CS 131 Contents

- Bugs in Grammars
- Ambiguous Grammars

Lecture 7 (pg 52)

- Parsing
- JIT
- Byte code
- Java and Inheritance

Lecture 8 (pg 61)

- Inheritance (Classes, Interfaces, Abstract Classes)
    - Object Class (root of all classes)
- Java Multithreading Model

Lecture 9 (pg 71)

- Exchanger class
- Synchronization and Concurrency
- Exchanger, Semaphores, CountDownLatch, CyclicBarrier
- Java Memory Model
- Volatile
- Types
- IEEE Floating Point

Lecture 10

- Intro to Logic Programming
- Prolog
    - Syntax
- Predicates
    - Sort
    - how does prolog get an answer
- syntactic sugar

Lecture 11

- Prolog arithmetic
    - piano arithmetic
- Prolog execution model
- Unification

- Trouble with unification

Lecture 12

- Grammars with Prolog
- \+
    - closed world assumption
- Debugging
- Reverse list
- Propositional Logic
    - implication
- Predicate calculus
- Resolution principle
- Scheme intro

Lecture 13

- Scheme
    - syntax
    - cons to build data structure
    - and or , short circuit eval
    - named let
- Errors in scheme

Lecture 14

- Memory Management (start)
- Stack allocation
    - static and dynamic chain
- Continuations

Lecture 15

- Types
    - Polymorphism
    - overloading
    - coercion
    - templates and generics
    - Wild cards
- Duck typing
- Names and bindings

        - namespaces
            - Java

Lecture 16

- Heap memory management
    - garbage collection
    - mark and sweep

Lecture 17

- Error handling
- Parameter passing
- Semantics
    - operational
    - axiomatic
    - denotational

Lecture 1 UCLA CS 131 lecture 2022-02-03

Let’s start with a quiz (does not count for grade!).

A little problem - solve in some real-world programming language you know well enough to write code.

Standard Input: A text file: finite sequence of ASCII characters (1-byte characters), 0-127 decimal (top bit in 8-bit byte is always zero).

Standard Output: Concordance: a list of all words in the input, each preceded by a count of the number of time it appears, sorted in decreasing order of count (if ties, order doesn’t matter). “word”: a nonempty maximal sequence of ASCII letters [a-zA-Z]+ “nonsense” is a word, but it doesn’t contain “no”. “xyzQRrewe” is a word.

Example input:

```
   Four score and seven years ago,
     and ago

   and more score four?!?!
```

Example output:

```
    3 and
    2 ago
    2 score
    1 Four
    1 four
    1 seven
    1 years
    1 more
```

Solution: `tr -cs a-zA-Z '[\n*]' | grep . | sort | uniq -c | sort -rn`

The story behind this problem.

Professor Donald Knuth (Caltech / Stanford, Turing award) wanted to write *the* book about computer programming: “The Art of Computer Programming” Unfortunately the field grew faster than he could write the book. 1 volume -> 7 volumes (3.2 written so far) volume 4A is the latest

Volume 1 (1969) published by Addison-Wesley
  did good job of publishing math + code + text + illustrations

Knuth wanted better typography

So as a side project: Knuth wrote a program called TeX
He chose the language Pascal to write it (1970s most popular language for teaching CS). So he wrote a file:

tex.pas - source code for TeX, written in Pascal

pasc tex.pas -o tex (translates TeX source code to TeX executable)

tex knuth-vol1.tex -o knuth-vol1.pdf

knuth-vol1.tex:

```
\begin{book}
\begin{chapter}
  \p
Here's the first sentence of the book.
Here's an equation: $E = mc^2$.
\end{chapter}
\end{book}
```

Knuth wrote a book about TeX. The TeXbook. (Wrote it using TeX.)

tex texbook.tex -o texbook.pdf

It's widely used today for CS math physics papers.
(In LaTeX form.)

Problem with this technology:

We have two files:
tex.pas (source code for TeX, written in Pascal)
texbook.tex (source for TeX book, written in TeX)

They go out of sync.
Knuth would add a feature to tex.pas, or make a fix.
Forget to update texbook.tex appropriately.

Bad news to see this:

```
/* If you change this function, don't forget to also
   change foo.java to be consistent!  */
```

Knuth attacked this problem with another idea "Literate Programming".

```
   He created a single file:   tex.tangled
                                    Contains both the Pascal source
code
                           and the TeX documentation,
                           together.

      begin
        (* \begin{paragraph} The $i$ variable is very important.
\end{para} *)
        var i:int := 27;
      end

   tex.tangled
      -> textbook.tex
      -> tex.pas
```

Literate programming is quite commonly used.
For example: the standard Java library is written this way.

Knuth marketed this idea by writing a short paper illustrating it, and publishing this in CACM (computer scientists read this a lot) an 8-page paper, say.

In this paper (edited by M. D. McIlroy, Bell Labs) Knuth took on McIlroy's challenge to do a literate program for our quiz problem. He wrote a (program in Pascal + description in TeX) tangled together, and used his software to generate the camera-ready copy. It's good work - new data structure hash trie to solve this problem and it covers literate programming.

Afterword by M.D. McIlroy said, "OK, but ..." (his symbol '|' for pipes)
Here's McIlroy's solution:

```
tr -cs a-zA-Z '[\n*]' | grep . | sort | uniq -c | sort -rn
```

```
tr
  -c complement of the set
  -s squeeze together adjacent outputs

uniq
  -c count lines

sort
  -r reverse order
```

```
  -n numeric sort


Which solution is better, Knuth or McIlroy's?

  Knuth's is wayyy more efficient.
  McIlroy's is wayyy simpler and more understandable.

McIlroy's sort of approach can often win nowadays.

So: we need to pick the right notation for the problem you want to
solve.
How do you choose the right notation?

Also: how do you *develop* the right notation?
  I.e., invent a programming language!
  This happens more often than you might think.
  for special-purpose jobs
    (even one-off jobs)

You'll need practice with several different notations.
In this class, we'll do:
  C/C++/sh (I assume you know)
  Java   (popular
  Python   beasts)
  OCaml   influential functional programming
  Scheme  AI/symbolic functional programming
  Prolog  AI/symbolic logic programming
  misc  assignment at the end

Rust/Go/lots of others

Categories of programming languages
  several ways to categorize
  compiled vs scripted (one way to do it, perhaps not best)

  3-way categorization

     imperative (C/C++, Java, Python, ...)
       a program is a series of commands intended for the computer
       you build programs by S1; S2; S3; S4;
          ';' is the most important way to glue together programs

     functional (OCaml, Scheme/Lisp, F#, Scala?, ...)
       a program is a set of functions to be called by whoever wants
```

to call them
you build programs by f(g(x), h(z)) by calling functions
(no assignment statements)
(variables are like math: their values never change once set)

Q. can you program functionally in imperative languages?
A. Yes, by avoiding assignment statements (and other
statements involving *side effects*
side effect - change in state of the program or of the environment
) And I've seen that done - A. Appel compiler construction books
comes in 3 versions: ML, Java, C.
all written in a functional style

logic (Prolog)
a program is a set of predicates (true/false statements about the
world) + a question about the world

You build programs (and questions) using logical connectives
AND OR NOT* IMPLIES
(no assignment statements)
(no function calls)

Functional programming is not just a language/notation thing.

Aside: "programming languages" -> "programming notations"

Sample problem at Google in the 1990s

big backend queries running for hours or days in distributed systems
for advertisers
C++ code for efficiency
The networking/scheduling was the bottleneck!
To fix this: functional programming to the rescue!
Idea: MapReduce
map + reduce operations are standard in OCaml etc.
map means apply function to every element, get a resulting data structure
reduce - reduce a big data structure to a smaller one
both of these are nondestructive

Lecture 2 UCLA CS 131 lecture 2021-01-05

general language design issues
functional programming
(syntax)

How can we tell a good job when we're designing a language?
   using a language?
   learning a language?
This can help you pick a good language.

* orthogonality (math: orthogonal axes - all at right angles to each other)
                 (all independent of each other)
            (If you choose an x value; this doesn't affect y or z choices)
   In programming languages:
     choice of one particular aspect of a feature
     should not affect other choices
   For example:
     In C/C++
       1 - functions returning values
       2 - values have types
      If they were independent axes, then a function should
      be able return any type of value:
         int long char*

        Programmer 1:
        typedef XXXX t;   // Programmer 1 chooses XXX.

        Programmer 2:
        t f(int);

        Counterexample showing why this is not orthogonal.

           typedef int t[10];   // t is an array of 10 ints

           t f(int);   // This is an error: functions cannot return arrays.
                    // Put in for efficiency.
                 // The usual answer is, "Return a pointer instead".

           typedef struct { int a[10]; } u;   // u is a struct containing

```
                                        // an array.
          u f(int);   // This is OK.

* efficiency - you want your programs to be efficient
        time (CPU time, or real time)
      space (RAM, or virtual memory)
      I/O (access to local storage)
      network usage (DNS lookups, say)
      energy (for IoT apps)
      ...
    This is important in an indirect way - it's the implementation
that counts.
    We want languages that allow for efficient implementations.

* Simplicity
    simple languages are
      easier to explain
      easier to implement

* Convenience
    It should be easy to write code that you write a lot.
       (++i) short for (i = i + 1), or for (i += 1)
    Convenience means you've added features to the language,
       i.e., you've complicated it.

* Safety (this is a big deal in large apps, or embedded apps)
    We don't want core dumps, BSoDs, etc.
    We'll need safety checks in our implementations
      (because humans make mistakes).

         Q. When you say "in our implementation" do you mean the
         implementation of the languaage or when we write a. Program?
      A. A typical implementation strategy involves
         compiling to machine code
         linking that machine code with implementation libraries
         I mean the two things together.

        Therefore there are two major kinds of checks:
          static checking (done by the compiler before your program
runs)
          dynamic checking (done by the runtime as your program runs)
                      (either in a library, or in code the
                       compiler generated)

    How much rope will we give to the programmer?
```

C - glorified assembler (CS 33 stuff - machine language),
programmer can do just about anything, unsafe
OCaml - high level, many safety checks

* abstraction
- Make it easy to program at a higher level than the bare language.
- classic object-oriented thing

* concurrency
- Divide our program into cooperating pieces each running somewhat independently.
- this can improve real-time performance
- this can make our programs clearer

Q. why concurrency makes programs clearer? If we have a multithreading program, I think it is harder to understand and debug?
A. Yes it is - particularly in languages like C/C++
Java attempted to make a lot of this clearer,
hasn't fully succeeded.
There are newer alternatives.
This is a big problem.

By clearer, I mean:
if your app has several things to do somewhat simultaneously
e.g., a web server getting requests and sending responses.
if you do this all by hand, it's error prone
multithreading gives you one way to do it more reliably
there are other ways - notably event handling

* mutability / evolvability
- The language should be easy to improve in later versions.
- Languages either evolve or die.
- We want them to evolve without breaking existing applications.

* ...

These principles / design goals are competing; there's typically no obvious optimum.

Functional programming
a programming style motivated by two main things:

* clarity - In mathematics, functions are well studied
(at least four centuries!) and well understood.
We computer scientists should build on this
rather than reinvent the wheel.

```
int i;
     i = i + 1;  // C++ code but it's mathematical nonsense
```

Why are we abusing notation like this?
For example, in Math:
If E = F (where E and F are expressions)
then f(E) = f(F) (where F is a function)

```
        If i = i + 1;
then i < 0 == i + 1 < 0;
```

This does not work!

Q. Did functional programming come after imperative programming languages as a response to this abuse of notation?
A. Yes, it certainly came later. And this was part of the reason.
Fortran (1956?)'s designer John Backus
repented of the design later, partly because of this clarity problem.
In his Turing Award lecture, he proposed functional programming as penance.

* efficiency - parallelizability
Backus: We need to escape from the von Neumann bottleneck.

This bottleneck is caused by things like this:

```
a = b + c;
```

Q. Are there any existing languages that avoid the bottleneck?
A. None avoid it entirely, but some do a much better job.

We want to avoid loads+stores.
We want to avoid *side effects* - effects on the machine state
that come as a result of executing the program.
Changing memory is a side effect
Input/Output is a side effect
The problem is *state* itself.

Q. Wouldn't you still be beholden to the complexity of what you

can do with the number of registers?
A. The registers live in the CPU and are not part of "side effects"
as long as you don't copy them into memory.
This makes things more complicated.
GPUs sort of work this way.
Yes you still run into problems if your goal is to change the state of the registers. You'll still run into clarity problems.
Q. And don't most compilers fix this to some extent already as well?
A. Yes they do help.
They can reorder operations.
This help is not enough, in general, as compilers are not "smart" enough.
We want a more natural notation that will work well in a parallelizable environment

Brief review of functions

* A *function* is a *mapping* from a *domain* to a *range*.
sin : R -> R
< : R * R -> B
* A *functional form*, or *higher order function*, is a function who domain (or range) is a set of functions
* In our discussion, each function has exactly one argument.
* A standard way to represent a function with two arguments, is as a function with a single argument that's a pair.
* The same trick works for functions that return more than one result.
* There is a different way to represent a function with multiple arguments, called *currying* - named after Haskell Curry - he wrote papers about how to do functions in functional "programming" and hated all the subscripts f(x1, x2, x3), so he simplified his papers by saying all functions take one argument, and you curry multiple-argument functions by turning them into higher-order functions:

/ : R * R -> R don't do that.
/ : R -> (R -> R) This is currying.

f = /(3) Gives a function from Reals to Reals
f(5) = 0.6

As a notational convenience, since every function
has one argument, we can drop the arguments.

```
  f = / 3
f 5 = 0.6
```

Q. wait shouldn't this be 5/3 ? or am i understand this backwards
A. The tradition is to curry on the first argument.
That's the tradition that ML, OCaml follow.

* *referential transparency* - the notion that it's trivial to figure out what name refers to.

C++ code that violates referential transparency.

```
   x = f(y);
z = w;
return f(y);
```

should be equivalent to this:

```
   x = f(y);
z = w;
return x;
```

But this doesn't necessarily work in C++.
If z is a global variable, and f uses z to decide what to return.

If we have referential transparency:
- we can use math principles, substituting equals for equals.
- Compilers can generate faster code, by caching results in registers.

To get referential transparency, we must give up side effects,
no more assignment statements!

OCaml vs ML
they're different languages with different syntax
list in OCaml [1; 3; 19]
list in ML [1, 3, 19]

OCaml is pickier about floating point

```
1.5 +. 3.7
```

```
     ML: 1.5 + 3.7

   OCaml:   0 < i && i < n
   ML:      0 < i and also i < n

OCaml basic properties

 * Static type checking (compile-time type checking)
      like C/C+/Java

 * You don't need to write down types in your program for each
declaration etc.
      like Python, Scheme, ...
    OCaml does *type inference* to figure it out statically.

 * No need to worry about freeing storage.
      It has a garbage collector.

 * Good support for higher-order functions.

Q. OCaml is interpreted, right?
A. Yes and no. I'm using the interpreter,
   it has two compilers: one into bytecode, the other into native
(machine)
   code.
   For byte code see next week.

Q. when we write our hw1.ml, do we need to include any ;;'s?
A. No, it's just part of the REPL (Read Eval Print Loop)


# 3 + 4*5;;
- : int = 23
Name : Type = Value

Q. Is "int * int" analogous to R x R?
A. Yes, * means multiplication of integers, also means cross product.

Q. can you index the tuple that's returned?
A. No.  Tuples are not arrays.
      T[i]   type is not well-defined

Q. Do numerical data types like ints have fixed lower and upper
bounds? Or do they adjust according to the value like in Python?
A. Try it!
```

Q. No == for equality?
A. OCaml uses the math notation: = stands for equality.

= is a curried function:
  'a -> 'a -> bool
  'a -> ('a -> bool)

Q. would casting be considered a side effect?
A. Simple casting used in C - no, it's just conversion
   of a value from one form to another - no side effects.

```
# let cons (x, y) = x :: y ;;
val cons : ('a * ('a list)) -> ('a list) = <fun>
```

This is a bad way to do 'cons'.
We'll do it better next time, via currying.

Q. Why did you have to write -3 with parentheses around it earlier but without parentheses in the list?
A. precedence.

Q. why does typeshowing not work with ::?
A. :: is not a function in the usual sense, it's a constructor (later)

Q. Why do you have parens around the parameters of cons? The code I saw online didn't do that and it seems to work just fine without that
A. The other code you saw was curried.

Lecture 3 UCLA CS 131 lecture 2022-01-10

More on functional programming
  Currying
  Recursion
  Pattern matching
  Recursive types
Syntax

Last time:

```
  let tcons (x, y) = x::y     (* tupled cons *)
  cons: 'a * 'a list -> 'a list
```

  syntactic sugar for:

```
   let tcons = fun (x, y) -> x::y
   cons: 'a * 'a list -> 'a list = <fun>
```

    let NAME = VALUE
      NAME - an identifier
      VALUE - an expression that is evaluate, to figure out what
         to bind NAME to
     NOT an assignment statement in the C/C++/Java/Python sense.
     NAME won't change later.

```
       let x = (27 + (let x = 19 in x + 5))
       in x / 2
```

   We must know x and y before calling tcons.

This time:

```
   let ccons x y = x::y
   ccons: 'a -> 'a list -> 'a list = <fun>>

   ccons: 'a -> (('a list) -> ('a list)) = <fun>

   let ccons42 = ccons 42
   ccons42: int list -> int list

   let foo = ccons42 [19; 12]
   foo: int list = [42; 19; 12]
```

   Q. Do x and y have to be the same type? To make a list?

A. Whatever type 'a that x is, y must be of type ('a list).

Q. Is the second definition of ccons a curried function?
A. Currying is the mathematical device of turning a multiple-arg function into a single-arg function that returns a function that cconsumes the remaining arguments. It's chainable.

Q. can you insert into second parameter rather than first in ccons42?
A. ccons42 is hardwired - the 42 is specified, it's too late to curry
it in the opposite way. Reverse currying will take more work.

```
let mylist = [20; 22]
let cconsmylist i = ccons i mylist
```
works, but it's awkward.

Q. How does is ccons42 of type int list -> int list but we pass it just an int.
A. We passed an int list to ccons42.
cconsmylist 13 yields [13; 20; 22]

Q. Is ccons an inbuilt function or one we have defined?
A. It's one we defined.

Q. What is i in this example?
A. It's the argument to the cconsmylist function.

Q. Would it be correct to conceptualize multi-argument functions as templates for curried functions that are built from them?
A. I wouldn't call them templates (not in C++ sense)
There is some commonality here.

```
let tccons (x, y) = ccons x y
let ctcons x y = tcons (x, y)
```

A tangent: let's write a converter t2c_conv from tupled to curried functions

```
let t2c_conv tf = (let cf a b = tf (a, b)
                   in cf)
let c2t_conv cf = ( ... )   (* fill in the blank *)
```

Q. why we want to use tangent?
A. A general problem in writing code, is that you want to use a module with an API X. But the calling code is written

to a module with a different API Y.
OCaml makes it easy to to glue together modules in this situation,
not by special purpose code
but by writing a converter function that you can reuse
to glue together code.

Part of functional programming is making it easy
to glue together code. You'll do this in HW2.

In ML, function application is left associative

```
   f a  b  c  d
(((f a) b) c) d
```

The -> operator on function type is *right* associative

```
t -> u -> v -> w -> x
t ->(u ->(v ->(w-> x)))
```

Q. Isn't a better way to do consMyList "let consMyList x y = y::x", so you don't have to do the reverse currying?
A. Yes.

Q. could i write something like this for n arguments?
A. Like, for tuples of any size? Not here, no.

Q. Can you do a sample run of your converter code?

Q. what is the performance cost of calling a function? Is it often worth inlining things for performance as one might in C/C++?
A. Let's not worry too much now. Don't be O(2**N) in your homeworks.
We'll talk about this in week 8?

Q. Follow up - do we pay for stack frames or can we do something simpler now that we don't have to worry as much about locals and things?
A. A bit of an advanced question; but I will mention something. You can't reasonably curry in C or C++, which may give you a hint.

Let's try to curry in C.

```
typedef struct il { int v; struct il *next; } *intlist;
```

```
        typedef intlist (*i2if) (intlist);  // i2if : int list -> int
list

        static int saved_i;

        static intlist
      auxiliary (intlist l)
      {
        intlist r = malloc (sizeof *r);
        r->v = saved_i;
        r->next = l;
        return r;
      }

        i2if
        ccons (int i)
      {
        saved_i = i;
        return auxiliary;
      }

       This sort of works, but not really:

        i2if f1 = ccons (27);
      i2if f2 = ccons (19);
      f1 (NULL) returns a list [19]!
```

Q. Can you summarize your C/C++ example (i.e. the point of the example?)

A. If you try to curry in C, the problem is that when ccons returns,
its locals have vanished (popped from the stack);
but you don't want that: you want the function that
the curried function returns, to remember those values.
C/C++ throws them away! Ouch!

Q. Is it because that there is global state we can't do functional programming in C?

A. Global state does make things harder (because side effects make functional programming harder), so the point is we shouldn't need global state to curry; it should "just work".

Q. Was the assumption that we made last class that all ocaml functions take in one argument incorrect? Does ocaml actually interpret functions with multiple arguments in this curried higher

order composition way?
A. It was correct: curried functions all take just 1 argument; it's just that they return functions that take later arguments. (OCaml can optimize away some of these internal function calls.)
f 3 12 19.0

Q. In the f a b c d, t u v w x example, can you show how the variables correspond to the sets? IE elaborate on the currying

A. Suppose f: t -> u -> v -> w -> x
a: t
b: u
c: v
d: w

Then: f a : u -> v -> w -> x
f a b : v -> w -> x
f a b c : w -> x
f a b c d : x

Q. Is it true that in OCaml all curried functions are higher order functions but not all higher order functions are curried?
A. Yes.

Q. In C/C++, recursion is often much less efficient. So we would prefer a for loop (i.e. for something like factorial). Is this problem present in OCaml, that recursion is less efficient? If so, why, and if not, why not?
A. Yes. For now let's not worry. Later we'll see how to optimize much recursion into loops.

Q. What's a higher order function that isn't curried? What would that look like?
A. A curried function is : int -> (int -> int)
A non-curried but higher order function is : (int -> int) -> int

Curried functions are also typically defined via syntactic sugar

```
let ccons x y = x::y

let ccons = fun x y -> x::y

let ccons = fun x -> fun y -> x::y
let ccons = (fun x -> (fun y -> (x::y)))
```

At the low level, functions are nameless (just as ints are nameless)
but you can give them names via ‘let’ or by making them named parameters.

Tupled functions are the same way:

```
let tcons (x, y) = x::y
let tcons = fun (x, y) -> x::y
let tcons = fun a -> match a with
                        | (x, y) -> x::y

let tcons = (fun a -> (match a with
                         | (x, y) -> (x::y)))

match EXPR with
  | PAT1 -> EXPR1
  | PAT2 -> EXPR2
  | ...
  | PATn -> EXPRn
```

Q. So this syntactic sugar helps us avoid Lots of Irritating Silly Parentheses?
A. Yes! It’s syntactically more friendly than Lisp.

Q. Are you saying that the tcons def always expands to match, or is it juts equivalent?
A. Formally, the way you understand the sugared version is to understand
the low version that uses only simple primitives.

Patterns in OCaml

```
0       matches 0  (any constant matches itself)
_       matches anything (like an identifier whose value can’t be accessed)
          Each _ in your program is a separate variable whose
        value is inaccessible (because it was discarded)
a       matches anything, and binds a to the matched value (any identifier)
```

to match tuples:

```
   (P,Q,R) matches any 3-tuple such that it elements are matched
             by P,Q,R respectively (where P, Q, R are any patterns)

 to match lists:
   []      matches the empty list (it's a constant)
   P::Q    matches any nonempty list whose head (1st element) is
matched
             by the pattern P, and whose tail (2nd and succeeding
elements,
          if any) are matched by the pattern Q.

 syntactic sugar for the above
   [P;Q;R] matches any 3-element list with respective items matched by
            P, Q, R - less commonly used
   [P]     matches any singleton list whose element matches P - less
common

     [P]  ==   P::[]
     [P;Q;R]    ==   P::Q::R::[]    (:: is right-associative)
                     P::(Q::(R::[]))

   Q. What pattern matches a list in generality?
   A.  l (any identifier)
   Q. If you want a pattern that matches any list, but doesn't
      match anything that's a non-list?
   A. Don't do that. (I'm joking, but not really.)
       Pattern matching is strongly typed.
       Even single identifiers in patters will have types.
       So, arrange for the identifier to be of type X list, for some
X.
       HOWEVER - my advice is don't bother.
       That is, if you write code where you thing x is of type 'a
list,
       but OCaml says x is of type 'a, you should be happy:
       your code is more general than you though.  So let it rip!

  Some examples involving an 'a' identifier:

     match l with
       | a::b -> (b, a)
       | []   -> ([], 42)

  Q. What if an expression matches multiple patterns?
  A. The patterns are tried left to right; the first match wins
      and you evaluate the corresponding subexpression of 'match'.
```

Q. What does it mean for the head to be a and the rest to be b? A and b are not defined.
A. They are defined by ‘match’.
The ‘match’ expression disassembles a data structure and binds its local variables to the parts that it found. ‘match’ takes apart values (nondestructively) and gives you references to the values’ components

```
:: : 'a -> 'a list -> 'a list
 @ : 'a list -> 'a list -> 'a list
```

A prototype function (which I’ll write in OCaml today; you’ll see this later in other languages) to reverse a list (nondestructively)
create a list in the reverse order of its argument.

```
 let rec reverse = fun l -> match l with
                              | [] -> []
                      | h::t ->  (reverse t) @ h;;
```

Q. why does ocaml need this keyword ‘rec’, can it not figure it out from the function definition?

A. It could easily figure it out, except
1. Sometimes you don’t want recursion, and ‘rec’ is a safety mechanism.
2. There are sound theoretical arguments for making ‘rec’ not the default.

```
           let X = ... X ....
```

e.g.,

```
           let N = 0 * N
```

It’s philosophically troubling to define something in terms of itself. It’s also tricky mathematically. So OCaml by default doesn’t let you define anything in terms of itself. It does make an exception for recursive function, you use ‘rec’ to say “I know what I’m doing.”

Q. Why bother having :: and @ as two separate things when @ can do the job of ::?
A. @ is toooooo powerful!

```
        match l with
        | a@b -> ....
```

Q. Given list l, and when h::t is matched, should h be an 'a type?
Then how can it use @ operator, which is 'a list -> 'a list -> 'a list
A. You're faster than I was.

1. This how I learned ML back in the 1970s!
2. I haven't typed this in yet.

```
# let rec reverse = fun l -> match l with
                          | [] -> []
                  | h::t ->  (reverse t) @ h;;
val reverse : 'a list list -> 'a list = <fun>
```

Ouch! This is really a reverse-and-flatten function.
I wrote the wrong function.
And I discovered this without running the code!

This is static analysis debugging your code.

A fixed version:

```
let rec reverse1 = fun l -> match l with
                            | [] -> []
                     | h::t ->  (reverse1 t) @ [h];;
let rec reverse1 l = match l with
                        | [] -> []
                   | h::t ->  (reverse1 t) @ [h];;
```

Syntactic sugar

```
let rec reverse1 = function
   | [] -> []
   | h::t ->  (reverse1 t) @ [h];;
```

Lecture 4 UCLA CS 131 lecture 2022-01-12

Let's take a look at 'reverse' again.

```
  let rec reverse = function
    | [] -> []
    | h::t -> (reverse t) @ [h]

  let rec reverse = fun l -> match l with
    | [] -> []
    | h::t -> (reverse t) @ [h]
```

This code is O(N**2), because @ is O(N).
We want it to be O(N), despite not being able to stomp on the list.
  (*stomp* means modify the 'next' field of a linked list).

Accumulators are a standard way to accomplish this sort of thing.
  They're extra arguments designed accumulate work done so far.

We'll solve a more general problem: we'll compute the reverse of L,
  concatenated with M.

```
   let rec revapp l a =
     match l with
       | [] -> a
       | h::t -> revapp t (h::a)      (* O(1) each time through the
recursion *)
```

    O(N) overall.

      Q. If we do h::a, aren't we appending the h to the front? Would
      it make it reversed?
      A. Yes, we are prepending a single item h to the accumulator;
      this doesn't change a; it merely builds a slightly longer
      list

      Q. If we cannot modify a list, doesn't every (::) operation
      create a new list based on a present list, and doesn't the
      new-list-creation causes O(n) time for each operation?
      A. No, because a::b doesn't change b; it merely creates a
      new list element that points to b.

   Q. Why not have one argument and directly do prepend? Something
like

```
let rec rev l = match l with
   | []-> []
   | h::t -> (rev t)::h
```

A. Lists can be built cheaply only at the front,
types don't work for this formulation.

Q. What happens if you later write to b? Does that affect the result of a::b after the fact?

A. It would, if you were allowed to write to b. But you're not allowed.

This is one of the virtues of functional programming.

Aside: Traditional JavaScript has this problem.

Nowadays many JS programs are written in a more functional style; objects once set up should never be changed.

Q. or why is double-ended linkedlist break functional programming

A. More generally, watch out for data structures with cycles, (even in imperative programs they can cause loops); you can't create them in fp; so no loops!

```
let rec revapp l a =
  match l with
    | [] -> a
    | h::t -> revapp t (h::a)      (* O(1) each time through the recursion *)

let reverse l = revapp l []
```

```
let rec revapp a = function
  | [] -> a
  | h::t -> revapp (h::a) t      (* O(1) each time through the recursion *)

let reverse = revapp []
```

Q. You can just ignore the "function" keyword right?

A. "function" is short for "fun xxx -> match xxx with"

Q. Why don't we define revapp inside reverse with "let...in" rather than keeping it separate?

A. Perhaps we'd like revapp functionality elsewhere.
BUT Perhaps you're right.

Q. wait how come there is no "l" in "let rec revapp a"
A. The 'function' keyword means arg is nameless;
that's a feature!

Q. when and how would you actually need that argument?
A. Supppose you write this:

```
fun l -> match l with
        | [] -> [47]
      | h::_ -> h::l
```

is slightly more efficient than:

```
fun l -> match l with
        | [] -> [47]
      | h::t -> h::(h::t)
```

```
function
        | [] -> [47]
      | h::t -> h::(h::t)
```

Q. Is there a good reason to use function rather than 'arg ='
syntax.. I mean aside from looking cool in front of your
friends?
A. Isn't looking cool a good thing?
In OCaml we often use nameless functions
fun x -> x+1
In OCaml we often use functions with nameless args
function | [] -> true | _ -> false
Understandability is the goal.
Don't pollute your program with useless names.

Let's try one more function.
Compute the minimum value in a list.

```
let rec minlist = function
   | h::t -> (let i = minlist t in
              if i < h then i else h)
   | [] -> int.max_int
```

Q. Maybe do matching with lists that have one element left?
A. It's a bit awkward

```
let rec minlist = function
```

```
    | [h] -> h
    | h::i::t -> (let j = minlist (i::t) in
                  if j < h then j else h)
    | [] -> int.max_int

  minlist : int list -> int
```

Q. Does "in" not need a named argument, because the function has a nameless argument?

Q. What large software systems are written in OCaml? Are there any million-line programs that use it? I'm struggling to see how it would be easier to design a large program with functional programming vs. object oriented programming

A. The OCaml compiler (joke)
State Street (Wall St company) uses it.
OO programming and functional programming both give you leverage in large systems, but it's different kind of leverage.
I'm assuming you know OO (C++);
I'm focusing on FP here.
OCaml is object-oriented - but I'm ignoring that here.

Q. How can you write a compiler for a language in that language?

A. Verryy carefully.
The first C compiler was written -- in assembler (I think).
THey hated it.
The wrote a C compiler in C.
THey threw the first compiler away.
The OCaml compiler was done in a similar way.
First they wrote an OCaml interpreter in some other language.

Q. OCaml seems really useful for smaller parts of other things, does it have an easy way to integrate (ffi) or something into larger programs?

A. Yes.

Q. I'm a bit confused, where is j used in the second manliest function?

A. Corrected above.

Q. The h::t divides a list into 2 parts, but I don't get how

```
       h::i::t would work, could you please explain what it does?
       A. The pattern h::i::t is equivalent to h::(i::t)
          because :: is right-associative.
        We have nested patterns here.

       Q.  Can we match l with h::[] to get the last element?
       A. No, a::b matches only the head of a list.

       Q. Could you elaborate a bit on cyclical data structures in
OCaml?
       A. I don't want us to get into that mess; I'll do that later
            in Scheme and Prolog and etc.

   minlist is only (int list -> int) - not very general

      let rec gminlist lt id = function
         | h::t -> (let i = gminlist lt id t in
                    if lt i h then i else h)
         | [] -> id

      let minlist = gminlist (<) int.max_int

     Q. Why do you need the parentheses here?
     A. Because OCaml has <, +, *, as builtin binary operators.
        Without the () we'd be comparing numbers

      let minlist = gminlist < int.max_int

Restart at 16:58

  Qs during break. (I see students have already answered many.)

  Q. would it be safe to say that parameterization be a case where
  youd want to explicitly define parameters because you'll explicitly
  *reuse* them in the function details/impl
  A. Yes, typically one has function parameters because one intends
  to use them at least once in the function body.  You can use the
  'function' keyword if you plan to use the parameter just once
  in a top-level match.

  Q. What kind of coffee is it?
  A. Peet's Major Dickinson Decaf. Decaf because my doctor told me
     not to float through the ceiling with caf.

Let's talk a bit more about types
```

```
type  mytype = int

type mytype =  | Foo
               | Bar of int
          | Baz of int * float
```

This defines a "discriminant union".

discriminant - lets you discriminate at run-time among the possibilities

In contrast to the 'union' in C/C++.

```
    typedef union {
                   struct {} foo;  // is this even allowed?
                   struct { int x; } bar;
              struct { int x; float y; } baz;
             } mytype;
```

Generic types in OCaml

```
    'a -> 'b

  'a -> 'a
```

Defining your own generic types

```
     (* Builtin but let's pretend it's not. *)
     type 'a option =
      | None
      | Some of 'a

         int option
       string option
       int option list option

     This defines None and Some in two ways.
      First, they can be used as *constructors*
           Some (x+y)
        None
        Second, they can be used as *patterns*
        match val with
           | None -> 15
        | Some x -> x + 5
```

Q. Are "None" and "Some" keywords?

A. No, but should be capitalized.

Q. What does it mean to have a type None, where there is no value?

A. One way to think about this:
OCaml doesn't have NULL (in C/C++ sense).
Instead, it has 'a option.
If you want something like a C/C++ function
that returns (struct foo *), possibly NULL,
in OCaml you return foo option instead.
This means your caller must do a match
and cannot dereference NULL.

Q. Is there a way to make None and Some generic in order to work with any 2-type union?
A. We need static checking.

Q. Could you give an example of what None and Some might be?
A. Sure, the C function getenv.
getenv ("PATH") returns either a null pointer,
or a char * string representing the value of the
PATH environment variable.

In OCaml:
getenv : string -> (string option)

```
type 'a sequence =
   | Empty
   | Nonempty of 'a * 'a sequence
```

This type acts just like 'a list does.

"type 'a list =
| []
| :: of ('a * 'a list) "

* means cross-product in types

Syntax

Classic definition of language syntax:
"form independent of meaning"

Good syntax, nonsense meaning.

"Colorless green ideas sleep furiously." - N. Chomsky

Good meaning, questionable syntax.
"Ireland has leprechauns galore." - P. Eggert
What part of speech is "galore"?
It's an adjective.

"Ireland has galore leprechauns." is syntactically incorrect English!
It's a special case.
There are a few other adjective like it "arms akimbo".
English grammar is more complicated that one might think.

"Time flies."
N V.
V N. An ambiguous sentence.
(both syntactically and semantically)

Can be trouble in English. (soldiers)
Can be useful in English. (diplomats)
Usually trouble in programming languages.

Good syntax vs bad.
1. Unambiguous
2. It's what you're used to. (Reuse math syntax, say.)
a + 1 (ADD 1 to A.) (a 1 +) (+1 a)
3. It's simple and regular. (as few rules as possible)
By regular I mean orthogonal - no special cases,
or few special cases, anyway.
It's nice for you to write any sequence of operators
and for it to just work.

```
a = b / c;
a = b /+c;
a = b /*p;
a = b +-c;
a = b ++c;
```

4. It's "readable".
Leibniz invented/discovered calculus (Newton was there too)
Newton's notation for calculus was an utter disaster!
Leibniz's was much better, long S and dx/dt is from Leibniz.
He worked at notation and he was a genius at it.
His criterion was:

"A proposition's form should mirror objective reality."
Example of applying his criterion to programming:
Comparing integers.
E < F vs E > F
write it E < F, that way textual order reflects
numeric order in the number line.

```
int fd = open("foo", O_RDONLY):
if (0 <= fd)       /* good */
  return fd;
if (fd >= 0)       /* bad programming */
  return fd;
```

5. It's "writable".
It's easy to write code; you don't have to waste a lot of time getting every detail right; it'll work quickly.
This can go too far: some languages are too writable.
APL (A Programming Language)
Hard even to read your own code! But you cdan write it fast.

6. It's redundant.
People make mistakes all the time.
A language with no redundancy means less opportunity to catch stupid mistakes.

Add a rule to C++.
When you see ';', assume enough '('s here to make your program valid.

```
i = j * (3 + k;
```

Q. Is python indentation a form of redundancy?
A. Yes. It helps catch dumb mistakes.

Tokens (and Tokenization)

Split up your program into simple *tokens* (categories of *lexemes*).

```
|int| |main| /*a comment */ |(|
 12    1                    27

|void| |)| |{| |return| |27| + |30| |;| |}|
  15   28  19    51      2   23  2   10   20
```

The tokens tell you what the program's syntax is.
Each token in general can stand for one of several lexemes.
Most token stand for just one lexeme

```
        27  (
      19  {
```

A few of them can stand for different lexemes

```
         1  main  cos   x     lots of possibilities
       2   27   30    23242342
```

Grammars.

Grammars assume you've solved the tokenization problem.

```
    token t = get_next_token ();
```

Q. So all lexemes have a single token, but tokens may refer to one or more lexemes?

A. A token is a class of lexemes.

```
   token t = get_next_token ();
     t.token might be 1
  t.value might by "main"   value is what attributes this
                              particular lexeme has.
```

Q. with English, an example of a token I think would be 'verb' but a lexeme would be 'to eat'

A. Yes.

Q. Is this related to Knuth's problem?

A. Yes, but it's a different problem!

Lecture 5 UCLA CS 131 lecture 2022-01-19

Syntax and grammars
  tokenization
  context-free grammars

Last time: tokenized a finite sequence of characters
              into a finite sequence of tokens
         (already done for you, in homeworks)
  lower-level issues
  . Character set? ASCII, UTF-8 (Unicode characters each
                            a small sequence of bytes
                     'char' means byte in C++)
       int â = 12; // This works!

  . Non-tokens in the input (discarded by tokenizer)
     - spaces, tabs, newlines (separate the tokens)
        "white space"
     - comments
       ... what if the comments are important?
           literate programming

   . numbers
         112123   123.45e-27 Just one token in C/C++
              -123.45e-27 two tokens in C/C++ (not some other
languages)
     - 112123 - 123.45e-27

        -2147483648   is two tokens in C/C++
     - 2147483648   but 2147483648 is too big to be an 'int'.
                    In C: 2147483648 is 'unsigned' (big enough to
                                                hold the number)
                  -2147483648 is also unsigned
              -2147483648 < 0   yields 'false'!
              -1 - 2147483647 < 0 works as expected

    . identifiers
        - Are identifiers case-sensitive?
          int XYZ = 1; return xyz;
          UCLA.EDU   UCLA.edu ucla.EDU
        - WHat about ambiguous-looking i18n-oriented identifiers.
           i18n - internationalization

                int o = 27;

```
          int o/*Cyrillic*/ = 28;
          return o + o/*Cyrillic*/;  // 55

                microsoft.com   microsoft.com  -- latter uses
Cyrillic

       - keywords     let let = 15 in let + 5
                       This is confusing!
                    + Many languages therefore disallow it,
                       by *reserving* keywords,
                       you cannot use them as identifiers.
                    - Programmers must know all keywords.
                      Later extensions to the language can
                      break existing programs!

                                  int class = 131;
                              Some languages have no reserved words;
                           they have keywords, but not reserved.
                           (e.g., Scheme)
```

Q. I was a bit confused at the precise definition of token and lexeme from last time. Could you please define them here?
A. Tokens can be characterized as being members of a small set.
(internally, represented by bytes, or small ints, or whatever)
We need this for grammars to work efficiently (later).

```
      abc def ghi (lots of identifiers are possible)
     asdfasfasfasfdsfasfsfasdfdsafsadfsfd
    We tell the compiler that each of these identifiers
    is the token ID.
       ID ID ID ID
    The compiler ignores which identifiers are which, while parsing.
    Associated with each ID token is *extra* info, needed
      for semantic analysis (much later).
  The *lexeme* = *token* + extra information.

       aaa = bbb + ccc ;
     ID  =  ID + ID  ;
     2   25 2  16 2  13
```

Q. How about tokens in ML? There can be thousands of them in a dataset.
A. An ML source can contain thousands of tokens, but
they typically are not *distinct* tokens.
let x = fun x -> y + 1 ....

```
        let ID = FUN ID -> ID + NUM
```

-

Context-free grammars

Quick backgrounds:
verrry simple grammars: regular expressions
A regular expression describes a simple language,
namely, the set of all matches to that regular expression.
This is not powerful enough for programming languages.
Because regular expressions can do iteration:

```
     ab*c  - iteration
   [^{}]*{[^{}]*}[^{}]*
         Our attempt at regular expression
              for properly nested braces
          a{b{c}d}e
```

We want the ability to nest: This is what context-free grammars buys us.

There are other things we might want too, and there are fancier grammar technologies to support that, but we'll stick to context-free grammars for this class.
"context-sensitive grammars" -- grammar hierarchy of Chomsky
We can do without these for most programming languages.

Definitions

token - a member of a finite (typically small) set
string - a finite sequence of tokens (can be empty)
language - a set of strings
== predicate (function returning boolean) on strings
TRUE - your string is grammatically correct
FALSE - your string has bad grammar,
it's not in the language

We want a good way to define a language.
regular expressions aren't enough to do ML, C, Java, ...

CFG (Context Free Grammars)

CFG = (finite set of tokens (or "terminal symbols"),
finite set of nonterminal symbols,
finite set of rules,
start symbol)

nonterminal symbol
represents a phrase (token sequence) in the language
in the parse tree, it can occupy an internal node
(or it can be a leaf, see later)
whereas terminal symbols must be leaves

symbol = nonterminal symbol
or terminal symbl

rule = (nonterminal symbol, finite sequence of symbols)

start symbol must be a nonterminal

Q. Why does a start symbol have to be a nonterminal? What if we want a grammar that only generates 1 symbol?
A. I suppose you're right, we could allow that without changing much, but such a grammar would be sort of boring; no rules could be used, no noterminals would be used, and only one of the terminals would be used.>

Q. How could a nonterminal be a leaf?
A.

```
S -> S a
S -> S b
S ->
```

A rule with an empty RHS is allowed!

Q. (during break) Could we speculate that the epsilon is something that's implied, but not pronounced explicitly (in natural languages)?

A. It's certainly not pronounced, since no tokens are there. Whether it's implied is a bit trickier; I'd say that in many natural languages when there's nothing there then nothing is implied (I'm handwaving here tho). Also, in natural languages the level below tokens is phonemes, not characters, and this makes the empty string a bit different (you can have tones, for example, even when you don't have "letters"). (Oh, now I'm *really* handwaving!)

A sample grammar for a data language.
Language for email.
There's a grammar for it: Internet RFC 5322

grew out of UCLA (done in Boelter Hall by Postel et al)

It's a contract for Mail User Agents + Mail Transfer Agents

One little corner of RFC 5322: Message-IDs

Message-ID: <eggert.p.r-abc$def@cs.ucla.edu>

The grammar for what you can put into that line (after the ":")

```
msg-id = "<" dot-atom-text "@" id-right ">"
id-right = dot-atom-text / no-fold-literal
no-fold-literal = "[" *dtext "]"
dot-atom-text = 1*atext *("." 1*atext)
dtext = %d33-90 / %d94-126
atext = ALPHA / DIGIT / "!" / "#" / ...    (printable ASCII characters
                                            that are not special)
```

%d33-90 means "Any ASCII character whose decimal representation is in the range 33-90"

id-right = dot-atom-text / no-fold-literal
equivalent to:
id-right = dot-atom-text
id-right = no-fold-literal

no-fold-literal = "[" *dtext "]"
equivalent to:
no-fold-literal = "[" dtexts "]"
dtexts =
dtexts = dtexts dtext

METANOTATION

X/Y here means "X OR Y" - This is metanotation, it's not something you see in a CFG rule

As an aside: why "/" not "|"?
It's because Jon Postel's terminal didn't have "|".

*X stands for zero or more occurrences of X
1*X stands for one or more occurrences of X

(X) stands for X

This a notation that stands for a CFG but it's not strict CFG.
It's called EBNF
Extended Backus Naur Form
Extended Backus Normal Form

BNF - a common notation for grammars
<msg-id> ::= < <dot-atom-text> @ <id-right> >

Whatever extension you come up with in your EBNF
must expand straightforwardly to straight BNF

Q. Is BNF standard, but EBNF up to you how you want to extend BNF?

A. BNF corresponds to CFGs in their basic form (Hw 1 & 2)
EBNF uses a few metasymbols to make it easier.
We're using "RFC EBNF", used in Internet RFC.s
Internet RFC 5234 describes RFC EBNF.
There are other EBNFs out there.

Q. So, EBNF still represents a context-free grammar right?

A. Yes, because EBNF grammar can be easily translated to BNF.
EBNF can be more convenient, but it doesn't
expand the set of languages you can define.

Q. Can we formally define a macro? (If that matters) I know what they are as far as making them, but I don't Really know what they are.

A. Yes. A macro is a function from text to text that can be explained by substituting an invocation with the macro's definiens (this is handwavy - we'll see more later).
Often language-specific.
Definition:
name stands for a definiens

Q. So X* means X or more of whatever comes next?

A. Internet RFC EBNF uses *X (normally r.e.s use x*)
to mean repetition.

```
*"a"
*atext
*("." *(atext dtext))
```

You can put any nonnegative integer in front of the *.

Q. Do we need to know this grammar, or is the purpose of going through it to understand some feature of EBNF?
A. YOu needn't memorize it! This is about EBNF; the grammar is just an example. (Open-book exams - no need to memorize!)

Q. For this class are we working with RFC EBNF or the textbook version of EBNF/
A. For this lecture, RFC EBNF. EXCEPT:

Q. So can a EBNF be parsed by a finite automata (i.e. the description of the EBNF grammar is a regular language)?
A. Good question, see the next topic!

Let's look at another EBNF form.

ISO EBNF
ISO - International Standardization Organization
it standardizes lots of things
JavaScript electrical connectors ....
ISO faced a problem: lots of standards for languages,
each used its own EBNF format.
so: let's standardize EBNF! (ISO didn't like RFC 5234,
because it's ugly "/" "|"
and because of N.I.H.)
Not Invented Here
(Prof. Sahai on YouTube zero knowledge proofs explained to USC)

ISO EBNF examples
nonterminal symbol They can contain spaces!
"terminal symbol"
'terminal symbol'

[option] zero or one occurrence of 'option'
{repetition} zero or more occurrences of 'repetition'
(grouping) grouping
(*comment*)

N*X N times repetition
A-X A, but not X (X must be a set of tokens)
A,B A followed by B ("," is an explicit symbol for concatenation)

```
A|B   A or B
LHS = RHS;    grammar rule.
```

Here's an example of ISO EBNF. This is the grammar for ISO EBNF,
written in ISO EBNF.
(The committee is eating its own dog food.)

The start symbol is 'syntax'.

```
  syntax = syntax rule, {syntax rule};
syntax rule = meta id, '=', defns list, ';';
defns list = defn, {'|', defn};
defn = term, {',', term};
term = factor, ['-', exception];
exception = factor;
factor = [integer, '*'], primary.
```

Q. Wait why does it explain higher precedence?
A.

```
a = b, c | d, e;
  defn | defn

a = b   , c | d, e;
  term, term , term    XXXXX not possible
```

Q. The lower the operator in a parse tree, the higher precedence?
A. Higher-precedence operators are applied first,
so the must be lower in the parse tree (when root is at the top).

```
syntax rule = meta id, '=', defns list, ';';

    meta id   = d e f n s       l  i  s    t ;
```

So, is ISO EBNF defined in terms of itself?!?!?
Any such definition is questionable.
No - they defined ISO EBNF twice, once in English,
once in itself, as a good example.

Lecture 6 UCLA CS 131 lecture 2022-01-24

syntax and grammars
practical aspects (syntax and compilers)
parsing (hw 2)
types (probably not)

last time
  different syntax notations

one more notation today:
  syntax diagrams / syntax graphs / syntax charts / railroad diagrams
    for when even EBNF is not enough simplification of true syntax
    we want something simpler than EBNF

   A syntax chart is a blueprint for your parser's source code,
     if you use a particular style.
   Your programs have bugs -- your grammars can have bugs too.

   Q. Would a non-terminal leaf be a bug?
   A. That depends on how you define a parse tree.

Debugging grammars.
What can go wrong with grammars?

 * Nonterminal that's used but not defined.

```
S -> S a
S -> b
S -> T c   <-- this rule does not affect the language
                defined by the grammar
               It's at best a useless rule.
```

 * Nonterminal that's defined but not used.

```
S -> S a
S -> b
T -> c   <-- These two rules
T -> d   <-- don't affect the language.
```

 * The above notions apply recursively.

```
S -> S a
S -> b
```

```
      S -> T c   <-- These three rules
      T -> T d   <-- don't affect the language.
      T -> e T   <--
```

* How much detail about the syntax should be specified by the grammar?

 * Grammar doesn't represent the whole syntax : it's too generous
      it allows token sequences that are grammatically incorrect.
 vs
 * Grammar is too specific : it captures so many details of the language
      that it's overly complicated and hard to follow

Here's a grammar for C++:

```
program ->
program -> program TOKEN
```

TOKEN is any C++ token

In English, this problem can be represented by a simple grammar for very short English declarative sentences.

Tokens = { DOG, CATS, BARK, MEOWS, GREEN, BLUE, LOUDLY, QUIETLY. }

Grammar:

```
   S -> NP VP .
   NP -> N | Adj NP
 VP -> V | V Adv
   N -> DOG | CATS
   V -> BARK | MEOWS
 Adj -> GREEN | BLUE
 Adv -> LOUDLY | QUIETLY
```

GREEN DOG MEOWS QUIETLY.
BLUE GREEN CATS BARK LOUDLY.
ok so far, BUT

GREEN CATS MEOWS. CATS is plural, MEOWS is singular.
We don't want to allow this! It's not English.

We can fix this bug by complicating the grammar.

```
      S -> SNP SVP .
         | PNP PVP .
      PNP -> PN | Adj PNP
      SNP -> SN | Adj SNP
    PVP -> PV | PV Adv
    SVP -> SV | SV Adv
      PN -> CATS
      SN -> DOG
      PV -> BARK
      SV -> MEOWS
    Adj -> GREEN | BLUE
    Adv -> LOUDLY | QUIETLY
```

This works, BUT the guts of the grammar grew by a factor of 2.
It's sorta OK here, but suppose we needed to cover:

```
  number + gender + tense +
   sing    masc     past
pl      fem      present
        neuter   future
     2      3        3
```

2*3*3 expansion!
Grammar grows exponentially on the number of attributes.

Q. Is there a complete and correct grammar for the English language? IE could you write it down?
A. No.

Q. What about grammar check?
A. It doesn't really rely on this kind of grammar.
CFGs are not nearly enough for this.

Corollary: CFGs don't capture *every* syntactic constraint of every programming language.
They're enough to write parsers, but rest of compiler will have to do further checking.

Two sorts of things CFGs don't catch:

* type checking.

```
      int i;
    int f (char *);
    ... f(&i) ...
```

* scope checking

```
{ { int i = f(x); if (i) g(i); } return i; }
```

A trickier one that CFGs won't catch:

```
typedef int pid_t;
pid_t p = 27;  // not a syntax error
xxx q = 27;    // a syntax error
```

Q. Could we expand the grammar to handle type checking, but we just prefer not to in order to avoid complexity?
A. If there's a finite number of types, yes in theory, at least in some languages. But it's not practical as most languages have an infinite number of types.

Q. Functional languages check types though, right?
A. OCaml does, yes. But there are functional languages that don't do static type checking (e.g., Scheme).

Q. How does a compiler support the parser by checking the stuff you wrote down?
A. That's part of : how does a parser fit into a compiler?

Q. so are you saying that when we make a grammar, generally, we don't want it to admit precisely every string we want in the language and reject any string that we do not want in the language? I don't see the point for subtlety / grey lines -- a grammar for C should admit a string iff it's a valid C program, right?
A. Yes, that would be the goal, BUT it's not practical.

```
int i;
char *p = &i;
```

Q. So does a CFG check syntax, while a compiler check semantics?
A. Mostly. In some languages yes; to some extent it's a tautology.

* Ambiguous grammars

Your grammar is too generous in a particular way:
it allows two or more ways to parse the same sentence.

```
E -> E + E
E -> E * E
E -> ( E )
```

```
E -> ID
E -> NUM
```

```
2 + 3 * 4
```

We'll fix this by complicating the grammar (hopefully not too much).

fix precedence problem

```
E -> E + E
E -> F
F -> F * F
F -> ( E )
F -> ID
F -> NUM
```

Q. But since we give precedence to rules that are higher on the list, don't we give preference to 2+(3*4) already?
A. HW 2 parser does prefer earlier rules first, so in that sense yes.
HW 2 parsers do not tell you when the grammar is ambiguous, and they'll just return the first parse that they find, even if there's some other way of parsing the sentence.

Q. can other parts of the compiler rearrange the tree to their liking? I'm thinking of something analogous to a movement in a linguistics syntax tree if you're familiar
A. Yes, parse trees are commonly data structures, and the compiler can mess with them whichever way it likes. A very common way is to generate "abstract parse tree", a simplified version of the concrete parse tree, it discards nodes present only to avoid ambiguity problems

Q. How would we eliminate ambiguity due to different ways of associating operations with the same operator? i.e. can we enforce left-associativity or right-associativity with a CFG?
A. Yes. +, * left associative, ** right associative

```
a+b+c == (a+b)+c        a**b**c  ==  a**(b**c)
```

```
E -> E + T
E -> T
T -> T * F
```

```
T -> F
F -> G ** F
F -> G
G -> ( E )
G -> ID
G -> NUM
```

Q from PE. Won't this sort of added complexity lead to combinatorial
explosion, like attributes did?
A. No, as ambiguities tend to be independent from each other
and they can be resolved independently.
precedence and associativity are about the worst case
in terms of making the grammar grow.

Q. Are there other ambiguities other than precedence and associativity?
A. Yes, and this can be a deep topic - ambiguities can exist but not be obvious.

for example: a simple grammar for C/C++ statements

```
stmt:
  ;
  expr ;
  'break' ;
  'continue' ;
  'return' ;
  'return' expr ;
  'if' ( expr ) stmt
  'if' ( expr ) stmt 'else' stmt
  'while' ( expr ) stmt
  'do' stmt 'while' ( expr ) ;
  'for' ( expr-opt ; expr-opt ; expr-opt ) stmt
    ...

expr-opt:
  expr
  /* empty */
```

As an aside:
Why this:

```
        'return' expr ;
```

and not this:

```
'return' ( expr ) ;

if (i < 0)
return i;
```

because parentheses are not needed to prevent ambiguity.
expr cannot contain ‘;’, so ‘;’ ends the expression.

Why this:
'while' (expr) stmt
and not this:
'while' expr stmt
because parentheses are needed here to avoid ambiguity.
C++/C would be ambiguous if the parentheses were omitted.

```
while p*q;
while (p) *q;
while (p*q) ;
```

Q.Why this:
'do' stmt 'while' (expr) ;
and not this:
'do' stmt 'while' expr ;
Let’s design a new language: C131
It’s like C except the grammar is relaxed a bit for do-while as above.
C131’s grammar is unambiguous!
A. consistency and beauty (in the eye of Kernighan & Ritchie)

Q. Can you prove a grammar is or is not ambiguous? What type of proof is required for that?
A. It’s relatively easy to check a proof that a grammar is ambiguous:
give an example sentence that has two parses
The reverse is harder, though. See theory class.
In practice for practical languages, it’s not that hard usually.
A. precedence,
B. associativity
C. see above, in the grammar for C statements

Q. Is ambiguity always bad, or is there a scenario where two different parses of a statement are equivalent?
A. It’s typically bad.
Counterexample: + means string concatenation
a + b + c can be evaluated either way (let the implementation

choose, as an optimization)

```
stmt:
  'if' ( expr ) stmt
  stmt1

stmt1:
  ;
  expr ;
  'break' ;
  'continue' ;
  'return' ;
  'return' expr ;
  'if' ( expr ) stmt1 'else' stmt
  'while' ( expr ) stmt
  'do' stmt 'while' ( expr ) ;
  'for' ( expr-opt ; expr-opt ; expr-opt ) stmt
     ...

expr-opt:
  expr
  /* empty */
```

Q. Would this mean you always need to have an else following an if (expr) stmt1?
A. Yes.

How this fits into the compiler

```
   #include <stdio.h>
   int main(void) { return !getchar(); }
preprocessor ->
  extern int getchar (void);
  int main(void) { return !getchar(); }
lexer ->
  token sequence
parser ->
  parse tree
semantic analysis (type checking, name checking) ->
  decorated parse tree (nodes have more attributes)
     expr - what's its type?
code generation ->
  assembly language
assembly ->
```

```
  object code (binary)
linker ->
  executable
loader ->
  running program
```

It’d be nice to have something simpler --

Lecture 7 UCLA CS 131 lecture 2022-01-26

a bit more on parsing
implementation strategies
Java (concurrency)

about parsing again:
revisit Homework 2
take a look at the hint code (again)

Three basic problems in writing a simple parser.

1. Recursion in the grammar.
2. Alternation (OR)
   In a grammar:
   T -> T a
   T -> T U
   T -> c
3. Concatenation
   T -> T U
   You're trying to parse the concatenation of T and U.
   To some extent this means the cartesian product
   of the set of strings matched by T
   with the set of strings matched by U

The hint code does this (for regular expressions)

1. recursion - OCaml recursion

2. alternation - make_or_matcher
   we want to build a matcher for P|Q (P and Q are patterns)
   To do this, compute mP = matcher for P
   and mQ = matcher for Q
   Now, all we have to do is glue mP and mQ together
   to get a matcher for P|Q

```
fun mP mQ ->
(fun tokens accept ->
    match mP tokens accept with
      | Some x -> Some x
    | None -> mQ tokens accept)
```

3. concatenation - make_appended_matchers / append_matchers

```
          To do this, compute mP = matcher for P
                         and mQ = matcher for Q
            Now, all we have to do is glue mP and mQ together
          to get a matcher for P Q

              fun mP mQ ->
             (fun tokens accept ->
                 mP tokens (fun token_suffix -> mQ token_suffix
accept))

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls

let rec make_or_matcher make_a_matcher = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher head
      and tail_matcher = make_or_matcher make_a_matcher tail
      in fun frag accept ->
         let ormatch = head_matcher frag accept
         in match ormatch with
            | None -> tail_matcher frag accept
            | _ -> ormatch

This is not how most practical parsers work; it's too slow,
has too much backtracking.  To make it faster, see CS 132.



Implementation strategies
  last time how a parser fits into a compiler
           classic software-tools compiler (Unix/Posix/Linux
approach)
            preprocessor
```

parser
type+name checker
code generator
assembler
linker
loader
whew!

another popular approach:

IDE - integrated development environment (Xerox PARC/Smalltalk)

one big happy program (IDE) that's working in harmony with your app

You run the code without ever leaving the IDE.

another axis (way to think about how to run programs)

1.compilers vs 2.interpreters

1. translates source code into machine code, then executes that.

+ better runtime performance

1.5. Start by interpreting the program;

just compile the parts of the programs that need to run a lot.

"Just-In-Time Compiling"

"JIT"

Java sort of pioneered it (you'll see this in next assignment).

Now popular in web browsers, that use it to run JavaScript

2. keeps source code (or a closely related version in byte code)) in memory

an interpreter written in a low leve language

executes source code (or byte code)

+ better develop-time performance

+ easier to debug

Q. What is byte code?

A. You're trying to run Java fast, but still in an interpreted way.

i.e., You've picked Door#2 but want whatever speed you can get.

You don't want this in memory:

"if (a < b) return c;"

you need to reparse every time you execute!

So instead, you invent a nice "virtual machine"

with instructions that you define.

Java VM is stack oriented.
It has instructions like 'ADD' (pop top two items off stack;
push their sum)

We decide 12 represent ADD
13 represent SUB
1 represent PUSH
...

So instructions now look like this:

```
   PUSH a
  PUSH b
  <
  JT x:
 x:
  PUSH c
  RET
```

1 200 1 199 47 15 19 .... representing the above.

A C or machine-code program can easily interpret these bytes.

Java runtime now comes with a "byte-code compiler".
You give it bytecodes for a program.
It can translate them into machine code.

```
$ javac Foo.java  # produces Foo.class containing the bytecodes
$ java Foo        # reads Foo.class bytecodes and interprets them.
```

at runtime, the java runtime calls the byte-code compiler
as a subroutine in main memory
and executes the machine code that it generates.

Q. What do you mean by better develop-time performance?
A. You're developing your program faster.

Q. Is bytecode similar to "pseudo-instructions" that a compiler might output?
A. It's a very similar idea. The Java bytecodes were designed for Java. They also work for some other languages. And they've been extended to support JS, Python, etc.

Q. But then doesn't something have to translate byte code down to machine code? So why its byte code fast?
A. Translation is only done for often-executed byte codes, so it's

worth it.

One extra thing that's needed:
 * Profiling - at runtime, you keep track of how often
      statements (or byte codes, etc.) are executed.
    This helps you decide what to optimize later.

BUT this comes at a cost.
   Traditionally: you do all translation before program starts
      Therefore: program startup is fast

   JIT: Programs start slow and can be sluggish,
      and users don't like to wait.

Q. so most of the code is translated into byte-code and get executed by JVM. only a small part of it will be translated into machine code to get faster. I can develop fast because "code -> byte code" is faster than "code -> machine code", right?
A. Yes.

Also, we have *dynamic linking*
  When the program starts, only a small part of it can actually run.
  Big chunks of it are not linked into the main program until later.

```
    $ ldd /usr/bin/sort
      linux-vdso.so.1 (0x00007ffc75b0a000)
      libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007ff860a99000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff860871000)
      /lib64/ld-linux-x86-64.so.2 (0x00007ff860ad9000)
```

With dynamic linking, we also have "self-modifying code" --
   programs that change themselves while they're running.
   (JIT also does this, internally.)

How does this affect programming languages?
  - Applications are commonly multilingual
       'sort' might be written in C++
     pthread library might be written in C.

There's a problem here: how to glue together code written in different
languages?

Languages may have different calling conventions at the machine level.
Typical answer: use C as a “lingua franca”
Everybody else understands C’s API,
and can be somewhat-compatible with C.

With Java, there’s something called JNI
Java code can call C or C++ code
Java compiler knows the C/C++ calling conventions

Q. Is this about how dynamic linking affects programming languages?
Or how JIT affects programming languages?
A. Both - both techniques can be used, even simultaneously in the same app. Self-modifying code is a powerful and dangerous concept,
and JIT and dynamic linking try to use it without running into the dangers.

Q during break. Can't we also run local sockets and have them talk to each other for systems using multiple languages?
A. Yes, and this can simplify the linking issues, but it often has worse performance.

Java

background
let’s fix the problems we have with C++
application area (originally, 1990s,
Sun Microsystems: workstations and servers on the Internet,
$2,000 - $100,000 each
all running the SPARC processor
running Unix (C) + apps (C++/C))
Looking ahead to IoT (programs in your toasters)
Can we run Unix etc. on these small devices?
Tried it: BUT ran into problems
1. everybody wants to use their own CPU to save a few bucks
they didn’t want to buy SPARC cpus - too expensive!
2. C/C++ are reliability disasters
programs crash too often, not good for toasters
3. Internet speeds were slow (20 kb/s)
C/C++ executables are large
4. C++ is toooooo complicated!
Tried working around the problems (2, 4)
stripped down version of C++, “C+-”

Still had problems 1,3 and 2 to some extent.

So, they went to their competitors and stole their ideas.
Xerox PARC - Ethernet, mouse, bitmap display, desktop publishing,
Smalltalk
Smalltalk - IDE for a language called Smalltalk
+ OO (like C++)
+ runtime checking for subscript errors etc.
+ garbage collector (you needn't free storagea)
+ bytecode implementation (more compact than machine code)
+ ...
- language is weird syntax
- more dynamic (runtime type checking, so reliability issues)

Q. how is bytecode more compact than machine code? trying to beat an the computer architecture guys at their own game seems like a recipe for disaster
A. Architecture guys have two major disadvantages
- general purpose, for any language
so not tuned for any particular language
- backwards compatibility with 1970s software

Q. Wait is Smalltalk an IDE or language?
A. It's a language; there is the Smalltalk IDE (written in Smalltalk).

Sun glued together good ideas from Smalltalk
with good ideas from networked Unix machines

OAKxxxxxxx
Java
In labs, it worked! They could program their toasters!

As a demo, they wrote a browser.
competing with #1 browser at the time: Mosaic (ancestor of Netscape / Mozilla / Firefox / IE / etc.)
called HotJava
like Mosaic BUT extensible:
you could download Java bytecodes
run them in your
and do fancy animations, etc.
eventually this died out, because

1. Competition added plugins (machine code to run in browser)

---

2. Competition added JavaScript (source code to run in browser)

In the meantime, while this was being shaken out:
Java designers used to multiprocessor servers, networked.
Java is really good for this environment
Java morphed into a big-app server-side OO language.

Java basics - as opposed to C++

C, C++ 'long' has to be at least 32 bits but can be more.
for efficiency reasons.
'float' doesn't have to be IEEE floating point format,
if some other format is more efficient
This leads to portability problems.
Java - all integer and floating point types are standardized
same on all platforms

```
byte   8
short 16
int   32
long  64
float 32-bit IEEE
double 64-bit IEEE
```

Java: variables, and object slots, cannot be uninitialized
rule is: you have to initialize them
or they default to zero

Q. So Java eliminates undefined behavior?
A. It certainly tries to, and it mostly succeeds.
Programs fail reliably (at some performance cost).

```
a = f(x) + g(y);
```

In Java, expressions are always evaluated left to right.

Arrays in Java.
Each array is an object, allocated via 'new', lives in the heap (not on stack, nor in static storage). You don't have to worry about pointers into an array being misused after the function returns and the stack is popped.

```
       int guesses_a[6];
       int guesses_b[] = new int[6];

     Java does subscript checking at runtime, and throws
     an exception if there's a violation (likewise for null
references).

     Array sizes are determined when the array is created at runtime,
     and are fixed thereafter.

Java is object oriented BUT not every value is an object

  primitive types (types of non-object values)
     small fixed number of these
     int long byte char boolean ... a few mor3e

vs
  reference types (types of objects)
    These include arrays, strings, classes of your own design.


  Q. what was the motivation for doing this? Is performance cost that
  significant? wouldn't it have made the language much more uniform to
  have everything an object?
  A. Performance, and yes it can matter for some apps.
     Admittedly this is controversial as it complexifies the
languages.
     Java folks have worked around the issue to some extent,
     by supplying "boxed types" as part of the library
        the Float class has objects that exist only to hold a 'float'
value.

Java has *single inheritance* - every class has at most one parent
   (In Java, every class except 'Object' has exactly one parent;
    'Object' is the root of the class hierarchy)
  C++ has multiple inheritance: classes can inherit from two or more
parents.
  Java approach is simpler - you needn't deal with "arguing parents"
     it can be more efficient

   But suppose you want something like multiple inheritance?
   *Interfaces*
```

Lecture 8 UCLA CS 131 lecture 2021-01-31

midterm next time - in-person WGYoung CS 50
  Please remember to do the survey before coming to class Wednesday.
  plan to do assigned seating

interfaces, inheritance etc. in Java
multithreading model

Last time:
  ordinary classes - you inherit an implementation (asset) from parent class
  interfaces   - you implement an API (obligation) from interface

abstract classes - (ordinary class + interface bolted together)

```
    abstract class C extends D {   // D could have an abstract method m.
      int v;
      int m() { return v + 1; }
      abstract int n();
    }

    class B extends C {
      int n() { return v + 2; }
    }

    C x = new C();   // error - no implementation for n
    B y = new B();
```

  Q. I think children of abstract classes can be abstract too right,
  if they don't wish to implement the parent's abstract functions?

  Abstract classes are used for shared patterns of programming

```
     abstract class List {
       some API (for primitives)
       and some code (assumes the primitives, may not be efficient)
     }
     class ArrayList extends List {
       list is implemented via an array of items...
       must implement all the API
     }
     class LinkedList extends List {
       list is implemented via a linked list of items...
       must implement all the API
```

```
    }
```

Q. Can you elaborate on the List example? Why do we care about abstract classes, it looks like interfaces would suffice for that.
A. Interfaces would suffice, but now you have two things to worry about: C+I instead A (C class, I interface, A an abstract class).

final classes - ordinary classes that are not allowed to have subclasses

```
final class A extends B {
   private int w;
   int o() { return v + w; }
}
```

```
class Z extends A // error - you're not allowed to subclass final classes
```

You can also have final methods.

What is 'final' good for?
- performance - methods cannot be overridden, so they can be inlined because the compiler knows their implementation.

Q. What is "inline"?
A. A form of optimization where the compiler treats a function or method call as if it were the body of the function,

```
Source:
final int m() { return v; }
final int n() { return m() + 1; }

...
A x = ...;
return x.n();  // can be compiled as if it were x.v + 1;
```

This optimization cannot be done if some subclass of A has overridden the n() method.

Q. Can methods be final and abstract?
A. No, because you cannot override final methods, and you must override abstract methods in any implementation.

A little tour of the Object class - see the Java SDK documentation
This tells you what the Java designers thought was core to the language,
to some extent. Compiler support is needed for some of these
(the implementation cannot be written in Java directly

```
public class Object {

   // This gives you an object.
* public Object() { }
   // Object o = new Object();  -- a vanilla object with no special
properties.
   //   some possible uses - test case
   //                        sentinel   a[n] = new Object ();
                                       while (a[i] is x)
                                         continue;
```

Q. If you have an array of Person objects, how could you add an element of type (generic) Object?

A. You can't. Each element of a Person array must be a Person. If you have an array of Object, it can contain whatever object you like.

Q. Why ever have an array of Object?

A. Rare for an ordinary application.
More for a code that needs to deal with any kind of object (say a Java debugger).
Java has generic types (like ML) today
However, original (circa 1990s Java did not)
array of Object was the only way to write generic code

```
* public boolean equals(Object obj);
   //   o1.equals(o2)    -- default is reference comparison (pointer
comparison)
   //                      You can override this in your own classes.
   //                          e.g., string comparison.
```

Q. Any superclass can be casted to one of its subclasses, since object is the first class, you can cast it into anything.

A.

```
    Person p = ...;
  Object o = p;    // This is OK.
```

```
        Object o = ...;
      Person p = o;  // not allowed
      Person p = (Person) o;  // Compiler allows this BUT
                              // inserts a runtime check to make
                        // sure o's value is really a Person
                        // or some subclass of Person.
                   // Java designers prized safety.
```

Q. If youset an object to another object, does it deep copy or just copy the r reference?

A. Just copies the reference.

```
          p = q;   // copies reference q to p
       p = q.clone();  // clone q's top level into a new object;
                       // assign a reference to new object to p.

*  public boolean equals(Object obj);  // defaulting to pointer
comparison
*  public int hashCode();              // defaulting to be like "take
bottom
                                    // 32 bits out of the pointer"
           Like C/C++   (int) p  but you can't do (char *) i

                char *p = ....;
            int i = (int) p;   // Store bottom 32 bits of p's
representation
                                // into i.

                Object o = ....;
            int i = o.hashCode();

                int i = ...;
            char *p = (char *) i;   // OK in C

                int i = ...;
            Object o = (Object) i;   // not allowed in Java


          goal:    o1.equals(o2) should imply o1.hashCode() ==
o2.hashCode()
                  This is the only way that the Java library's
               hash table methods will work.
```

```
* public final Class getClass();

      Object o = ...;
      Class c = o.getClass();   // c is an object that represent's o's
class.

          'Class' is an identifier.
        'class' is a keyword.     They're quite different things.


       class Z extends A { .... }

       Class c = Z;   // Not allowed; Z is a class, not a Class.

         class - compile-time notion, used for static checking
                 (like 'class' in C++)
       Class - run-time object, used for computation at runtime.
                  Class is a class, just like Object is a class,
            and Thread is a class.

       getClass is useful for metaprogramming -
          code that reasons about your program.
        Say, a Java debugger that wants to display something
          about a random object,

          This is an example of *reflection* - when a program
          looks at itself and see something about its structure.

     Q. so if you do something like string.getClass(), the
     corresponding class object returned might have an object like
     getMethods() that will return a list saying "substr,getIndx,..."
     A. Yes, that sort of thing.

   Why is getClass 'final'?
     - So programs can't lie about themselves.
     - Java runtime needs to know the class of each object anyway
        to implement things like casts.
         (Person) o
       So all getClass does is give that info to you, the programmer.

* public String toString();
      // default is something boring (address converted to hex, say)
      // Typically overridden to be something more useful.
      //    class String { String toString() { return self; } ... }
```

```
      // Used for printing.

       System.out.println(o);  // operates by invoking o.toString()

    Q. Is the address you refer to the address in virtual memory?
    A. That's a common implementation, yes, for Java references.
       It's not required.  A Java reference could be just an index.

    Q during break. Can you have an abstract method in an ordinary
class?
    A. Not a good idea, because then you could instantiate the class.

* protected void finalize() throws Throwable;   // DEPRECATED - avoid
now
    In Java, you have a garbage collector GC that reclaims
      storage of objects that a no longer accessible from the program
         none of the variables in your program refer to these objects
        or refer to any objects that refer to these objects
        or refer to any objects that refer to objects
           that refer to these objects,
        ...
         Just before the GC reclaims the storage for an object o,
        it calls o.finalize()
        The default finalize() does nothing.

       What would you use this for?
       objects that represent system resources outside the control
         of the garbage collector
          e.g., a file descriptor in Linux (needs to be closed)
              a database connection to an outside database
               (needs to be dropped)

       Q. If it is now deprecated, what does GC do?
     A. It's supported, so GC still does it.

       Q. What happens after this feature goes away?
     A. You won't be able to rely on the GC to clean up
         your non-GC messes.
        This is kind of a good idea anyway;
          GC is not a good way to decide when to discard a DB
connection.

   'throws Throwable'
      In Java, if a method can throw an exception,
         you have to declare that in the methods API.
```

```
   'protected' ordinarily controls visibility of an identifier
      Here, it means only the GC should call 'finalize'.

* protected Object clone() throws CloneNotSupportedException;
   'protected' means not intended for general use;
      ordinarily you should create an object via new C()

There are more Object methods, and we'll talk about them
after talking about the Thread class.

Each object in the Thread class represents a thread of computation;
  a virtual CPU with its own instruction pointer (ip),
  computing at the same time as all the other threads.
  A Java program starts with a main thread.
  It can create other threads, and each thread can create other
threads.
  There has to be some limit here - depending on the implementation.

ordinary Thread's life cycle

 * creation via new Thread(...)          resulting state is: NEW

      Thread t = new Thread (r);
        // the object exists, but it
     // doesn't have an ip yet;
     // it's not running.
     // You're setting up the thread.

 * t.start();                           resulting thread state is:
RUNNABLE
        // allocates OS resources to
     // actually get the thread to run
     // with its own ip
        // RUNNABLE does not mean running,
     // it just means the OS can run it now, if it likes

       // Q. How is each instruction pointer managed?
       // A. see CS 111
       //      There can be more runnable threads than processors.
       //      OS decides which get to run (details don't matter
here).

 * While the thread is running, it can:
```

```
     - execute ordinary Java code                RUNNABLE
     - yield()                                   RUNNABLE

     - sleep                                     TIMEDWAITING
     - wait for some event                       WAITING for something
in Java
                                          we'll see methods for this
     - do I/O                                    BLOCKED for external
action


* exit the thread                                TERMINATED


Threads a bit of a disaster
  + multiple threads can let you compute your answers more quickly
  - multiple threads can step on each others' toes
      race condition when one thread writes
                     at about the same time that another reads or
writes.

           n++;   is a problem, because it's not atomic
                     Thread 1                  Thread 2
                   LOAD  n,r1                LOAD n,r1
                ADD   $1,r1               ADD  $1,r1
                ST    r1,n                ST   r1,n
                  This is faster on modern hardware.

Java has several solutions in this area.

  * synchronized methods.

      class C {
          private int n;
          synchronized int next () { return n++; }
      }


     Synchronized method works as follows:
     1.   get an exclusive lock on the current object
     // now we know, no other thread is executing
     // any synchronized method in this class
     2.   int tmp = n++;   // do the body of the synchronized method.
     3.   release the lock.
     4.   return
```

Q. What do you mean by the current object?
A. Every method is executed on behalf of an object,
and I mean that object.

Q. Does it prevent access to all synchronized methods, or just the one being called?
A. All of them; just one lock per object (not one per class).

Why aren't synchronized methods enough?

These are spin locks - sort like "while (o.locked) continue;"
so they tie up your CPU; that's inefficient if there's
contention for the lock.
Suppose the method's body takes a while to execute -
it's gonna be *really* inefficient.
We need a way to get a lock on an object while not chewing up CPU time.

```
public class Object {

* public void wait();

   o.wait();

    - Remove all locks held by this thread.
    - Wait until o "becomes available".   -- Thread state is WAITING
    - Reacquire the locks (spinning again)

* public void notify();

   o.notify();
    - wake up one of the threads waiting for o
        that thread changes from WAITING to RUNNABLE

        Q. Which thread?
      A. Up to OS.  FIFO is common but threads can have priorities.

* public void notifyAll();

   o.notifyAll();

    - wake up all of the threads waiting for o
        that thread changes from WAITING to RUNNABLE
```

for when you don’t trust the OS that much

Even wait and notify are pretty low level; it’s easy to get this wrong.

Java provides several higher level library classes to do synchronization.

class Exchanger

```
Exchanger x = new Exchanger();
```

```
Thread 1:                              Thread 2:

Object b = ....                        Object p = ....;
Object a = x.exchange(b);              Object q = x.exchange(p);
  at this point a = p                    at this point q = b
```

Q. =-

Lecture 9 UCLA CS 131 lecture 2022-02-7

Exchanger - exchange objects between two threads
whichever thread gets there first waits, second thread gets immediate answer (object from first thread)

neither thread has to use keyword synchronized
(complicated stuff is hidden away in Exchanger class)

Semaphore - represents a collection of n resources

acquire() -> wait for one of n resource to become available and acquire it

tryAcquire() -> doesn't wait, returns boolean if was able to

release() -> give up resource so that others may use it

CountDownLatch - synchronization method of threads, threads enter latch independently and are all released at the same time
(code can be executed while all the threads are waiting that can do whatever without worrying about thread tasks)
one time use
Horse race analogy

CyclicBarrier - repeating CountDownLatch, can use multiple times

Synchronization Hierarchy
synchronized -> wait/notify -> classes shown above

Below synchronized:
the motivation: if doing something that needs to be faster than synchronized and you need granular control of parallelism
"avoid synchronization when possible for performance"

The Java Memory Model
"As If" rule - it doesn't matter how Java is implemented as long as the programmer can't tell the difference; behavior is the same

i.e. assignment statements can be in executed in different orders if as long as end result is the same (to improve

performance)this rule applies to compiler and hardware interpreter (microcode)

this rule is followed per thread except
a) normal load, normal store
b) volatile load, enter monitor
c) volatile store, exit monitor

(monitor means monitor of critical section)
(yes means ok to reorder if ok in sequential code)
(no means no reordering allowed)

Second operation

First operation	A	B	C
A	Yes	Yes	No
B	No	No	No
C	Yes	No	No

multithreading causes problems when code is reordered!

Volatile variables - instance variable
value changes "spontaneously" (some other thread)
compiler cannot cache variable in a register so source code access implies machine code access

another method of inter-thread comms

vs

Atomic access - stores are atomic with respect to loads
(eliminate race conditions)

In Java, accesses and writes are atomic to single variables
(ref and most primitives except long and double)
Reads and writes are atomic for all volatile variables.

Types - set of values and set of operations on those values

```
description of implementation (data layout in a struct)

primitive - hard coded into language
                boolean, int, double, etc.
constructed - designed by programmer via type constructors
                array, struct, etc.

Uses of types
      annotations
      (for programmer understanding)
      (for compiler efficiency)

      inference
            a     +     b     =     c
      (double)        (long)     (double)
      (bool)          (char*)    (char*)

Static vs dynamic type checking
      static: compiler checks for type errors
            Java, C/C++, OCaml
            + reliability
            + performance

      dynamic: type errors at runtime during each operation
            Python, Javascript, Prolog, Scheme
            + fewer annoying diagnostics
            + flexibility

IEEE 32-bit floating point standard
            s     e          f
nbits       1     8          23
possible  0/1   0-255      0-(2^23 - 1)


E.g. +/-2^(e-127) *1f_2 0 < e < 255 normalized
+/-2^(e-126) *0f_2 e = 0 un-normalized (tiny)
+/- infinity e = 255, f = 0
+/- NaN (f) e = 255, f != 0
Inf - Inf or 0.0/0.0 -> NaN
Note: a-b==0 <=> a==b

Popping an empty stack is like 0/0, there is a special bit to
“flip” to a NaN-like value
```

Lecture 10 UCLA CS 131 lecture 2022-02-9

```
Logic Programming
      no functions
      no function calls
      no assignment statements  ] like functional
      no variables              ]

      + predicates - true or false statements about the world
      You make an assertion about the world and see if it is true.

Example predicate (sorting)

      sort(L, S)  <-- true when L is a list and S is sorted L

      sort(L, S) :- perm(L, S), sorted(S).

      see paper notes for more implementation details

How does Prolog get an answer?
      complicated pattern matching and unification process
      the sorting predicate is O(N!) complexity... yikes

Prolog Syntax
      Program = (Term)*
                 clause

      Term = number      |   atom             |     variable
             45, 5e-19    symbol/id           [A-Z_][[a-zA-Z0-9_]
                          [a-z][a-zA-Z0-9_]   _31
                          3 o' clock

            | structure
             f(3, a g(4))
             atom for a name, arrity (# args) > 0
             functor = atom/arrity i.e. f/3

      structure = atom ( args )

      args = term | term, args

Syntactic Sugar in Prolog
      [] = '[]'
      [a, b c] -> '.'(a, '.'(b, '.c'(c, '[]')))
      p :- q,r -> ':-'(p, ','(q,r))
```

Lecture 11 UCLA CS 131 lecture 2022-02-14

Prolog arithmetic - just syntactic sugar for a structure

```
3 + 4 * 5 -> '+'(3, '*'(4,5))
nothing is being evaluated here

to evaluate: use is/2
```

is(N,T) is true if N is a nm and T* is a struct that evaluates to N arithmetically

```
T* must be a ground term (not a log variable)
i.e. ?- 4 is N * N is not allowed!
```

Prolog is more about pattern matching and backtracking.

Define arithmetic in Prolog style.
notes: zero is 0, succ(T) is T + 1

```
add(zero, N, N).
add(succ(N), P, succ(NplusP)) :- add(N, P, NplusP)

sub(A, B, C) :- add(B, C, A)
```

Logical basis for arithmetic (Piano arithmetic) is not possible.

Prolog Execution Model

```
3 clause types
A. facts
     preq(CS31, CS131).
B. rules
     ipreq(A, B) :- preq(A, B).
     ipreq(A, Z) :- preq(A, B), ipreq(B, Z).
C. queries
     ?- preq(dunc101, cs131).

     backwards chaining computation - reasons backwards from
     queries not forward from facts and rules "goal oriented"

     depth first - a tree/list of goals is kept and left-most
     goal is always expanded first

Prolog unification - powerful pattern matching between single
                     goal and a clause head
```

```
      a unifier substitutes goal into clause head and clause head
      into goal -> if same, success.

      steps:
       1. find first clause that matches first subgoal w/ unifier
       2. replace goal w/ body of that clause
       3. repeat
       4. if no match, backtrack to next alternative

Be careful.
      loop :- loop.
      ?- loop -> infinite loop

      true.
      ?- true.
           yes.

      ?- fail.
           no. <- always fails (no src code, special predicate)

      repeat.
      repeat :- repeat.
      ?- repeat.
           yes.

      more commonly:
      ?- repeat, write('x'), nl, fail.
      outputs x on new line forever

Trouble with unification (see written)

not in Prolog means not provable, not that its false. \+ is a
better symbol.
```