

Lecture 2: Compilation, FORTRAN, and LISP

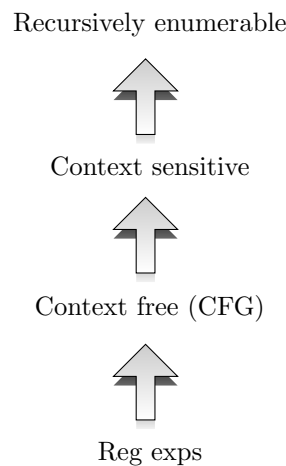
*Lecturer: Emery Berger**Scribe(s): Sarthak Nandi*

2.1 Last Class

FORTRAN and COBOL - FORTRAN was for scientists, COBOL was for business people

2.2 Grammar and Parsing

Chomsky hierarchy (1956):



We talk about grammar in "production style" from Algol-60 - Backus-Naur form (BNF)

$\text{expr} ::= '(\text{expr})' \mid \text{number} \mid \text{expr op expr} \mid \text{unary op expr}$

People have stopped working on parsing now.

backtracking exponential time

N tokens 2^N

Keep tokens in memory - needs to be efficient - Symbol table

Text \rightarrow Lexer (e.g. Lex, Flex) \rightarrow Tokens \rightarrow Parser (insert grammar) \rightarrow Parse LL(1), yacc/bison (LALR(1)), Java - Antlr

Packrat parsing - backtracking parser, general enough to handle things like C preprocessor `#define`

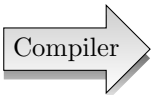
Newer PL don't have such things, macros are discouraged because you can do things like:

`#define while if`

2.3 Compiler Optimizations

One of goals of FORTRAN: fast code. Invented field of compiler optimization

Requirement: preserve semantics

Code(P)  Assembly (P')

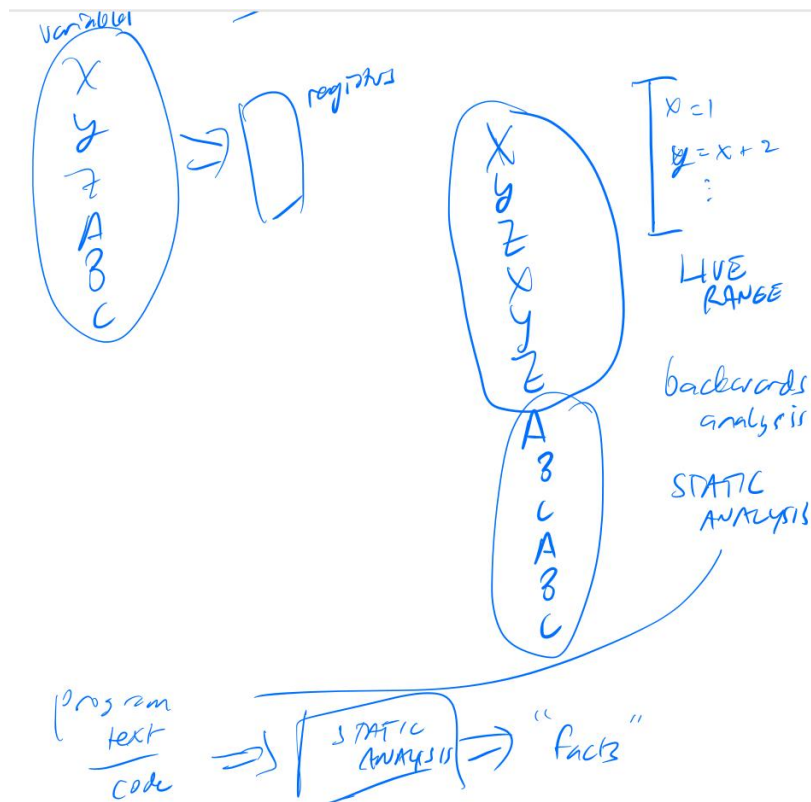
For all inputs: $P(i) = P'(i)$

Memory hierarchy:

Then: Registers, memory

Now: L0...L4, RAM

One problem: very few registers. So can keep some number of variables only in registers.



Good performance if number of variables in distinct live ranges is same as number of registers.

To find live range: backwards analysis, starting from bottom go up until you see the last use

First instance of static analysis:

Program text code  "facts"

Reality is subset of facts from static analysis.



Halting problem: Given program P: for all i inputs : P(i) terminates.

Proof by contradiction:

```

if halts(P)
    run forever
else
    halt

```

You can turn many complex analysis problems to halting problems, including static analysis for live range.

Basic static analysis on straight line code

Add complexity: Loops, Recursion, Pointers, Function calls, 1st class functions, EVAL (= EVIL) (Take string code and execute)

Intraprocedural: analyze within a function to optimize

Interprocedural: analyze whole program to optimize

(Why fortran is faster than C in next section)

Register allocation = graph coloring, NP complete problem.

Strength reduction: Replace an expensive operator by a cheaper one

$X = \text{pow}(3,2) = (3^2)$ can be replaced by

$X = 3*3 = 9$

Constant propagation: Replace constant usages with constant value $Y = 12$

$X = Y+1 = 13$

Inlining: Expand out some (small) functions in a program

```

int add(int x, int y) {
    return x+y
}

foo() {
    int c = add(2,3) = 2+3 = 5
}

```

Inlining effect is multiplied when function calls in loops are replaced. Inlining exposes optimization opportunities, reduces functional call overhead. Careful inlining combines with all optimizations

FORTRAN - optimizing compilation, LISP - interpreters

Starting in 90s we hybridised: JIT compiler

Ski-rental problem (CLRS): When you go skiing 1st time should you buy or rent equipment ? When should you buy ?

- Rental: \$10, Purchase: \$100
- Renting too many times and then purchasing is a bad idea, purchasing and using only once is also a bad idea.
- Hybrid: rent once, then buy
- Optimal strategy: Rent until you spend \$100 case: Spend \$200 when could have spent \$100.

Whole program analysis (with closed-world assumption): Big program would be slow

Modular analysis: C was designed for this, Go took this further. But can't determine how many times a function is called.

Real-time computation area - We need some deadline for real time systems.

JIT - Interpreter

Fast JIT - 00 (no optimization)

- 01

-02 (increasing optimization as we go down)

Modern JIT compilers will replace code within loops while it's in it if loop is long

(Paper by Jim Larus, repaying Moore)

LLVM is integrated in Safari for JS.

Compilers convert "dumb slow code" to "smart fast code", but can't replace algorithms.

2.3.1 Why FORTRAN is faster than C

Alias analysis, pointer analysis don't exist in FORTRAN (mostly)

```

X = [1,2,3] Y = [1,2,3]
INT *X;      INT *Y
FOR
    X[i]      Y[i]
    X+i        Y+i
    X=A        Y=B

if ()
    X = Y <- Don't know what happens in this case
else
    ..

```

2.4 Lisp

- Logical view of programming - Lambda calculus
- LISP is executable version of this logic, first functional PL (function-centric, functions are first class objects you can have list of function, can pass them)
- FORTRAN, COBOL are kinda hacks
- The code and data is same (homoiconicity)
- Compiler not needed - obviated
- Led to LISP being used for complex purposes
- Garbage collection