE HOURS (tentative)
Prof: M 10–11, F 3:30–4:30, or by appointment
Grad TA: TBA (see home page)

WORKSHOPS
Required for 254; recommended for 454. (Grads must tell me if they don't plan to attend.) UG TAs the leaders.
Ignore what you signed up for at registration; we'll assign based on forms that you're about to fill out.

PROGRAMMING PROJECTS:
4 planned:

1) familiarization assignment (combinatorial search, in several different languages)
2) table-driven top-down parsing and error recovery (in Rust)
3) interpretation (in OCaml)
4) concurrency (in Java)

These will be similar BUT NOT THE SAME as assignments I've used in the past.

Expect to work hard.
Comparable amount of code to 173 but MUCH more difficult.
About two weeks per project NEEDED.

Each project will begin with a pre-assignment ("trivia") whose goal is to force you to ***look*** at things early. Disproportionate share of final course grade (~5%)

COMPUTING RESOURCES:

Everyone (including grad students) will need a CSUG acct. CS majors should have one already. If you don't, contact the grad TA.

The Wegmans Hall majors lab is available if you want a big local screen.
Otherwise ssh should suffice. Connect to the csug cycle servers (cycle1, cycle2, cycle3). You will need VPN to access these from off campus.

Feel free to use your own machine for development if you want, but code will be turned in and graded at CSUG. Port and test early!

QUIZZES and EXAMS:

Quiz on Blackboard once per unit (~12 times this semester)
Based on the textbook reading

Midterm and cumulative final exams

GRADING (tentative):

Programming projects
- 5% "trivia" pre-assignments
- 25% main projects

Exams
- 24% midterm
- 28% final

Keeping up
- 5% weekly quizzes
- 8% workshop participation
- 5% lecture attendance

NO LATE ASSIGNMENTS OF ANY KIND WILL BE ACCEPTED.
EXCEPTIONS ONLY UNDER THE MOST DIRE OF CIRCUMSTANCES.
TURN IN WHAT YOU HAVE; I'M GENEROUS WITH PARTIAL CREDIT.

COLLABORATION AND ACADEMIC HONESTY:

Exams are individual effort only; closed book.
Quizzes are also individual effort, but open book.

COLLABORATION ON IDEAS is encouraged, but you have to work through everything yourself. You can explore anything you want with a friend or explore whatever you want on a whiteboard, THEN ERASE IT
- NO NOTES—just memories

COLLABORATION ON ARTIFACTS (copying) is EXPRESSLY FORBIDDEN, unless you

- have permission
- clearly indicate in your README file which parts were copied and from whom
- don't expect points for the copied parts (but may get the satisfaction of being able to see the whole thing work)

Encouraged to help others: won't hurt your grade.
Everybody gets an A if they deserve it.

LLMs *strongly discouraged*. If you use one, it counts the same as consulting a person: you can look *once*, memorize what you want, close the window, and then do your own work. Or you can copy code, explain your use in your README at turn-in time, and expect credit only for the other parts.

*** SEE FULL DETAILS ON ACADEMIC HONESTY ON THE WEB PAGE ***

Apparent violations will be referred to the Honesty Board.

GETTING HELP

This is a hard course. Don't wait to seek help.

- read the book
- attend lecture and workshops
- talk to fellow students
- go to office hours
- post questions in discussion group in Blackboard
- make an appt to talk to me
- check out the CSUG and CCAS tutoring services

FIRST ASSIGNMENT (ASAP):

1) Explore both the website and the Blackboard site for the course.
2) Get a copy of the book if it's available, or find the preprints in Blackboard.
3) Read (all of) chapter 1.
4) Make sure your CSUG account is up and working.

5) Take quiz Q1 on Blackboard (due 26 Jan.)
6) Complete “Initial Trivia assignment (T0)” on Blackboard (due 23 Jan.)
7) Check out the “Unix tools” assignment (A0) and work through it if it isn’t all familiar material.

That probably looks like a lot, but only #s 3 and (maybe) 7 will be time-consuming.

There will be similar requirements in future weeks. I won’t be putting lists like this in future lecture notes; follow the schedule page on the web site.

==========

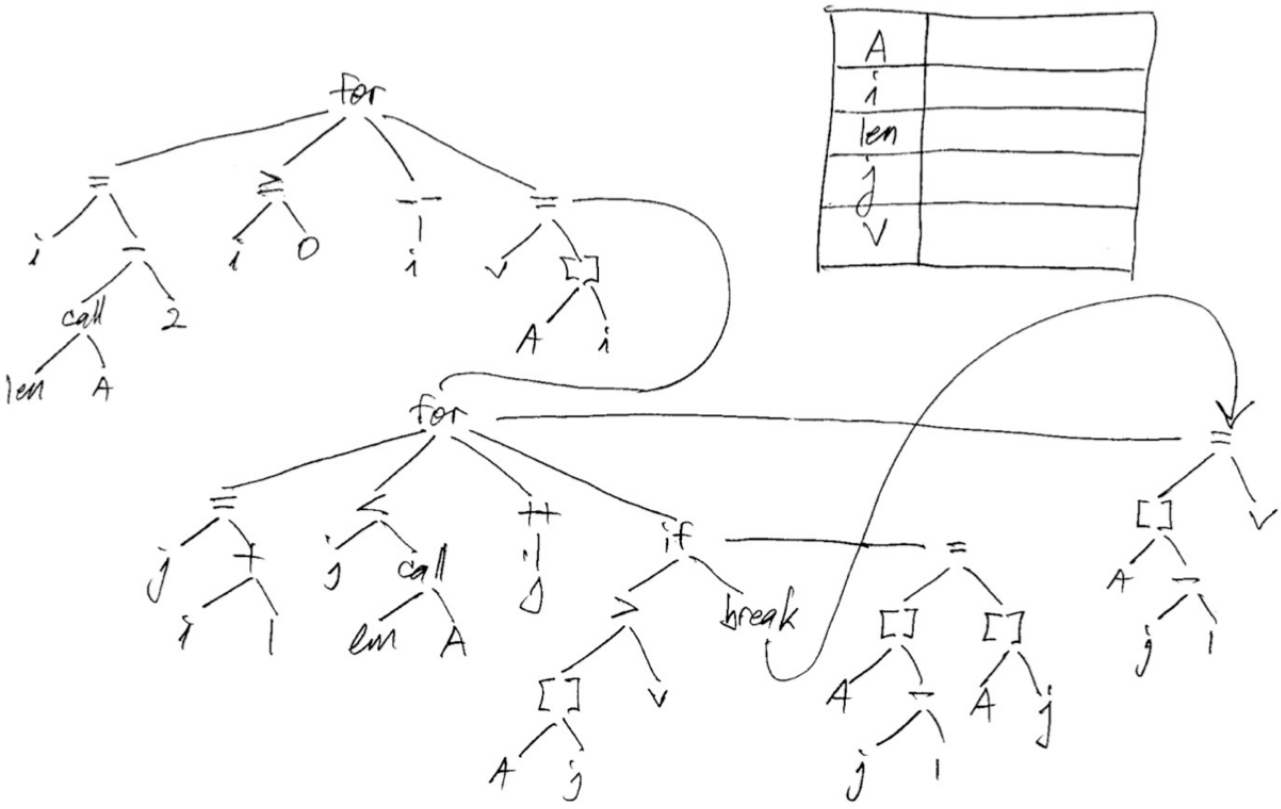
Compilation and Interpretation

Consider our insertion sort in C:

```
void sort(int A[]) {
    for (int i = len(A)-2; i >= 0; i--) {
        int v = A[i];
        int j;
        for (j = i+1; j < len(A); j++) {
            /* A[j..] is sorted */
            if (A[j] > v) break;
            A[j-1] = A[j];
        }
        A[j-1] = v;
    }
}
```

275 characters of text in a .c file. How do you *execute* that? Not immediately obvious, certainly, and a lot less obvious if it's 275 million characters.

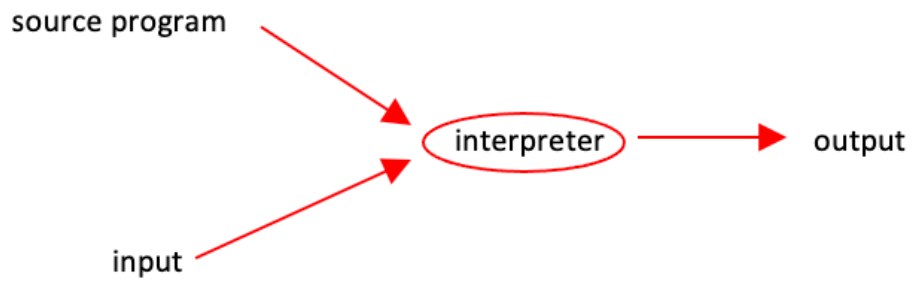
But suppose it's a tree data structure in memory:



Hopefully most of you believe that (given some time) you could write a program that would take any such tree and "execute" it. That's what an INTERPRETER does:

- translate the source program into a data structure that makes its meaning more obvious
- walk the data structure (in this case, a tree) and do "the obvious"

Most scripting languages (Perl, Python, Ruby, Javascript) are implemented in roughly this fashion.



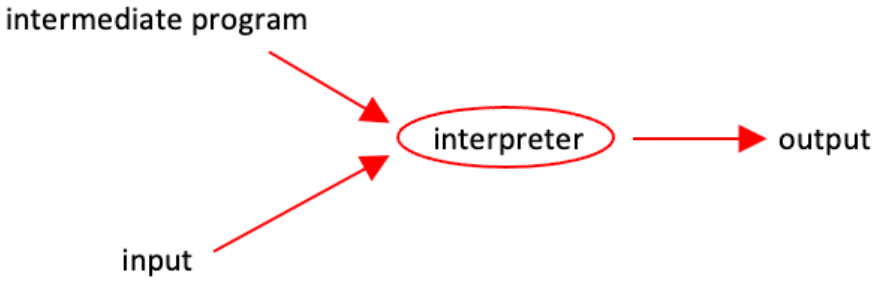
** The interpreter stays around at execution time.
Comparatively simple. Very flexible. But generally kind of slow.

At the other extreme (as in, say, Fortran or C) we can translate a program to machine language ahead of time. That's what a COMPILER does:

translate the source program into a data structure that makes its meaning more obvious (the same as in an interpreter!)
walk the data structure and *generate machine code* to do "the obvious"

A common intermediate is to employ a non-machine-language *intermediate form* and to separate the creation of the internal form from the "execution" part. Java does this:

AND THEN EITHER



OR

The former option (final step is interpreter) was common in early Java implementations. Most now do the second option: "just-in-time" (JIT) compilation. Advantages

- Intermediate program (Java byte code) is significantly smaller than textual source: good for shipping over the web.
- JIT compilation is faster than source-to-machine translation, because the intermediate program has lots of semantic information built in (doesn't have to be figured out again).
- Intermediate program is completely portable and self-contained: "run anywhere" on VIRTUAL MACHINE.

So compilation & interpretation are more shades of gray than distinct alternatives. In fact:

- In some systems, you'll see "pre-processing" prior to compilation or interpretation. The key difference between pre-processing and compilation is that compilation entails semantic *understanding* of what is being processed; pre-processing does not.
  - A compiler produces either error messages or output that will pass through further steps—more compilation, assembly, interpretation, execution—without syntactic or static semantic errors.
  - A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not.

- How you view all this also depends on how deep you look. Consider
  - microcoded processor (interpretation)
  - micro-ops on a modern x86 (JIT translation)

- Many compiled languages have interpreted pieces—e.g., printf in C

- Most compiled languages use "virtual instructions"—library routines that are called automatically by the compiler:
  - math
  - I/O
  - string manipulation
  - set and map operations

- some compilers produce nothing but virtual instructions—e.g. Pascal P-code, Java byte code, Microsoft CIL

What makes compilation hard?—late binding
names to objects—scope rules
types to objects/names—type rules
programs to code—dynamic classes in Java, new functions at run time in Scheme

Why interpret?
necessary for late binding, which may increase programmer productivity
small code size
good diagnostics
no (or reduced) compilation step—fast startup from source code

(possibly) enhanced portability
automatic inclusion of the latest libraries

Commonly interpreted languages
Scheme
Prolog
Shell
most scripting languages (Python, Ruby, PHP, JavaScript)

Compilers exist for some of these, but they aren't pure: selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source. Interpretation (or dynamic compilation) of parts of code, at least, is still necessary.

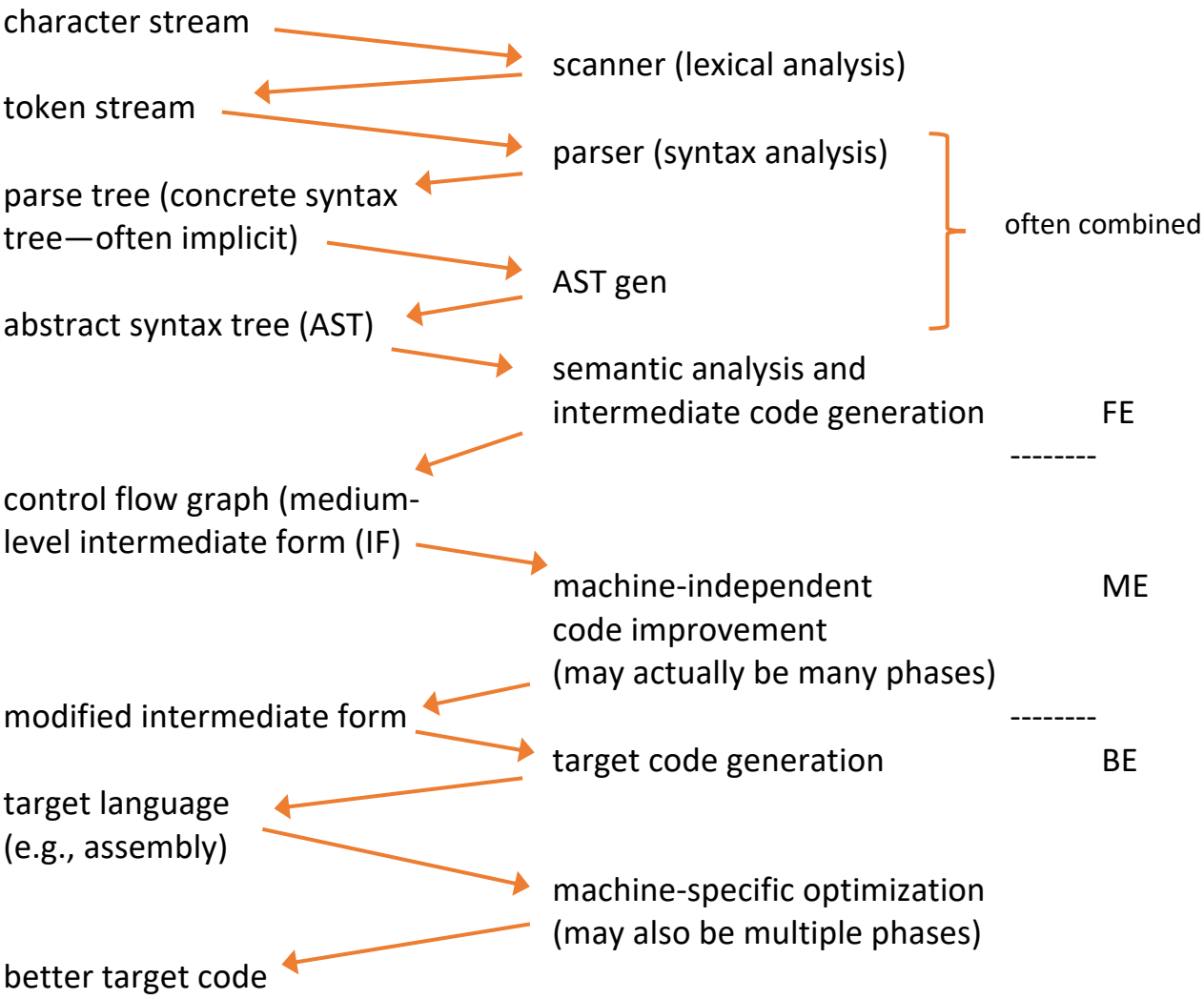
Unconventional compilers
text formatters
silicon compilers
database query language processors
XSLT

==========

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.



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.

Many aspects of compiler construction can be automated. Scanner and parser for sure. Often parts of the code improvers. Sometimes the code generator.

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       ⟶ stmt_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.]

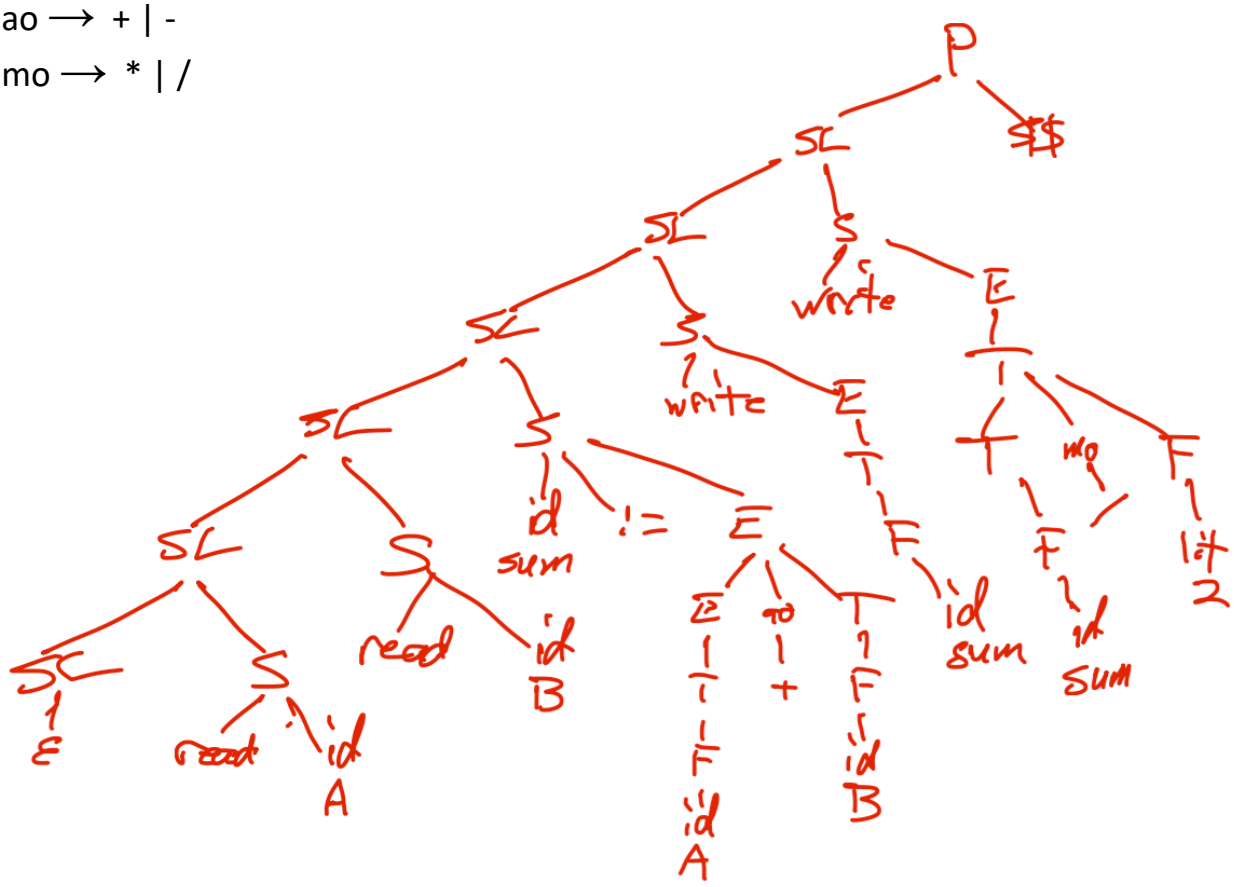
Using our grammar for the calculator language, consider the following input program to print the sum and average of two numbers:

```
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 $$
SL ⟶ SL S | ε
S ⟶ id := E | read id | write E
E ⟶ T | E ao T
T ⟶ F | T mo F
F ⟶ (E) | id | lit
ao ⟶ + | -
mo ⟶ * | /

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



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 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 *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.

INTERMEDIATE CODE GENERATION is often done together with semantic analysis, 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

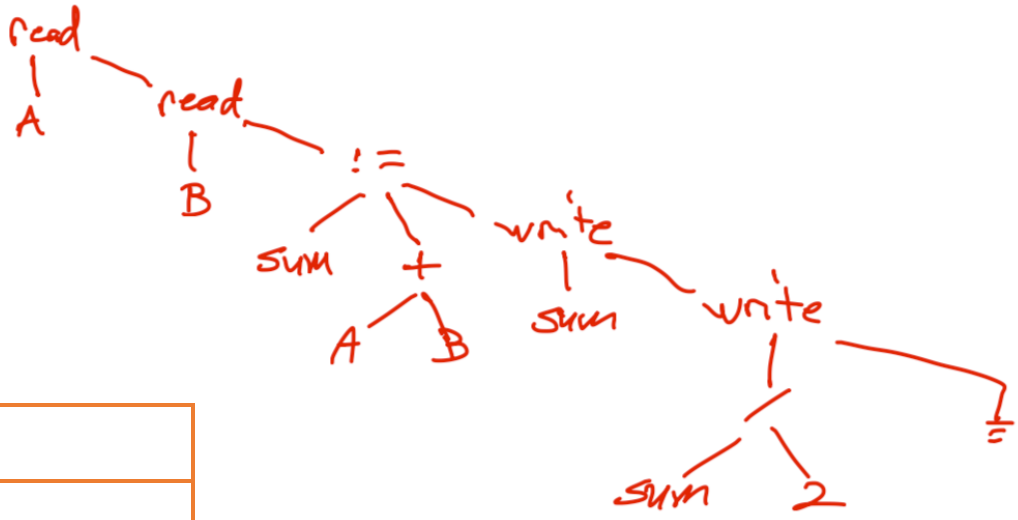
Before enforcing semantic rules, we typically want to create a more convenient structure—the ABSTRACT SYNTAX TREE (AST).

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:



A	
B	
sum	

If we traverse this tree left-to-right (given the calculator's simple linear control flow), we could (in a more complicated language) keep track in the symbol table of things like what has been declared and what type it has. That would let us issue "*X* has not been declared" and "*Y* and *Z* have incompatible types" messages.

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 compilers 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 expressions 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.

TARGET CODE GENERATION is the first phase in the back end of the compiler.
Typically produces assembly language or (sometimes) machine language.
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.