**Compilers**

* 3 Reasons for compilers: Make it easy to write a program, less training required, get errors out of program
* X86, ARM, MIPS assembly architectures. X86 used in servers, desktop computers. ARM used in portable devices like phones, tablets. MIPS used in embedded systems like microwave oven. X86 used in this class
* Compiler takes a source program and creates a target program and does not exist after creation of target program. Interpreter stays around for execution of target program. Interpreter is the locus of control during all parts of execution. Compilers translate once, run many times. Interpreters translate every time
* printf in C uses interpreter at run time.
* REPL (Read, Eval, Print, Loop), LISP first language with REPL.
* Compilers generate smaller target program and are efficient in creation of target program. Interpreters are less efficient but are more flexible when creating a target program.
* Interpreters don’t have to wait for compile and link steps, takes less space than compiler, much easier to port
* Compilers have better performance, catches errors earlier before target program executes
* True compiler is full analysis, non-trivial transformation
* Easier to know meaning of program at high level, so easier to optimize at high level instead of assembly level
* Check if a string matches Regular Expression in O(n)
* Deciding if a DFA accepts a string is O(n)
* CFGs do not generate all possible strings of its terminals
* Its undecidable if two CFGs generate the same language, generate all strings that the other CFG generates, any other string the other CFG generates, if the language the CFG generates is regular, or if the CFG is ambiguous
* Is decidable if the CFG empty, finite, if CFG itself is regular (left or right regular), does a CFG generate a given string O(n3), is a nonterminal reachable, is a nonterminal productive (does a nonterminal eventually go to a terminal (not epsilon)), and is a CFG LL(1)
* Why is “does a CFG generate a given string” O(n3)
* LL(1) is left-to-right scan of the input, producing a leftmost derivation, using only 1 token of lookahead
* LL(1) never backtracks and can parse in time linear to size of string
* first L stands for left to right, second L stands for leftmost derivation
* Predictive parsing example is nondeterministic because we don’t know how many times recursion takes place to reach goal. Only have 1 token of lookahead.

expr -> expr + term (1+2+3+4+5)+6 expr -> expr + term

expr -> term (1+2+3+4+5) expr -> term

term -> factor

factor -> (expr)

factor -> number

* When said “it can be done mechanically” it means you can write a program to solve the problem
* Eliminating left recursion

X->Xa  
X->B

can be replaced by:

X->BX’  
X’->aX’  
X’->epsilon

^ Eliminating left recursion by transforming it into right recursion

* Find slide about something being reachable, something nullable
* Regular expression does not have recursion, CFG haIs recursion
* factor -> |- factor is right to left recursion
* factor -> | factor add\_op number is left to right recursion
* Parsing an LL(1) CFG can be parsed in O(n), CFGs are parsed in O(n3). O(n3) is the upper bound for a CFG. LL(1) is a special case of CFG that can be parsed in linear time
* With right recursion, you get right to left associativity
* Change left recursion to right recursion to get linear parsing and LL(1) predictiveness. Not recommended though, better way to do it. LR(k) is a better way
* LL(1) is top-down parsing
* LR(k) parsing is bottom up
* Use stack in LR(k) parsing and use stack in a more creative way
* LR(k) has two actions, a shift and a reduce
* X -> A B C; pop C, B, A off stack and push X onto the stack
* If the grammar is ambiguous, it is not LR(k)
* use . to indicate where you are in a parse
* E -> E . + T, period indicates parser just read second E, expecting to read + sign next
* If E then:
  + If E then:
    - S 🡨 Shift reduce problem
  + Else:
    - S
* You can solve shift reduce problem by always shifting or using an end if like Ada or Matlab
* C is tricky to parse because of type casting
* The language is not LR(k) if you have reduce/reduce or shift/reduce problems
* Always shifting does not always solve the shift/reduce problem
* k is fixed before the parse, it does not change throughout the parse
* Lexers take characters as input and creates tokens
* Use buffer start instead of null to end string in 32-bit linux because you need to be able to write null. If you used null to end string in 32-bit linux there would be no way to represent null if you wanted to represent null
* Look up what . represents in len = . – msg
* movl, l stands for long as 32 bits
* In x86 a word is 2 bytes, 16 bits.
* mov = 2 bytes, movl = 4 bytes, movq = 8 bytes
* .a file is an archive file, they contain static libraries
* .text is read only so it can not be overwritten, security reasons.
* Operating system requires memory that is given to be zeroed for security reasons
* int is interrupt in 32-bit linux
* .asciz puts null value at end of string. Need to use .asciz when calling a c output function ex. puts
* System V AMD64 ABI is the function calling convention
* Bool, char, short, int, long, long long, and pointers are in the INTEGER class
* machine code bool is 00000000 or 10000000
* xmm0-xmm7 registers are where floating point values are stored in x86
* registers %rbp, %rbx, %r12-%r15 must be preserved in System V AMD64 ABI. A called function must preserve these registers for its caller
* The two categories of registers we care about: INTEGER and SSE
* INTEGER registers: integral types that fit into general purpose registers
* SSE registers: Vector registers, hold floating point type variables
* \_start not included in assembly code for c because \_start is in glibc before the main function in a c program is executed
* r registers are 64 bits wide, e registers are 32 bits wide
* There is no way to assign a 64 bit literal to a register. For example you cant do movq $3.14, %xmm0. Must save 64 bit values to variable.
  + val:
    - .double 3.14
  + movq val, %xmm0
* If you don’t know how to go about writing assembly, look at how gcc does it and use as reference
* conv = scanf( “ %lf”, &fpVal );
  + Space in front of % to ignore whitespace, lf is long float, same as double
* input: 123456
  + scanf(“%3d%3d ”, &intA, &intB)
    - puts 123 in to variable A and puts 456 in to variable B and ignores endline character
    - If this scan was successful, it would return 2 because two values were assigned, returns number of conversions that happened
    - printf returns number of characters were printed
* exit immediately ends the program, using return ends all other processes that are going on outside of main. Exit does bare minimum to end program successfully
* The value that main returns is a signed byte, ranges from [-128,127]
* Stack pointer is 16 byte aligned
* If you don’t put return in main, whatever is in $eax is returned
* int \*ptr is 8 bytes in 64 bit linux
* If you know a lot of values aren’t going to be saved, you can save them on persistent registers (%r12 for example) to avoid using stack. Can’t put floating point values on these persistent registers
* flush after printf to make content of what to print immediately visible, nothing will print until the buffer is full or it is flushed