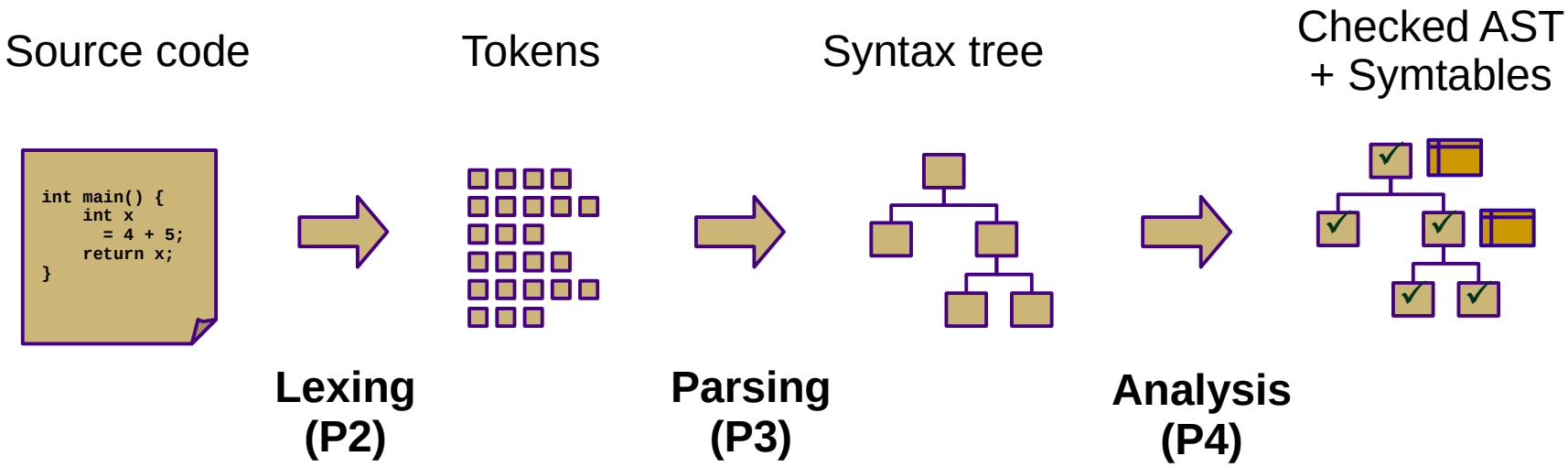
CS 432
Fall 2023

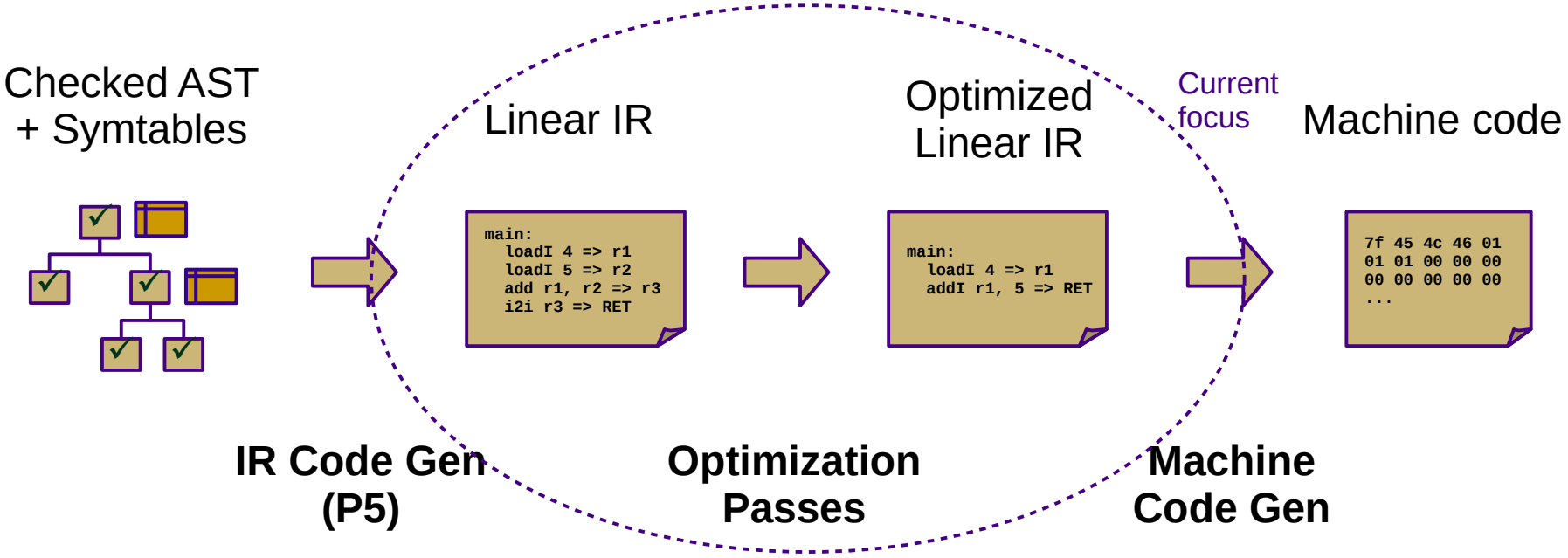
Mike Lam, Professor

Optimization

https://openclipart.org/detail/291407/performance

Compilers


Source code
Tokens
Syntax tree
Checked AST
+ Symtables
int main() {
int x
= 4 + 5;
return x;
}
Lexing
(P2)
Parsing
(P3)
Analysis
(P4)


Checked AST
+ Symtables
Linear IR
Optimized
Linear IR
Current
focus
Machine code
main:
loadI 4 => r1
loadI 5 => r2
add r1, r2 => r3
i2i r3 => RET
main:
loadI 4 => r1
addI r1, 5 => RET
7f 45 4c 46 01
01 01 00 00 00
00 00 00 00 00
...
IR Code Gen
(P5)
Optimization
Passes
Machine
Code Gen

Optimization is Hard

- **Problem**: it's hard to reason about all possible executions
  - Preconditions and inputs may differ
  - Optimizations should be correct and efficient in all cases

- Optimization tradeoff: investment vs. payoff
  - "Better than naïve" is fairly easy
  - "Optimal" is impossible
  - Real world: somewhere in between
    - Better speedups with more static analysis
    - Usually worth the added compile time
    - Requires working at multiple levels of granularity
    - Many active areas of research

Optimization (Ch. 8-11)

- **Local**
  - Local value numbering (8.4.1)
  - Tree-height balancing (8.4.2)
  - Peephole optimization (11.5)
- **Regional**
  - Superlocal value numbering (8.5.1)
  - Loop unrolling (8.5.2)
- **Global**
  - Constant propagation (9.3.6, 10.7.1)
  - Dead code elimination (10.2)
  - Global code placement (8.6.2)
  - Lazy code motion (10.3)
- **Whole-program**
  - Inline substitution (8.7.1)
  - Procedure placement (8.7.2)

Asides:

Data-flow analysis (Ch. 9)
Liveness analysis (8.5.1, 9.2.2)
Single static assignment (9.3)

Local Value Numbering

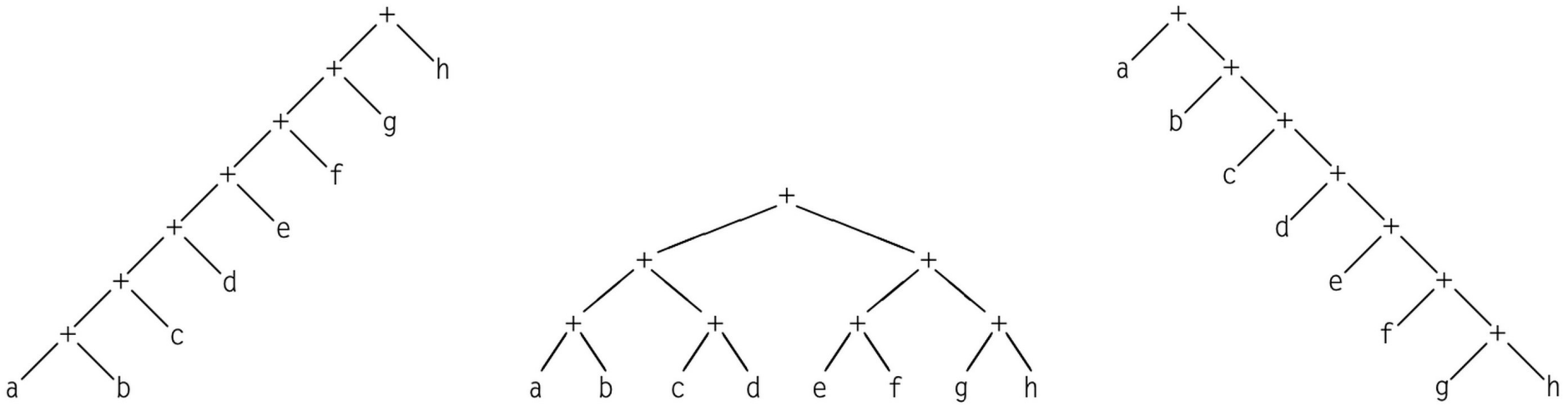
- Detect and remove redundant computation
  - Assign distinct numbers to each value computed
    - Typically using a hashing scheme
  - Requires SSA (or some similar scheme)
    - Avoids removing incorrect “redundancies”

```
a = x                    a = x
b = y                    b = y
c = a + b;               c = a + b;
d = a + b;               d = c;          // same number as c (a0 + b)
a = z;                   a = z;
e = a + b;               e = a + b;      // NOT the same number as c (a1 + b)
```

Original **Optimized**

Tree-Height Balancing

- Balance expression trees
  - Improves performance for CPUs with pipelining and/or multiple functional units (i.e., instruction-level parallelism)



■ **FIGURE 8.5** Potential Tree Shapes for a + b + c + d + e + f + g + h.

Peephole Optimization

- Scan linear IR with sliding window ("peephole")
  - Look for common inefficient patterns
  - Replace with known equivalent sequences

Example:

```
storeAI r5 => [bp+8]
loadAI [bp+8] => r7
```

```
storeAI r5 => [bp+8]
i2i r5 => r7
```

Generalized pattern:

```
storeAI a => b
loadAI  b => c
```

```
storeAI a => b
i2i     a => c
```

Loop Unrolling

- Replace loop body with multiple copies
  - Reduces number of comparisons and nonlocal jumps
  - Exposes additional opportunities for other optimizations

```
for (int i=0; i<N; i++) {
    do_something(i);
}
```

Original

```
int i;
for (i=0; i<N-4; i+=4) {
    do_something(i);
    do_something(i+1);
    do_something(i+2);
    do_something(i+3);
}
for (; i<N; i++) {
    do_something(i);
}
```

Unrolled by factor of 4

Profile: “Fran” Allen

- Frances Allen (b. 1932, d. 2020)
  - B.S. and M.S. in Mathematics
  - Worked at IBM, NYU, Stanford among others
  - First female IBM Fellow
  - First female ACM Turing Award recipient
  - Many foundational papers on optimizations and data flow analysis

Photo courtesy of Wikipedia

A
CATALOGUE
OF
OPTIMIZING
TRANSFORMATIONS

Frances E. Allen
John Cocke
IBM Thomas J. Watson Research Center
Yorktown Heights

Programming Techniques
G. Manacher, S. Graham Editors

A Program Data Flow Analysis Procedure

F.E. Allen and J. Cocke
IBM Thomas J. Watson Research Center

Profile: Frances “Fran” Allen

1. *Loop Unrolling*. A loop can be unrolled completely so that the successive computations implied by the loop appear sequentially or it can be partially unrolled as in the following example:

```
DO I = 1 TO 100 BY 1;
A(I) = A(I) + B(I);
END;
```

becomes when unrolled by 2:

```
DO I = 1 TO 100 BY 2;
A(I) = A(I) + B(I);
A(I+1) = A(I+1) + B(I+1);
END;
```

The advantages of loop unrolling are that

a. the number of instructions executed is reduced. In the preceeding example the number of increments and tests for loop control is halved.

b. more instructions are exposed for parallel execution. The two statements in the unrolled form of the preceding example can be executed at the same time since they are independent.

https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf

Dead Code Elimination

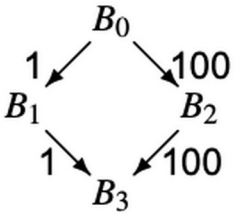
- Remove non-live definitions
  - Requires liveness analysis

```
int x = a + b;      // original
int y = a - b;
return x + 2;
```

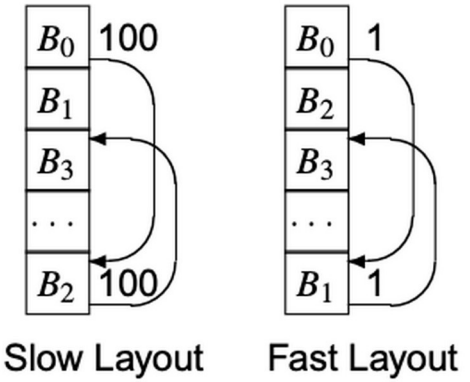
```
int x = a + b;      // reduced code size
return x + 2;
```

Global Code Placement

- Arrange basic blocks in order according to likely control flow at run time (hot path)
  - Takes advantage of code locality (lower cost of fallthrough vs. taken branches)
  - Requires profiling information
  - Build chains of hot paths



Hot path chains: $(B_0, B_2, B_3)_{100}$, $(B_1)_1$



Lazy Code Motion

- Move loop-invariant expressions out of loops
  - Combines three different dataflow analyses

```
// original

for (int i=0; i<nblocks; i++) {
    a = sqrt(cos(b)+sin(b));
    printf("%f\n", calc_something(a+(float)i));
}


// optimized

a = sqrt(cos(b)+sin(b));
for (int i=0; i<nblocks; i++) {
    printf("%f\n", calc_something(a+(float)i));
}
```

Tail Recursion Elimination

- Some recursive procedures can be converted to iterative procedures automatically
  - Avoids the overhead of multiple calls

```
// original

int fact(int n) {
    if (n < 2) return 1;
    return n * fact(n - 1);
}

// tail-recursive version

int fact(int n) {
    return fac_rec(1, n);
}
int fact_rec(int acc, int n) {
    if (n < 2) return acc;
    return fac_rec(n * acc, n - 1);
}
```

```
// removed tail recursion

int fact_rec(int acc, int n) {
loop:
    if (n < 2) return acc;
    acc *= n;
    n--;
    goto loop;
}

// inlined and condition-inverted

int fact(int n) {
    int acc = 1;
    while (n > 1) {
        acc *= n;
        n--;
    }
    return acc;
}
```

Inline Substitution

- Replace procedure call with callee's body
  - Avoids call overhead and exposes additional opportunities for other optimizations
  - Decision procedure to choose when to inline
    - Many potential criteria (callee/caller size, call count, etc.)

```
int foo(int x, int y) {
    return x + bar(y);
}
int main() {
    a = foo(1,2);
    b = foo(3,4);
    c = foo(5,6);
    return a + b + c;
}
```

Original

```
int main() {
    a = 1 + bar(2);
    b = 3 + bar(4);
    c = 5 + bar(6);
    return a + b + c;
}
```

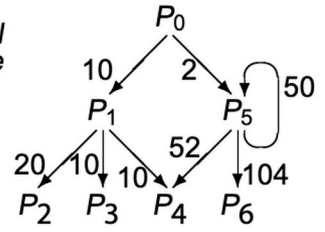
Inlined `foo` into `main`

Procedure Placement

- Extension of global code placement to procedures
  - Uses call graph instead of control flow graph

Initial State

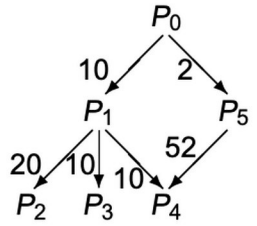
0.



	P_0	P_1	P_2	P_3	P_4	P_5	P_6
List(*x*)	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5\}$	$\{P_6\}$

Q: $\{(P_5, P_6)_{104}, (P_5, P_4)_{52}, (P_1, P_2)_{20}, (P_1, P_4)_{10}, (P_1, P_3)_{10}, (P_0, P_1)_{10}, (P_0, P_5)_2\}$

1.



	P_0	P_1	P_2	P_3	P_4	P_5	
List(*x*)	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5, P_6\}$	

Q: $\{(P_5, P_4)_{52}, (P_1, P_2)_{20}, (P_1, P_4)_{10}, (P_1, P_3)_{10}, (P_0, P_1)_{10}, (P_0, P_5)_2\}$

...

6.

P_0

	P_0	
List(*x*)	$\{P_0, P_1, P_2, P_5, P_6, P_4, P_3\}$	

Q: $\emptyset$

Link-Time Optimization

- Most compilers focus on a portion of the program at a time
  - Called a compilation unit (often a single file, class, or procedure)
  - Provides straightforward parallelization
  - Obstructs whole-program optimizations
  - Link-time optimization looks at the entire program

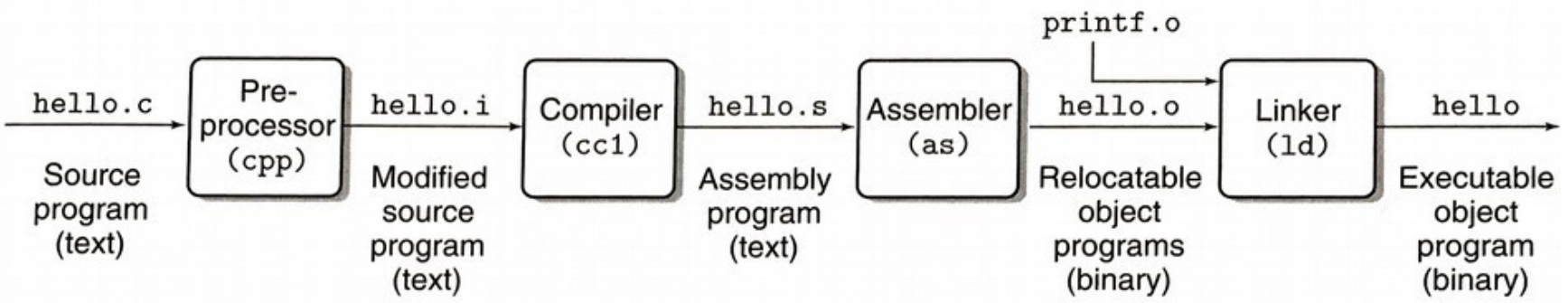


Figure 1.3 The compilation system.

Compiler Research

- Larger compilation units
  - What is the right balance?
- Smaller passes (e.g., nanopasses)
  - Allows for easy re-runs of passes
- Gradual typing: hybrid static & dynamic type checking
  - Some variables are annotated and checked statically
  - Others are checked at runtime
- Incremental compilation: cache IRs for future compilation
  - Improves re-compilation performance
- Integrated development environments (IDEs)
  - Greater collaboration w/ developer
  - Combination of systems and human-computer interaction (HCI)