Noam Chomsky - developed theory of languages - context-free, context-senstitive grammars
“The Dragon book”

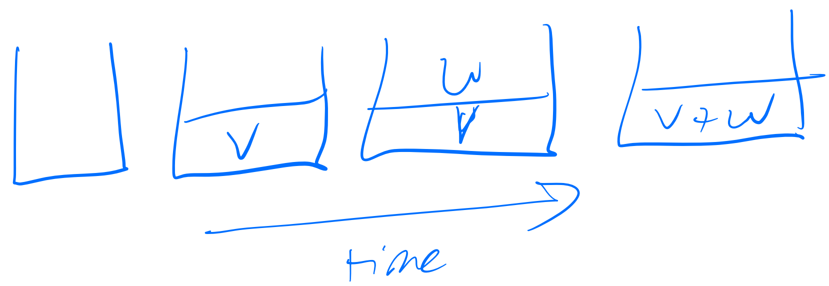
- 80s —> heavy on parsing and grammars (LL(1) LALR(1)…)

Line numbers
Punch cards (Hollerith)

Provide a higher-level language , raise the level of abstraction

COBOL - language for managers - Jean Sammet —> readability and understandability

Two kinds of assembly language - stack-oriented and register-oriented
PUSH v
PUSH w
ADD



Registers:
MOV v, A
MOV w, B
ADD A,B,C.   (C <- A + B)
MOV C, address

Instruction Set Architecture (ISA) x86 hybrid, ARM, RISC-V

FORTRAN

IBM - International Business Machines

COBOL “English-like” programming language

- Things we didn’t have: parsing, reserved words, scope, types, compilers, objects, complex data types, practically any memory, practically any processing power, practically any storage

“Ontogeny recapitulates phylogeny”

Every new computing technology - does a reboot of the evolutionary process of systems

- modern computing platforms- PCs, tablets, phones
    - Use languages with JavaScript, Swift, Java
    - Use OS - security, different users, multiple processes, multitasks
    - Hardware tons of processing power, tons of memory, tons of storage, multiple cores
- First PC: Altair 1975 - 4K of RAM, < 1 MHz, user interface lights and switches - assembly language (machine code), 1981 IBM PC - 640K RAM, 4.77MHz, screen, keyboard, BASIC (Gates & Allen)
- First mobile phones - first iPhone (single user, no virtual memory, no protection, single process)

COBOL - for business apps
ADD 1 TO 1 STORING IN A

FORTRAN - for scientists

“Automatic computers”

Scientists != programmers

FORTRAN - Formula Translation

PL - (COBOL) understandability and ease of use
(FORTRAN) programmability and efficiency - PORTABLE

John Backus

- Precedes scope, “structured programming”
- for-loop, “the goto statement considered harmful”

```
10   DO 100 I = 1, 10
..
.
...
100 CONTINUE
110 CONTINUE
```

110

I, J, K - integers
X, Y, Z - floats

$$\sum_{i=0}^{N} x_i \times y_i$$

10 DO I = 1, 10

context-sensitive

10 DO I = 1. 10

DOI = 1.10

3.4 ≥ 3

3.7

1000

Patriot missiles

counter

0 counter++

48 + 2

0

Compiler "optimization"

code in some lang. → semantically equivalent but faster code

source code analysis

STATIC analysis

If halts(P):
  While true:
    Pass
Else:
  Halt

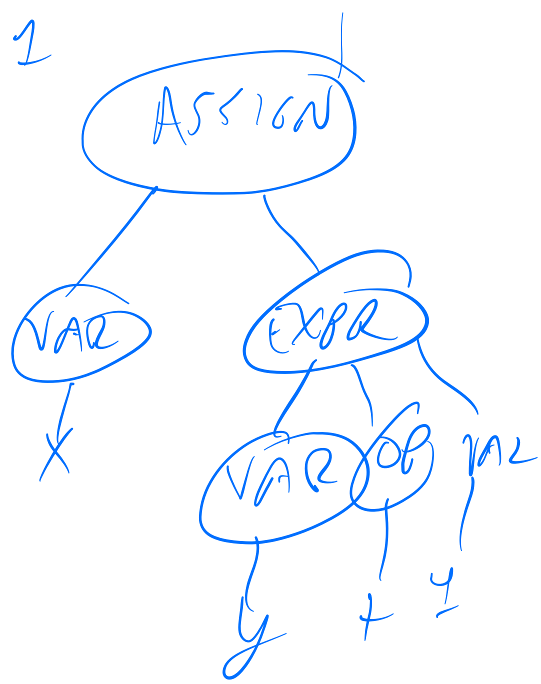
opti. compile → intermediate representation

Abstract Syntax Trees

Constant propagation
- x = 3
- Y = x + 12

—> x = 3, Y = 15

x = y + 1



Strength reduction:

- y = x ** 2
- –> y = x*x — y = 9
- X = x * 2 —> x <<= 1

Copy propagation
Inlining functions
- f(y(z))

F(x) = 2 * x
Y(x) = x * x

F(y(x)) = 2 * (x * x)

Global value numbering
Function prolog and epilog

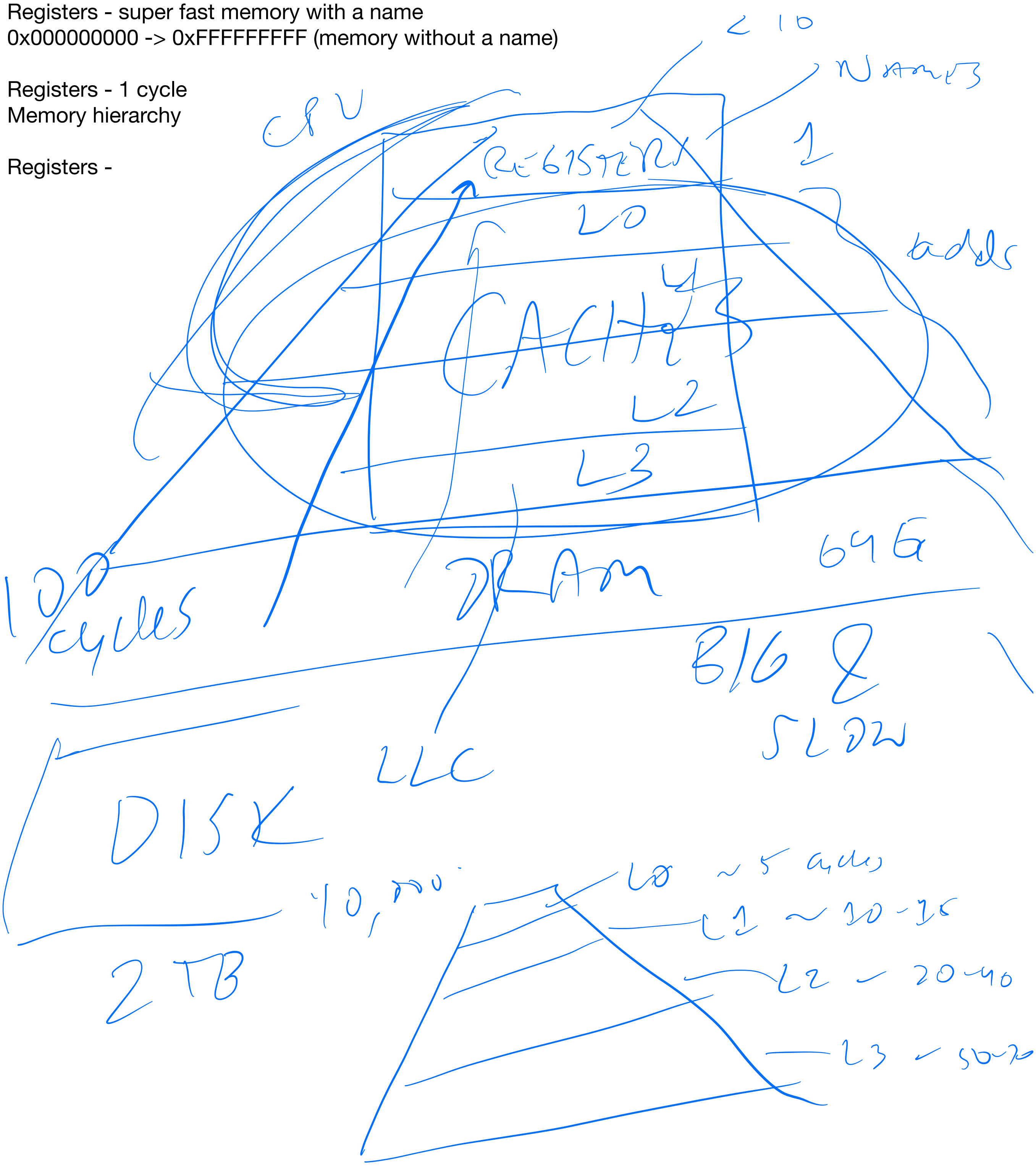
Parser
Analyzer & compiler (intermediate representation) — SSA static single-assignment
Compiler backend

Stack architectures vs. Register-based architectures

Registers - super fast memory with a name
0x000000000 -> 0xFFFFFFFFF (memory without a name)

Registers - 1 cycle
Memory hierarchy

Registers -



register allocation

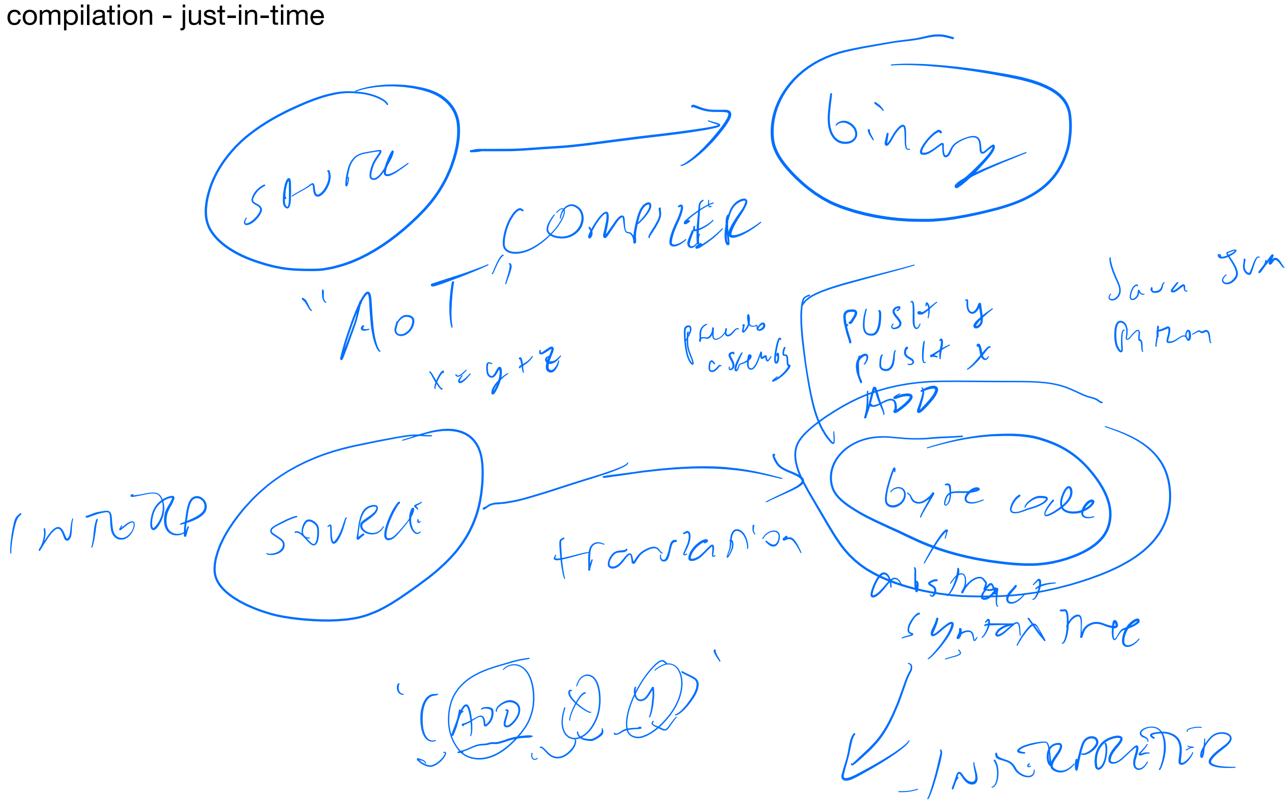
processors have limited number (under a dozen) registers

- super fast named memory (like A, B, …)
    - ADD EAX,EBX,ECX.   C <- A + B
- Void doSomething() {
-   Int a, int b, int c;
-   Float x, y, z;
-   Float arr[1024];
-   …
-   Int q = 12; /* dead */
- }
- Register access = 1 cycle
- L0 cache access ~= 5-7 cycles

arr is live

x y z

identify “live” variables
Maximize # of variables in registers over time - minimize “spills” to memory
“Graph coloring” NP COMPLETE
JIT compilation - just-in-time



JIT — compiler
interpreter

JavaScript

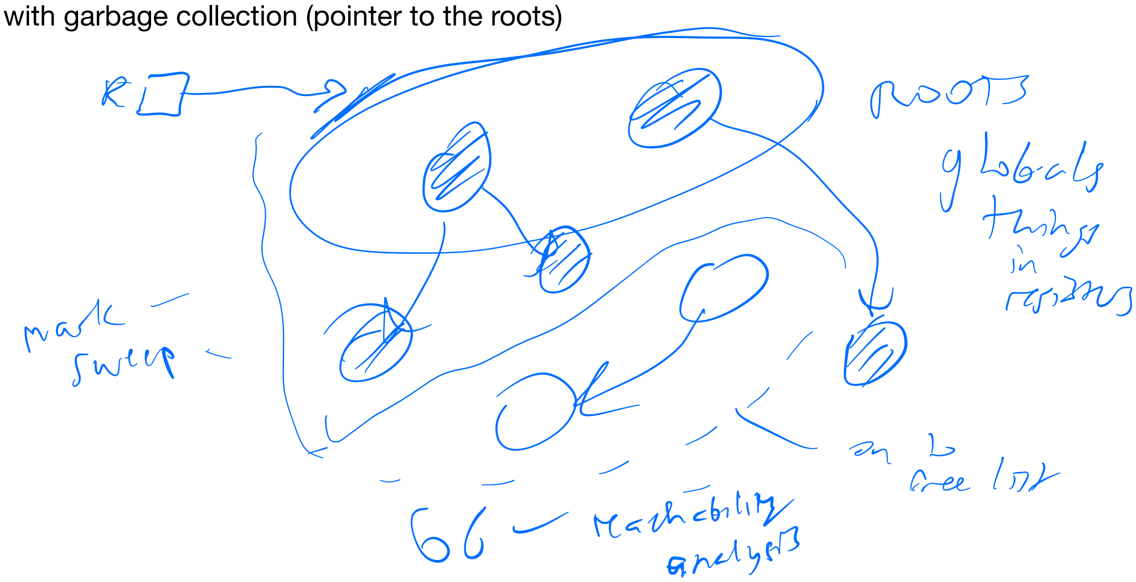
"Hot Spot" Oracle JVM

Background compilation, on demand, only for "hot" code

register allocation - polynomial time
accidentallyquadratic.com ?

Very expensive analyses - difficult for use in compilers (too slow for large codebases), almost impossible in JITs

- greedy heuristics, … (x86 very few registers - "register pressure")
- WebAssembly ("Not So Fast")
    - Dedicated registers - one of registers is for thread-local data, another one for something to do with garbage collection (pointer to the roots)



FORTRAN IS ALIVE?

APL

[x*x for x in range(100)]. "List comprehension"

Ken Iversion. Array Programming Language (IBM)

Numpy
BLAS. LAPACK
Basic Linear Algebra Subroutines
Linear Algebra …

FORTRAN

F2C -> 20-40% slowdown (1 processor)
Sequential FORTRAN code -> parallel execution (parallelizes loops over matrices)

Parallel - multiple threads, MPI, OpenMP

SQL. (IBM) Structured Query Language (follow-on QUEL) SEQUEL

“Turing completeness”

SQL program -> polynomial time —> NOT TURING COMPLETE (Excel, Datalog, regular expressions…)
All general purpose languages = Turing Complete
Some “domain-specific languages” (domain of databases, domain of spreadsheets, domain of regular expressions) are NOT

“Turing”

READ TAPE
WRITE 0
WRITE 1

IF R=0 GO LEFT
IF R=0 GO RIGHT

IF R=1 GO LEFT
IF R=1 GO RIGHT

…

—> MUL AX,BX, CX

“Peephole optimization”

(Very) low-level language —> very hard to compile
High-level language -> in theory, the intent is clear so you can compile it better

Map “*2” [1,2,3]

— in C, C++
POINTERS

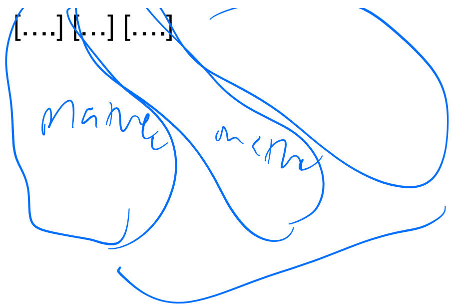
```
Int x[100];

For (int * p = x; *p != 0; ++p) {
}

Void sort(int * p, int N) {
}
```

FORTRAN
— NO POINTERS

```
Void matvec(float * m, float * v, float * r, int X, int Y) {
}
```

restrict restrict

Partitioning



combine

embarrassingly parallel

race condition

→ nondeterminism

✗ sequential version

float m[ ]

float n[ ]

m[3][j]

disambiguating pointers

pointer analysis

alias analysis

for ( )

for ( )

for ( )

m[x + i]..

foo()

bar()

intraprocedural

interprocedural

z = 2x;

x = 1;

y = 12;

xz = 14;

z

x $\boxed{14}$

y $\boxed{12}$

flow sensitivity

unitialization

path sensitivity

if ( )
  if ( )
    if ( )
      if ( )

exit

N branches

=> 2^N explosion info

int x;

int *p;

branch prediction

FORTRAN

- HLL for scientists
- portable
- readable/writeable
- fast

LISP

lambda calculus

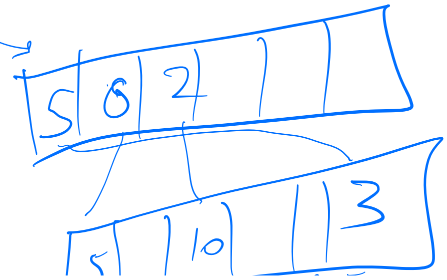
"symbolic" data structures

lists (heterogeneous)

nested lists

INDEX

offset cluster list



DATA ~~0 1 2 3~~ 4 5

3 → 5 → 10

'(3 5 10)

("hello"

5

("name"

"jones"

"height"

64

))

association lists

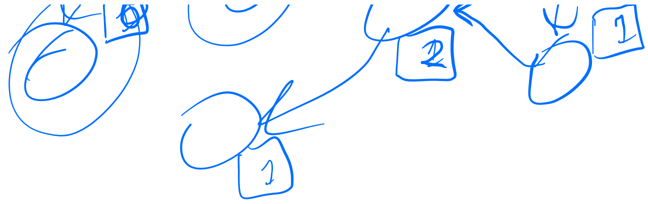
('dicts)

"JSON"

Lots of Insipid Silly Parens

SHRDLU

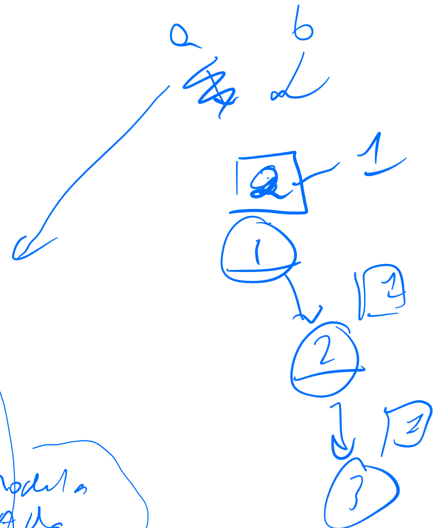
no cycles → ref counting Z



a = (1 2 3)

b = a

a = (4 5 6)



GC

Java

Perl RC

Ruby

Tcl

C#

Go

JavaScript

Python

OCaml

RC

Scala / JVM

Swift

Modula

Ada

Kotlin

LISP

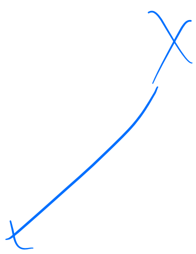
Scheme

Rust

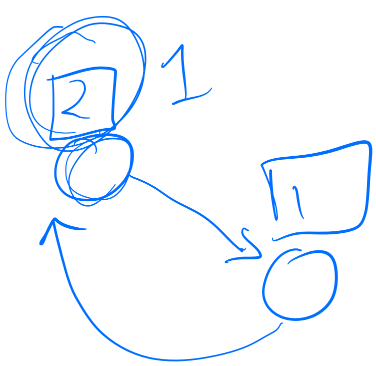
non-GC

C C++ FORTRAN

ALGOL → Pascal



NIL
(null)



RC + cycles = leaks

"incomplete"

ALGOL-like
LISP-like

Chomsky - language hierarchy - Language Acquisition Device

Recursively enumerable
Context-sensitive grammars
Context-free grammars.
Regular expressions

Cannot even count matching parentheses with regexes (((X))).
A B C 0 9
+
*
?

CFG
Tokens. “While” “for”. Lexical analysis: converts stream of chars into a stream of tokens

145. ->. Number 145
“While” -> WhileNode

Lexing -> Parsing

Backus-Naur form - BNF

Expr ::= ‘(‘ expr ‘)’ | number | Expr binaryOp Expr | unaryOp Expr
BinaryOp ::= ‘+’ | ‘*’

((3 + (-5)))

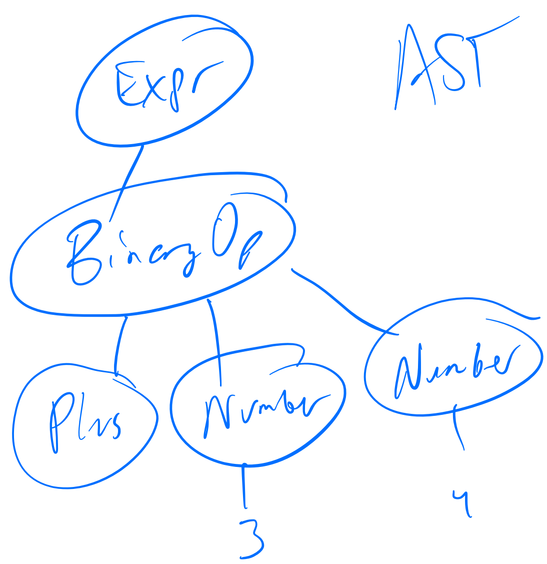
LL (1)
LR
LALR

(3+)

Abstract Syntax Tree - internal representation

AST

3 + 4



DO I = 1,

MOV 3, A

~~MOV 4, 0~~
ADD A, B, C

register A ← 3

“Packrat” parsing / Early parser

HTML, JavaScript, CSS
<BLINK>

<INPUT>
<SCRIPT DEFER>

WebAssembly
1. Emscripten - compile native code to run in the browser (JavaScript)
2. avoid parsing, faster to compile (faster than JS) PLDI 2016? WebAssembly - “Not So Fast”

Formally specified

Context-free grammar

Everything before ….. [this thing] -> -> ->

Backtracking

Garden path sentences

The old man the boat.
The horse raced past the barn fell.

Float x / int x
….
X = 1
Q = 1/ (X / 3)

Catching errors at Compile-time (static analysis) vs. At run-time (dynamic analysis)

- If you can catch errors at compile-time, you don’t need to check at runtime
- Program could trigger an error at run-time —> unpredictable / failure “in the field”

Semantic error
Printf(“hello worlb\n”);

Other errors:
- syntax errors
- Infinite loops?
- While (true) {
-   … (break)
- }
    - Embedded systems (we don’t know how long it’s going to run)
        - Reactive system / server - “run forever”
    - While (true) { }

- Byron Cook - Terminator
- Halting Problem - does program terminate for any input / for all inputs
- “Unsolvable” due to Turing

• Memory error - stack overflow, memory exhaustion, buffer overflow

unsafe langs: C C++ "unsafe" Rust

spatial error

use-after error

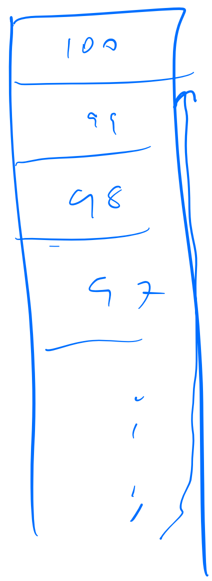
temporal error

Fibonacci(N) -> Nth Fib number

1 1 2 3 5 …

Fib(0) = 1
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)

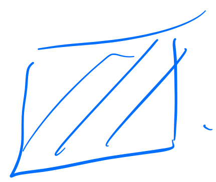
Fib(100)
Fib(99) + Fib(98)
Fib(98) + Fib(97) + Fib(97) + Fib(96)



Stack

Tail call elim.

4K 8K



PAGE

"memory protected"

"unmapped"

```
while (true) {
    malloc(1);
}
```

heap

OOM error

OOM killer

int x[100];
~~int y[1000];~~

x buffer

x [395] = 12

Out of Bounds exception Java

y [0 0 ... 12 ...]

17 bytes

32

24

Hoard

testing

Address Sanitizer

-fsanitize=address

BoehmGC

p = malloc(5)

p → 5

q

free(p)

q = malloc(5)

p → ...

use-after-free

dangling ptr

n-day

ALGOL like

Simula 68

C → C++

hierarchy
virtual functions
methods

functional language LISP → Common Lisp CLOS

Scheme

Clean

Python

Java . . .

Smalltalk

Self

David Ungar

dynamic optimizations
dynamic langs.

ML

OCaml

Haskell

Miranda

F#

JavaScript

V8

dynamic types

static types

int

let x = 3;

x = "hello";