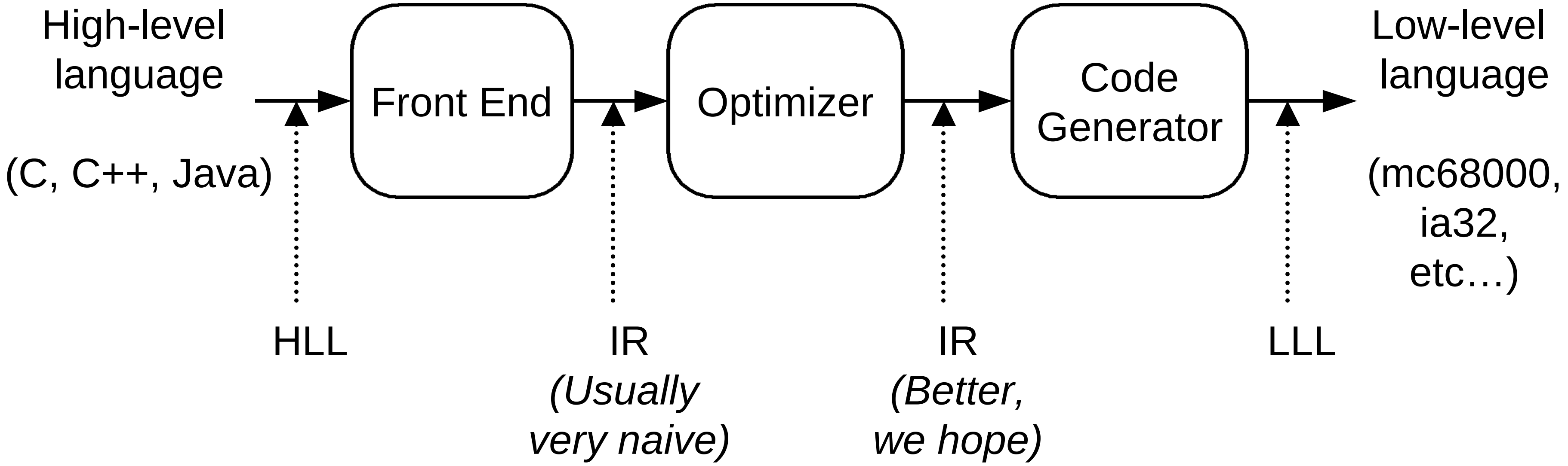
ECE1724F

Compiler Primer

http://www.eecg.toronto.edu/~voss/ece1724f

Sept. 18, 2002

What’s in an optimizing compiler?



High-level language

(C, C++, Java)

Front End

Optimizer

Code Generator

Low-level language

(mc68000, ia32, etc…)

HLL

IR

(Usually very naive)

IR

(Better, we hope)

LLL

What are compiler optimizations?

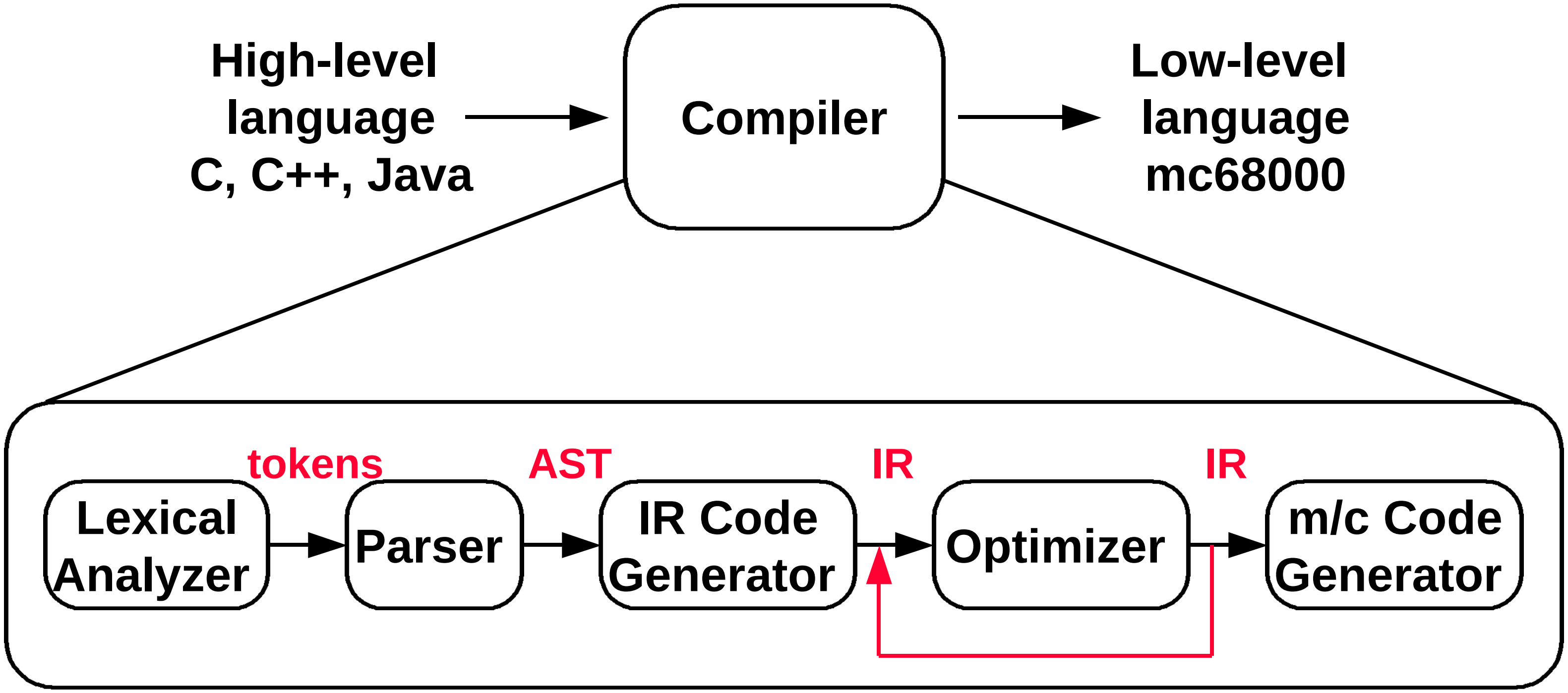
Optimization: the transformation of a program P into a program P’, that has the same input/output behavior, but is somehow “better”.

- “better” means:
    - faster
    - or smaller
    - or uses less power
    - or whatever you care about
- P’ is not optimal, may even be worse than P

An optimizations must:

- Preserve correctness
  - the speed of an incorrect program is irrelevant
- On average improve performance
  - P’ is not optimal, but it should usually be better
- Be “worth the effort”
  - 1 person-year of work, 2x increase in compilation time, a 0.1% improvement in speed?
  - Find the bottlenecks
  - 90/10 rule: 90% of the gain for 10% of the work

Compiler Phases (Passes)


High-level language C, C++, Java

Compiler

Low-level language mc68000

Lexical Analyzer

tokens

Parser

AST

IR Code Generator

IR

Optimizer

IR

m/c Code Generator

IR: Intermediate Representation

We’ll talk about:

- Lexing & Parsing
- Control Flow Analysis
- Data Flow Analysis

Lexing, Parsing and Intermediate Representations

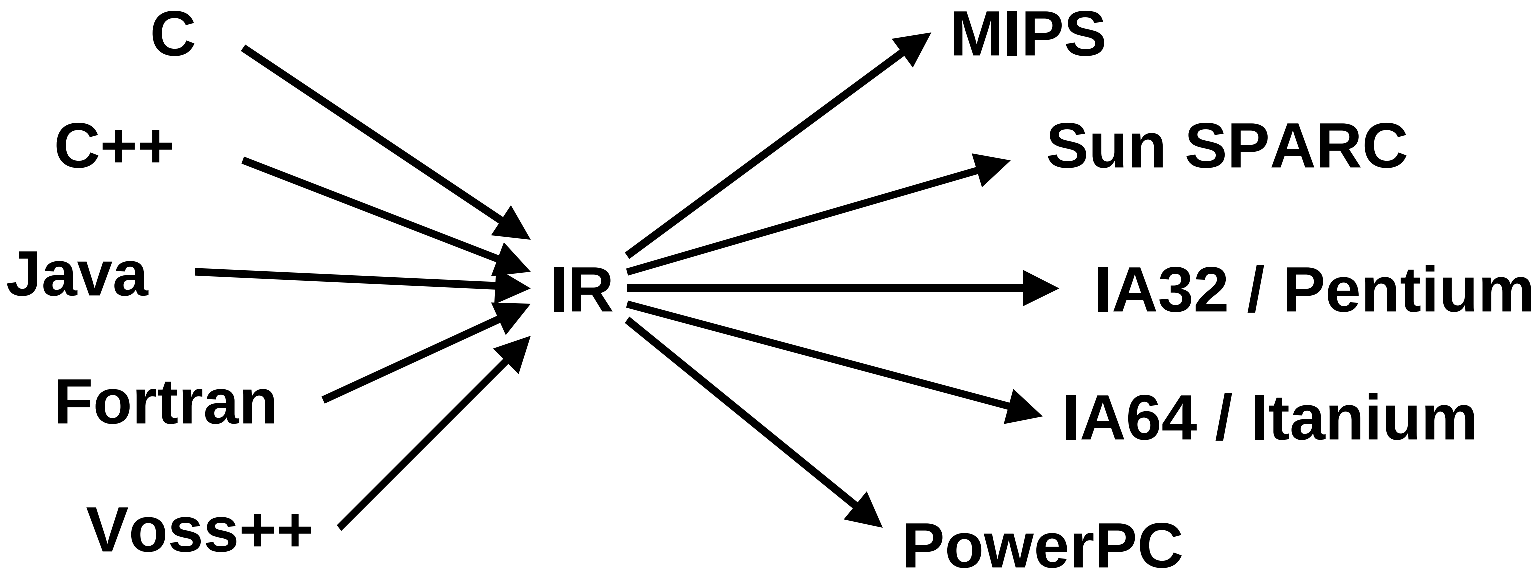
Lexers & Parsers

- ***The lexer*** identifies tokens in a program
- ***The parser*** identifies grammatical phrases, or constructs in the program
- There are freely available lexer and parser generators…
- The parser usually constructs some intermediate form of the program as output

Intermediate Representation

- The representation or language on which the compiler performs it’s optimizations
- As many IRs as compiler suites
  - 2x as many IRs as compiler suites (Muchnick)
- Some IRs are better for some optimizations
  - different information is maintained
  - easier to find certain types of information

Why Use an IR?


C
C++
Java
Fortran
Voss++
IR
MIPS
Sun SPARC
IA32 / Pentium
IA64 / Itanium
PowerPC

- Good Software Engineering
  - ☐ Portability
  - ☐ Reuse

Example:

```
float a[20][10];
… = a[i][j+2] …
```

```
t1    a[i,j + 2]
```

(a) High-Level

```
t1     j+2
t2    i*10
t3    t1 + t2
t4    4 * t3
t5     addr a
t6    t5 + t4
t7    *t6
```

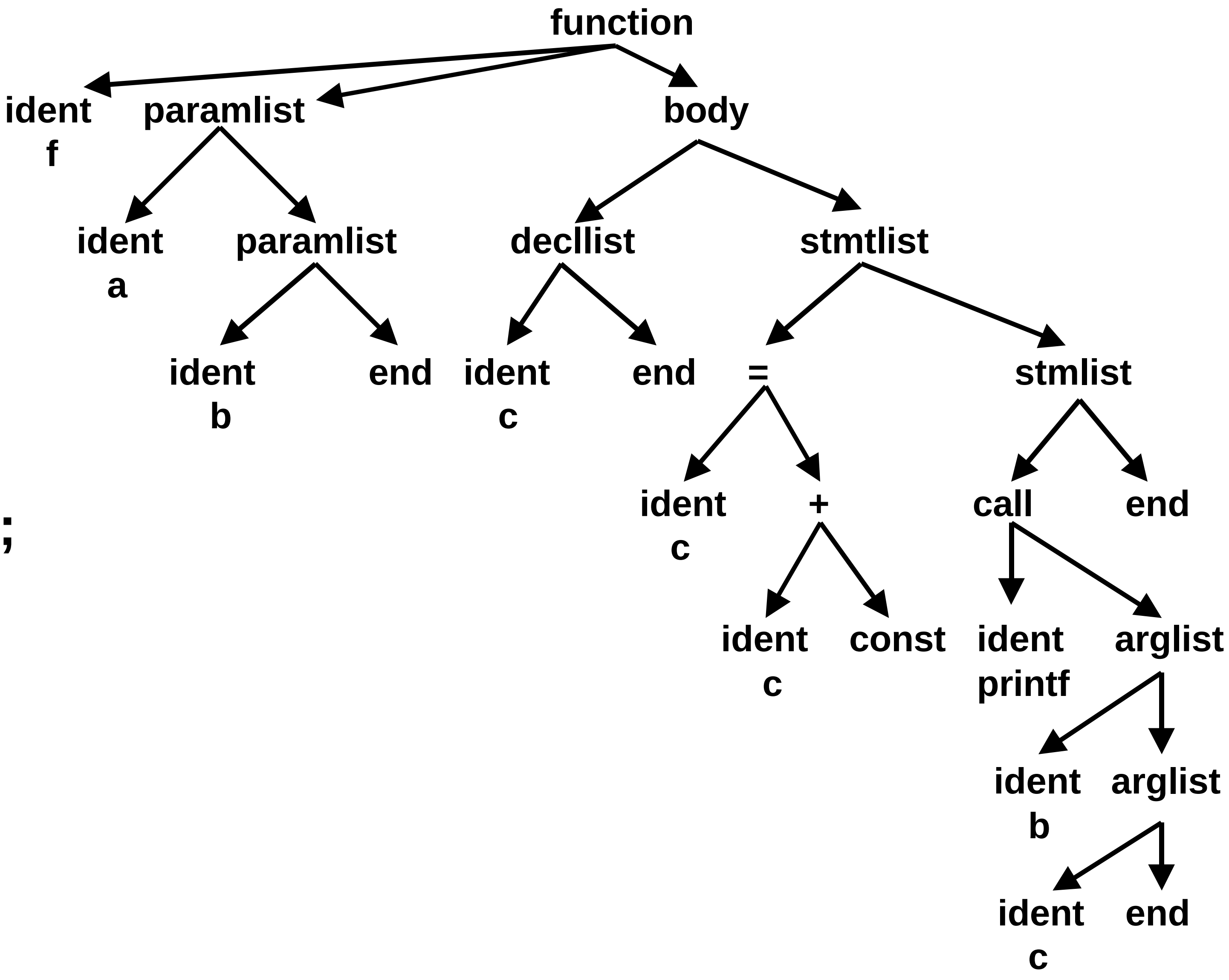
(b) Medium-Level

```
r1     [fp-4]
r2    r1 + 2
r3    [fp  -8]
r4    r3*10
r5    r4 + r2
r6    4 * r5
r7     fp - 216
f1    [r7 + r6]
```

(c) Low-Level

High-Level: Abstract Syntax Tree (AST)

```
int f(a,b)
int a,b;
{ int c;
  c = a +2;
  print(b,c);
}
```



function

ident f
paramlist
body

ident a
paramlist
decllist
stmtlist

ident b
end
ident c
end
=
stmlist

ident c
+
call
end

ident c
const
ident printf
arglist

ident b
arglist

ident c
end

Linear List (Very Similar to Source)

```
PROGRAM SIMPLE
    REAL  A(100,100)
    DO 100 I = 1,100
     DO 100 J = 1,100
100    A(J,I) = J*I
    WRITE (6,*) A
    END
```

```
S12     FLOWENTRY {succ = S1, line = 1,  }
 S1      ENTRY simple() {succ = S2, pred = S12, line = 1,  }
 S2      DO i = 1, 100, 1 {  follow = S7, succ = S3, S8, pred = S1, S7, out_refs
        * = i, line = 3, assertions = { AS_PARALLEL (i) } {AS_PRIVATE j,i }
        * { AS_LOOPLABEL SIMPLE_do100 } { AS_SHARED a }
 S3      DO j = 1, 100, 1 {  follow = S6, succ = S4, S7, pred = S6, S2, out_refs
        * = j, outer = S2, line = 4, assertions =
        * { AS_LOOPLABEL SIMPLE_do100/2 }  }
 S4   100 LABEL 100  {succ = S5, pred = S3, outer = S3, line = 5,  }
 S5      a(j, i) = i*j  {succ = S6, pred = S4, in_refs = i, j, j, i, out_refs =
        * a(j, i), outer = S3, line = 5,  }
 S6      ENDDO {  follow = S3, succ = S3, pred = S5, outer = S3, line = 5,  }
 S7      ENDDO {  follow = S2, succ = S2, pred = S3, outer = S2, line = 5  }
 S8      WRITE ([UNIT, 6], [FMT, *]) a  {succ = S9, pred = S2, in_refs = a,
        * line = 6,  }
 S9      STOP {succ = S10, pred = S8, line = 7,  }
S10       FLOWEXIT {pred = S9, line = 7,  }
```