Notes for CSC 2/454, Aug. 31 and Sep. 7, 2022

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

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

LOTS of programming languages
  Wikipedia's list has 671 entries as of Aug. 2020
    those are just the "notable" ones

Why are there so many programming languages?
  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)

Why do we have programming languages? -- what is a 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 coverage of these two angles. We will talk about language features for their own sake, and about how they can be implemented.

* 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

- syntax
- semantics
- functional programming
- names
- scripting

- control flow
- type systems
- concurrency
- composite types
- subroutines
- objects
- run-time systems

- (see the web site)

Traditional to group languages in terms of "paradigm"

| | |
|---|---|
| imperative | |
| von Neumann | (Fortran, Ada, Pascal, Basic, C, ...) |
| 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.

We'll discuss all of this much more as the semester goes on. For now:

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.

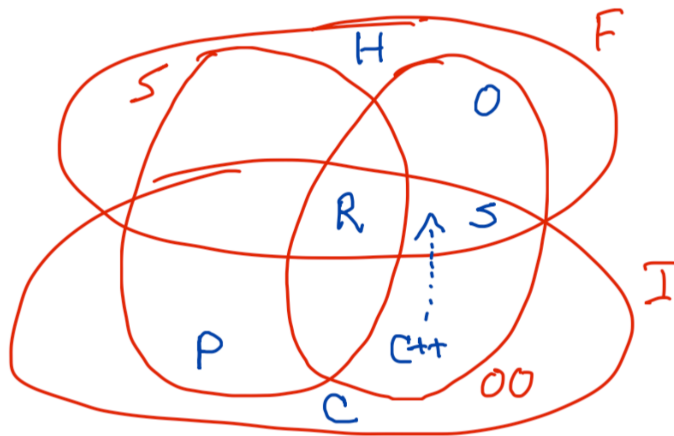
Object oriented languages emphasize structuring the code around abstract data types and their operations (methods).

Scripting languages emphasize delayed decision making and programmer flexibility.

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:

Haskell
C
OCaml
C++
Ruby
Scala
Perl
eLisp




Imperative languages have historically dominated -- usually, today, with object oriented features.

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, ...
    - 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)

Consider insertion sort. In no particular languages:

```
imperative sort(A):
    for i in len(A)−2 downto 0
        v = A[i]
        for j in i+1 to len(A)−1
            // A[j..len(A)−1] is sorted
            if A[j] > v break
            A[j−1] = A[j]
        A[j−1] = v

functional sort(A):
    if len(A) < 2 return A
    else
        let v be A[0] and R be A[1..]
        return insert(v, sort(R))
    where insert(v, S):
        if len(S) == 0 return { v }
        else
            let w be S[0] and T be S[1..]
            if v < w return v . S
            else return w . insert(v, T)
```

These implement the same algorithm. They are likely to compile to nearly identical machine code. But

a) The functional version has no assignments.
b) The imperative version has a more obvious implementation.
c) when I wrote these in C and Scheme, I had to fix two bugs in the C version, but the Scheme one ran the first time.

Will probably draw examples from about 40 languages this semester
Will do projects in 6 or 8 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

Tried ”flipping" the course last year; didn’t work well.
Going back to a traditional organization this year.

NAVIGATION:

All course materials will be online; no handouts.
Combination of open web and Blackboard; latter only for non-public stuff.
The course home page is
  http://www.cs.rochester.edu/courses/254/fall2022/

Log in and browse. Pay particular attention to the course description, schedule, policies, and grading standards.
Log into Blackboard as well (links are on the course home page).

The TAs and I will post announcements to Blackboard. Everyone should check both the announcements and the discussion board every day.

Lecture notes are available on the web site.

The hub of the course is the schedule page:
  http://www.cs.rochester.edu/courses/254/fall2022/schedule.shtml
It will guide you through all requirements.

PREREQUISITES: CSC 173 and 252, or equivalent.

If you have not had 252 (it's a new pre-req), you can get by, but read chapter 5 (on the web) on your own.

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 will be expected to do some extra work, and will be graded on a separate curve.

The text for the course is *Programming Language Pragmatics*, 4th

edition. Previous editions will not suffice. So-called "supplemental"
sections (some of which I'll be assigning) are available online at
Elsevier's web site (again, with a link from the course home page).

CLASS MEETINGS (mandatory)

Monday and Wednesday, 10:25-11:40 US Eastern time. I'll be taking attendance.

WORKSHOPS

Required for 254; recommended for 454. 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:

5 planned:

(1) familiarization assignment (combinatorial search)
 in several different languages.
(2) syntax error recovery (recursive descent review; C++)
(3) simple translation (tree traversal, OCaml)
(4) concurrency (probably in Java)
(5) disassembler (scoping, scripting)

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 (~10%)

COMPUTING RESOURCES:

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

The Wegmans Hall majors lab and Hylan hall minors lab are available if you
want a big screen. Otherwise ssh should suffice. Use 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
   10% "trivia" pre-assignments
   33% main projects
Exams
   16% midterm
   16% final
Keeping up
   10% weekly quizzes
   10% 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 on
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.

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

Apparent violations will be referred to the Honesty Board.
(I had a lot of these last year; let's not do a repeat.)

GETTING HELP

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

- read the book and watch the lectures

- attend lecture and workshops
- talk to fellow students
- go to TA office hours or send them mail
- make an appt to talk to me
- check out the CSUG and CCAS tutoring services

FIRST ASSIGNMENT (for Wed. Sep. 7):

1) Get a copy of the book if you haven't already.
2) Finish reading (all of) chapter 1.
3) Explore both the website and the Blackboard site for the course.
4) Take quiz Q1 on Blackboard.
5) Make sure your CSUG account is up and working.
6) Complete "Initial Trivia assignment (T0)" on Blackboard.
7) Turn in the workshop time slot preference survey.
8) 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 2 and (maybe) 8 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.

==========================================

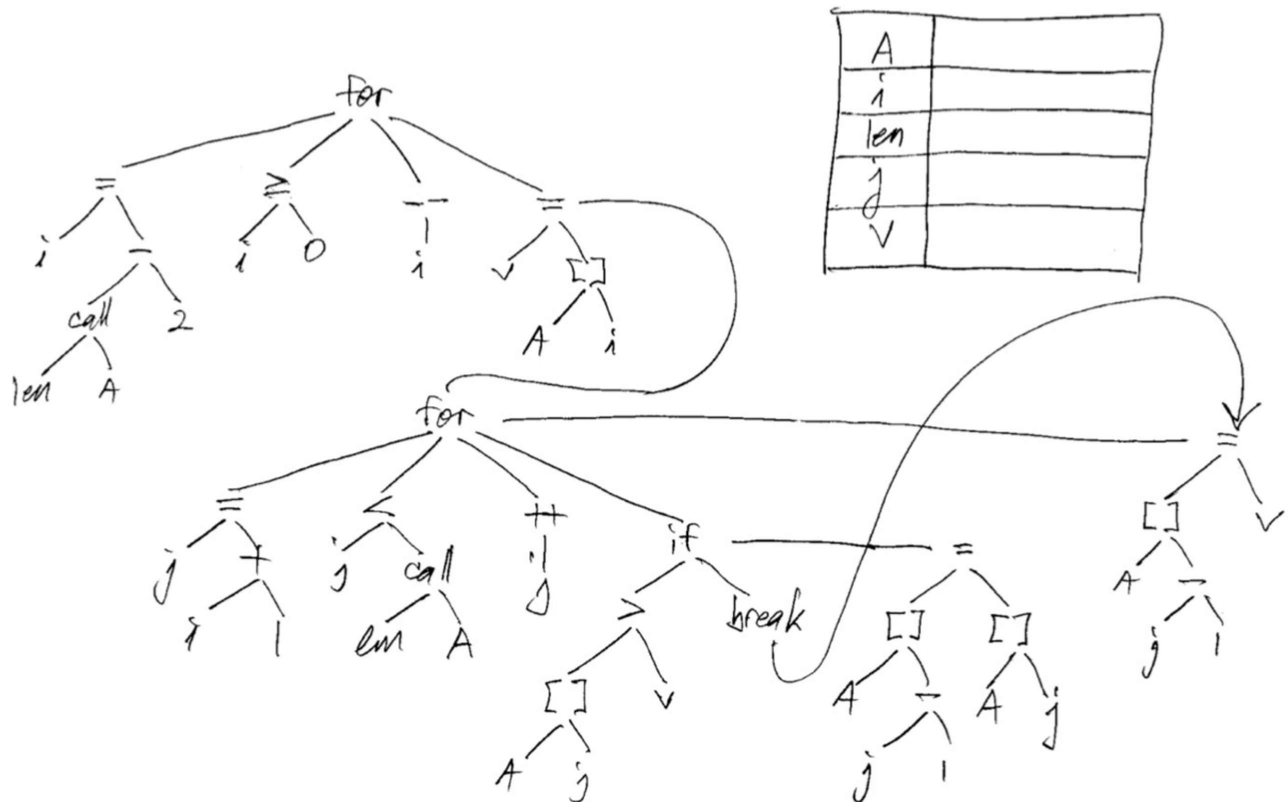
Compilation and Interpretation

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:

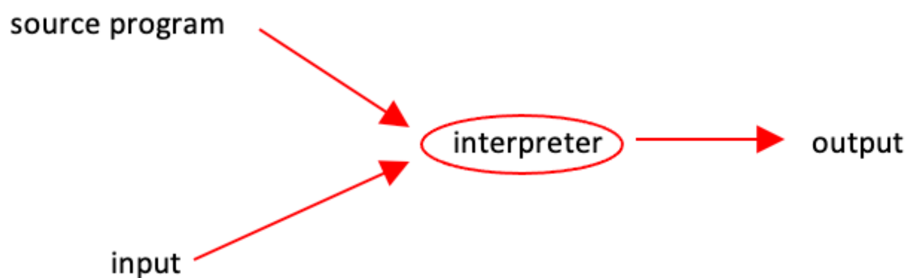


Hopefully most of you believe that (given some time) you could write a program that would take any such tree and "execute" it. That's what an INTERPRETER does:

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.

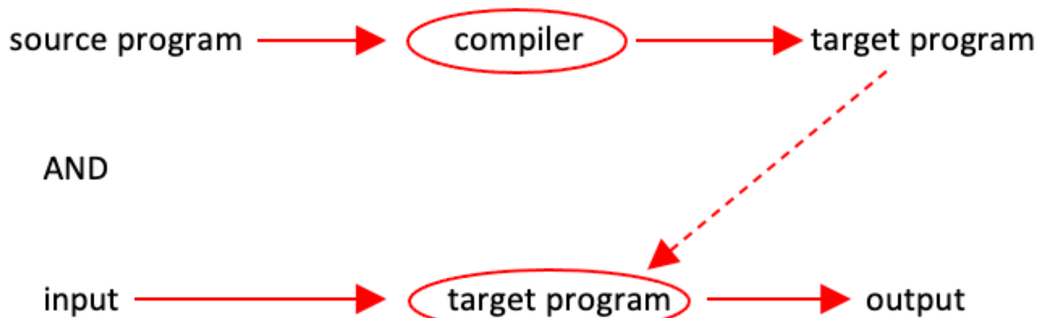



** The interpreter stays around at execution time.
Comparatively simple. Very flexible. But generally kind of slow.

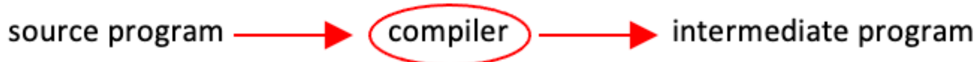
At the other extreme (as in, say, Fortran or C) we can translate a program to machine language ahead of time. That's what a COMPILER does:

translate the source program into a data structure that makes its meaning more obvious (the same as in an interpreter!)

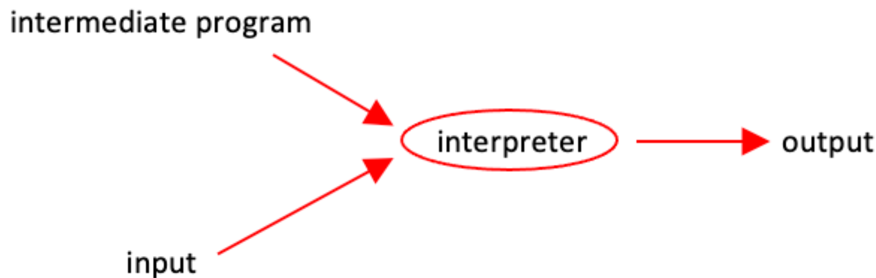
walk the data structure and *generate machine code* to do “the obvious”



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

AND

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 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 a microcoded processor (interpretation)
or micro-ops on a modern x86 (JIT translation)

many compiled languages have interpreted pieces
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.

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

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).

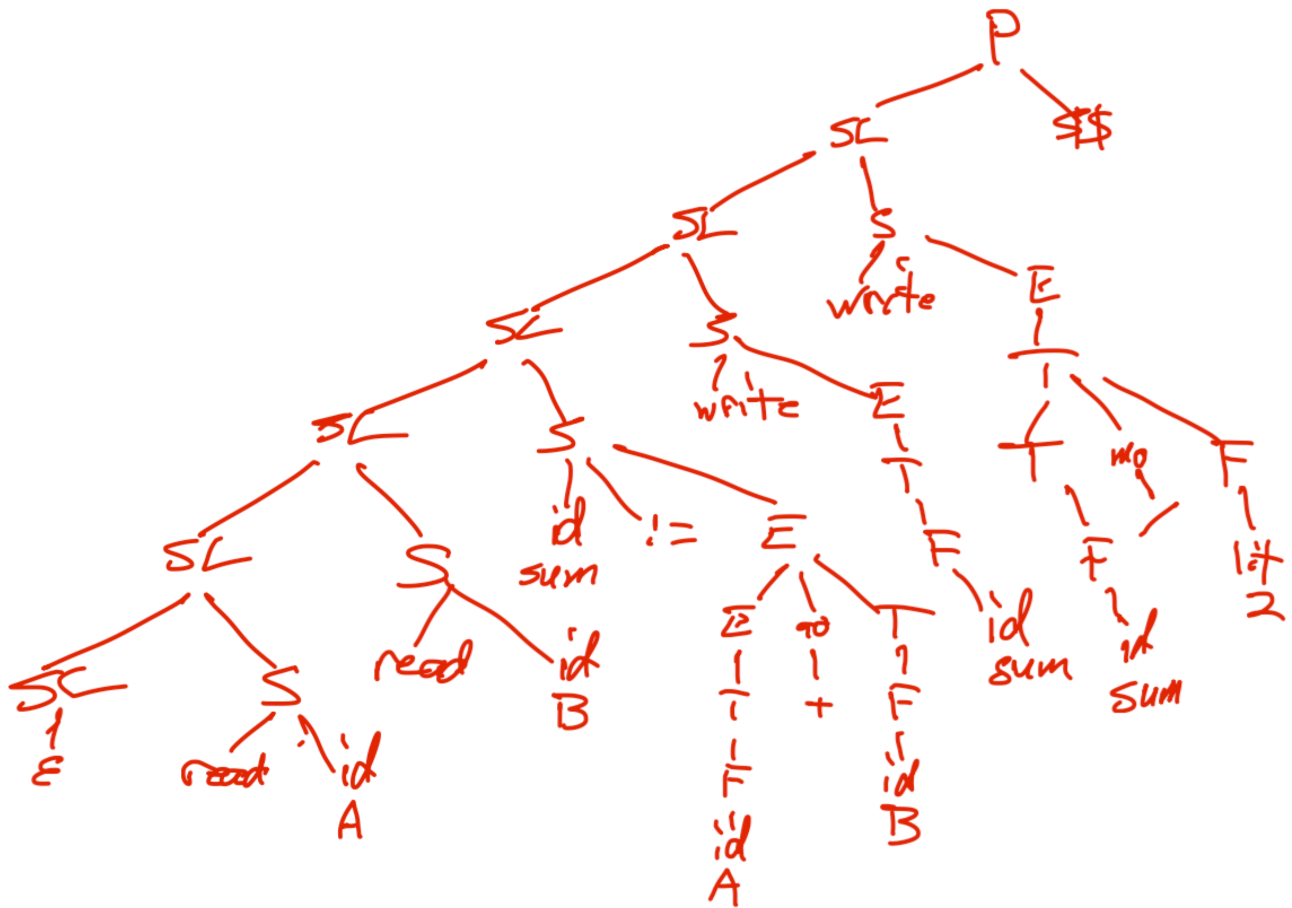
[ For those who've had CSC 280 or its equivalent, scanning is recognition of a regular language, e.g. via DFA; parsing is recognition of a context-free language, e.g. via PDA. If you don't know what that means, don't worry; we'll get to it. ]

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).

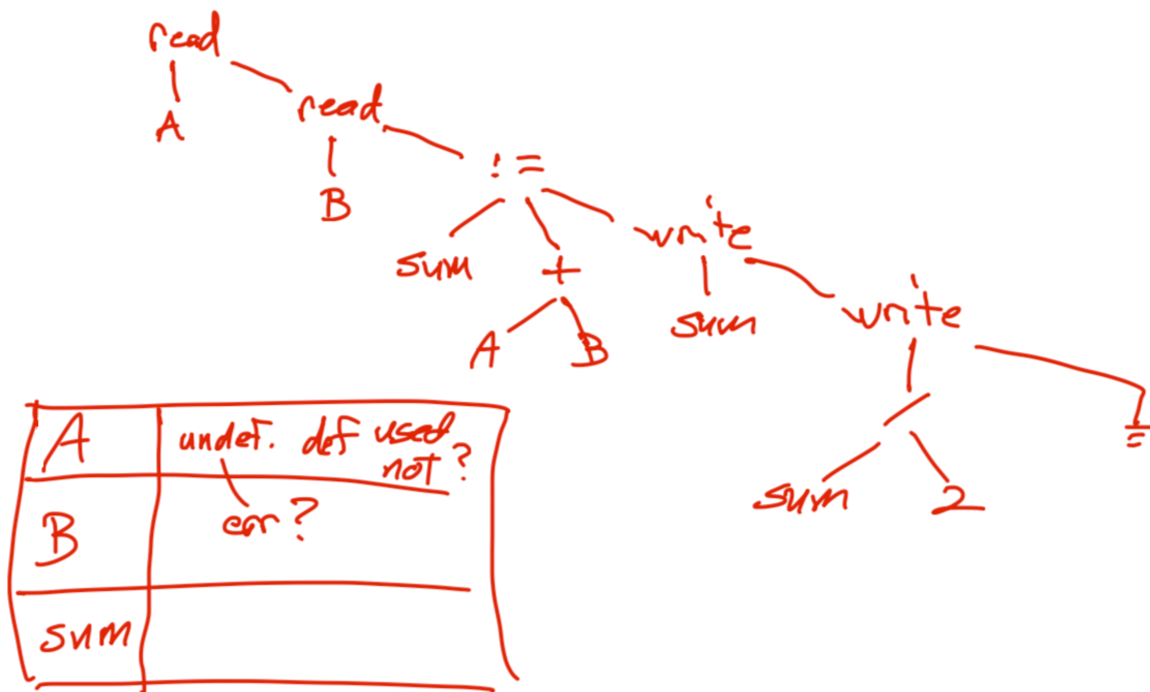
For brevity, I'll say "parse tree" instead of "concrete syntax tree"
and "syntax tree" instead of "abstract syntax tree" or AST.

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:




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.