CS 426
Compiler Construction

https://courses.grainger.illinois.edu/cs426/fa2023/
Instructor: Sasa Misailovic
Email: misailo@illinois.edu

TA: Yifan Zhao
Email: yifanz16@illinois.edu

We will use Piazza as our primary means of communication (including private messages)

- If sending an email to us, please include “[CS 426]” in the subject line
Schedule

Hybrid:

• Wednesdays 12:30pm-13:45 pm in Urbana: Everitt Building Room 1302
• Fridays 12:30pm-13:45 pm in Chicago: 200 S Wacker Drive, 7th floor Room 707

The other city can follow the Zoom live or recording (we will post all videos on Piazza)

Everyone can attend 50% of the lectures in person
Schedule

Course Format:
• Lectures – in-person/hybrid most of the weeks
• Projects – 4 (or 5) programming assignments (LLVM)
• Exams – midterm and final quizzes
• Office hours – both Sasa and Yifan, we will post the final schedule soon

Handling Questions during Lectures:
• If in the classroom, just ask
• If on zoom, type it in chat-box or ask ‘SPEAK’
• In both cases, I will repeat the question for everyone
Goals of the Course

Build a fundamental understanding of how to generate executable code from a programming language

Put your knowledge to practice by developing a working compiler from scratch in C++

Make the programs efficient by applying automatic program optimizations and code generation strategies

Learn about the long history and the bright future of compiler technology and recent trends
Brilliant but fuzzy idea
Programming language clarifies the ideas
Programming language clarifies the ideas

Compiler translates the idea to machine language

00111011101011001
How it started...

1930s
Using Heron's formula, we calculate the area of the triangle:

\[
S = \frac{IA + IB + IC}{2.0}
\]

\[
\text{Area} = \sqrt{S \times (S - \text{FLOATF}(IA)) \times (S - \text{FLOATF}(IB)) \times (S - \text{FLOATF}(IC))}
\]

Write output tape 6, 601, IA, IB, IC, AREA

601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2, 13H SQUARE UNITS)

Stop

End
How It’s Going

Program

Compiler

executable.exe

compile program.lang -o executable.exe

Executable Code
How It’s Going

Our Journey: What’s inside?
Compiler Overview
Role of a frontend

Program → Lexical Analysis → Parsing → Semantic Analysis

- Tokens
- Parse Tree
- AST + Sym Table
Role of a frontend

Program → Lexical Analysis → Parsing → Semantic Analysis

Tokens → Parse Tree → AST + Sym Table

\[ y = f(x) \]
Role of a frontend

Program

Lexical Analysis

Tokens

y = f(x)

NAME(y), EQ, NAME(f), LBR, NAME(x), RBR

Parsing

Parse Tree

Semantic Analysis

AST + Sym Table
Role of a frontend

- Program
  - Lexical Analysis
    - Tokens
    - Parse Tree
  - Parsing
  - Semantic Analysis
    - AST + Sym Table

Mathematical expression:

\[ y = f(x) \]
Role of a frontend

Program

Lexical Analysis

Tokens

Parsing

Parse Tree

Semantic Analysis

AST + Sym Table

y = f(x)

NAME(y), EQ, NAME(f), LBR, NAME(x), RBR

Assign

NAME(y) EQ Call

NAME(f) Arg

LBR NAME(x) RBR

Assign

Var(y) Call

Fun(f) Arg

Var(x)
Compiler Overview

Program → Front-end → Optimizer → Back-end
Role of a backend

Intermediate Representation

Assembly Language

sumcalc(int, int, int):
    mov    ecx, edx
    lea    eax, [4*rdi]
    cdq
    idiv   esi
    test   ecx, ecx
    js .LBB2_1
    lea    edx, [rax + 4]
    imul   edx, ecx
    add    eax, 5
    mov    esi, ecx
    lea    edi, [rcx - 1]
    imul   rdi, rsi
    add    ecx, -2
    imul   rcx, rdi
    shr    rdi
    imul   edi, eax
    add    edi, edx
    shr    rcx
    imul   eax, ecx, 1431655766
    add    eax, edi
    add    eax, 1
    ret

.LBB2_1:
    xor    eax, eax
    ret

Binary Code
Role of a backend

Intermediate Representation

```c
define i32 @sumcalc(int, int, int)
  (i32 noundef %0, i32 noundef %1, i32 noundef %2) {
    %6 = icmp slt i32 %2, 0
    br i1 %6, label %27, label %7

7:%8 = add i32 %5, 4,
%9 = mul i32 %8, %2,
%10 = add i32 %5, 5,
%11 = zext i32 %2 to i32,
%12 = add nsw i32 %2, -1,
%13 = zext i32 %12 to i32,
%14 = mul i32 %11, %13,
%15 = lshl i32 %14, 1,
%16 = trunc i32 %15 to i32,
%17 = mul i32 %10, %16,
%18 = add i32 %9, %17,
%19 = add nsw i32 %2, -2,
%20 = zext i32 %19 to i32,
%21 = mul i32 %14, %20,
%22 = lshl i32 %21, 1,
%23 = trunc i32 %22 to i32,
%24 = mul i32 %23, 1431655766,
%25 = add i32 %18, %24,
%26 = add i32 %25, 1,
    br label %27,

27:%28 = phi i32 [0, %3], [%26, %7]
    ret i32 %28,
}
```

Assembly Language

```assembly
sumcalc(int, int, int):
  mov   ecx, edx
  lea   eax, [4*rdi]
  cdq
  idiv esi
  test ecx, ecx
  js   .LBB2_1
  lea   edx, [rax + 4]
  imul edx, ecx
  add eax, 5
  mov   esi, ecx
  lea   edi, [rcx - 1]
  imul rdi, rsi
  add ecx, -2
  imul rcx, rdi
  shr rdi
  imul edi, eax
  add edi, edx
  shr rdx
  imul eax, ecx, 1431655766
  add eax, edi
  add eax, 1
  ret
.LBB2_1:
  xor eax, eax
  ret
```
Role of a backend

Intermediate Representation

```assembly
define i32 @sumcalc(int, int, int)
  (i32 noundef %0, i32 noundef %1, i32 noundef %2) {
    %6 = icmp slt i32 %2, 0
    br i1 %6, label %27, label %7
  7:%8 = add i32 %5, 4,
  %9 = mul i32 %8, %2,
  %10 = add i32 %5, 5,
  %11 = zext i32 %2 to i33,
  %12 = add nsw i32 %2, -1,
  %13 = zext i32 %12 to i33,
  %14 = mul i32 %11, %13,
  %15 = lshr i33 %14, 1,
  %16 = trunc i33 %15 to i32,
  %17 = mul i32 %10, %16,
  %18 = add i32 %9, %17,
  %19 = add nsw i32 %2, -2,
  %20 = zext i32 %19 to i33,
  %21 = mul i32 %14, %20,
  %22 = lshr i33 %21, 1,
  %23 = trunc i33 %22 to i32,
  %24 = mul i32 %23, 1431655766,
  %25 = add i32 %18, %24,
  %26 = add i32 %25, 1,
  br label %27,
  27:%28 = phi i32 [0, %3], [%26, %7]
  ret i32 %28,
}
```

Multiple HW Platforms

Reuses translation and optimizations!
Compiler Overview

Program → Front-end → Optimizer → Back-end
OF A COMPILER = Program Analysis + Program Transformation
Why is Optimization Important?

For source-level programming languages

Liberates programmer from machine-related issues and enable portable programming without unduly sacrificing performance.

John Backus on the first FORTRAN compiler:

“It is our belief that if FORTRAN, during its first months, were to translate any reasonable scientific program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger.”

“To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.”

Why is Optimization Important?
For better performance and portability

Current processors rely heavily on compilers for performance and domain specific processors and FPGAs require automated compilation of general purpose software.
Why is Optimization Important?

Because Moore’s Law is Dead

DARPA has an ambitious $1.5 billion plan to reinvent electronics

The US military agency is worried the country could lose its edge in semiconductor chips with the end of Moore's Law.

by Martin Giles    July 30, 2018

Last year, the Defense Advanced Research Projects Agency (DARPA), which funds a range of blue-sky research efforts relevant to the US military, launched a $1.5 billion, five-year program known as the Electronics Resurgence Initiative (ERI) to support work on advances in chip technology. The agency has just unveiled the first set of research teams selected to explore unproven but
Why is Optimization Important?

For new applications

Wearable computing (e-textiles)

Analog nano-computing (Bio)

Self Driving Cars

Edge intelligence
Compilers Optimize Programs for…

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging
Why is Optimization Important?
To Understand Benefits and Constraints

Three key properties for any optimization:

**Safety** — Does it change the results of the program?

**Profitability** — Is it expected to speed up execution?

**Opportunity** — Can we easily locate sites to modify?
Let's Optimize a Program...

```c
int sumcalc(int a, int b, int N)
{
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
```

Example from Amarasinghe, Rinard, Sarkar, MIT 6.035
Step 1: What’s interesting about y?

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
}
return x;
```
Step 1: What’s interesting about \( y \) ?

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
}
return x;
```
Constant Propagation

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*0;
}
return x;
```
Step 2: Do we need to compute this?

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*0;
}
return x;
```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
x = x + (4*a/b)*i + (i+1)*(i+1);
x = x + 0;
}
return x;
Algebraic Simplification

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x;
}
return x;
```
Step 3: Do we even need this statement?

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x;
}
return x;
```
Copy Propagation

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```
Step 4: What redundancy we have when computing with i?

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```
Common Subexpression Elimination

```c
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
Step 5: Did we forget about \( y \)?

```c
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```
Dead Code Elimination

```c
int i, x, y, t;
x = 0;

y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```
int i, x, y, t;
x = 0;
for (i = 0; i <= N; i++) {
    t = i + 1;
    x = x + (4*a/b)*i + t*t;
}
return x;
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;

Are we done now?
What more can we do?

• Avoid multiplication, instead use addition and shifting (Strength reduction):
  \[ x = x + \frac{4a}{b} \cdot i \]
  becomes:
  \[ v = v + \frac{a \ll 2}{b} \]
  \[ x = x + v \]

• Realize \( \frac{4a}{b} \) is never modified inside the loop
  Precompute it => Loop Invariant Code Motion
  \[ u = \frac{(a \ll 2)}{b} \]
  for (…) { …. 
    \[ v = v + u; \] …
  }
Register Allocation

Base pointer

<table>
<thead>
<tr>
<th>Local variable X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variable Y</td>
</tr>
<tr>
<td>Local variable I</td>
</tr>
</tbody>
</table>

mov $edx, [bp-12]  ;; load i into edx
add $edx, 1         ;; increment
mov [bp-12], $edx   ;; store result back to i
...

Register Allocation

$ r8d \equiv X$
$ r9d \equiv t$
$ r10d \equiv u$
$ ebx \equiv v$
$ edx \equiv i$

Then we know the specific local variable is always in this register, and this is all we need: add $ edx, 1 $
int sumcalc(int a, int b, int N)
{
    int i, x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for (i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}
**Unoptimized Code**

```
pushq  %rbp
movq  %rsp, %rbp
movl  %edi, -4(%rbp)
movl  %esi, -8(%rbp)
movl  %edx, -12(%rbp)
movl  $0, -20(%rbp)
movl  $0, -24(%rbp)
movl  $0, -16(%rbp)
.L2:
movl  -16(%rbp), %eax
cmpl  -12(%rbp), %eax
jg   .L3
movl  -4(%rbp), %eax
leal  0(%rax,4), %edx
leaq  -8(%rbp), %rax
movq  %rax, -40(%rbp)
movl  %edx, %eax
movq  -40(%rbp), %rcx
cld
idivl  (%rcx)
movl  %eax, -28(%rbp)
movl  -28(%rbp), %edx
imull  -16(%rbp), %edx
movl  -16(%rbp), %eax
incl  %eax
imull  %eax, %edx
addl  %eax, %edx
leaq  -20(%rbp), %rax
addl  %edx, (%rax)
movl  -8(%rbp), %eax
movl  %eax, %edx
imull  -24(%rbp), %edx
leaq  -20(%rbp), %rax
addl  %edx, (%rax)
leaq  -16(%rbp), %rax
incl  (%rax)
jmp  .L2
.L3:
movl  -20(%rbp), %eax
leave
ret
```

**Optimized Code**

```
xorl  %r8d, %r8d
xorl  %ecx, %ecx
movl  %edx, %r9d
cmpl  %edx, %r8d
jg    .L7
sal   $2, %edi
.L5:  movl  %edi, %eax
cld
idivl  %esi
leal  1(%rcx), %edx
movl  %eax, %r10d
imull  %ecx, %r10d
movl  %edx, %ecx
imull  %edx, %ecx
leal  (%r10,%rcx), %eax
movl  %edx, %ecx
addl  %eax, %r10d
cmpl  %r9d, %edx
jle   .L5
.L7:  movl  %r8d, %eax
ret
```

**Inner Loop:**

10*mov + 5*lea + 5*add/inc + 4*div/mul + 5*cmp/br/jmp = 29 instructions

Execution time = 43 sec

4*mov + 2*lea + 1*add/inc + 3*div/mul + 2*cmp/br/jmp = 12 instructions

Execution time = 17 sec
Code Optimization Example (For You!)

1 /* A, B, C are double arrays; X, Y are double scalars; rest are int scalars. 
2 int main(int argc, char** argv) {
3 ... /* Declare and initialize variables. */
4     X = ...;
5     N = 1; i = 1;
6     while (i <= 100) {
7         j = i * 4;
8         N = j * N;
9         Y = X * 2.0;
10        A[i] = X * 4.0;
11        B[j] = Y * N;
12        C[j] = N * Y * C[j];
13         i = i + 1;
14     }
15     printArray(B, 400); printArray(C, 400);
16 }
Code Optimization Example (For You!)

1 \( X = \ldots \)
2 \( N = 1; \)
3 \( j = 4; \) // Induction Variable Substitution (SUBST),
4     // Strength Reduction
5 \( Y = X \times 2.0; \) // Loop-Invariant Code Motion (LICM)
6 while (\( j \leq 400 \)) {
7     // Lin. Func. Test Replace (LFTR)
8     // Dead Code Elimination (DCE) for \( i \times 4 \)
9         \( N = j \times N; \)
10        // DCE of A, since A not aliased to B or C
11     \( \text{tmp} = Y \times N; \)
12     \( B[j] = \text{tmp}; \)
13     \( C[j] = \text{tmp} \times C[j]; \) // Common Subexpr. Elim. (CSE)
14     \( j = j + 4; \) // Induction Variable Substitution,
15         // Strength Reduction
16 }
17 printArray(B, 400); printArray(C, 400);
CS 598 SM

COURSE TOPICS
List of Topics (Part 1)

The order of topics is subject to change

Structure of a compiler and its history

Front-end of a compiler (review)
• Lexical analysis – identifying keywords
• Parsing – identifying structure of the program
• Abstract Syntax Trees (ASTs)
• Tradeoffs between different parsing strategies
List of Topics (Part II)

Back End of the Compiler

• Runtime Storage Management
• Intermediate Code Generation
• Machine Code Generation:
  • Overview: Instruction selection, register allocation, instruction scheduling
  • Global register allocation by graph coloring
List of Topics (Part III)

Optimizations
- Control flow graphs and analysis
- Static Single Assignment (SSA) form
- Peephole optimizations
- Introduction to iterative dataflow analysis
- SSA and iterative dataflow optimizations

Advanced topics
- Overview of advanced optimization techniques
- Compilers for Machine Learning
Topics We Will Not Cover

- Theory of program optimization, advanced program optimizations, pointer analysis (CS 526)
- Automatic vectorization, parallelization (CS 598dp)
- Compilers for Machine Learning (CS 598lce)
- New heterogeneous architectures (CS 598sa)
- Program verification (CS 476, CS 477….)
We touch many topics in Computer Science!

**Theory:** Finite State Automata, Grammars and Parsing, program logics, data-flow

**Algorithms:** Graph manipulation, dynamic programming

**Data structures:** Symbol tables, abstract syntax trees

**Systems:** Allocation and naming, multi-pass systems, compiler construction

**Computer Architecture:** Memory hierarchy, instruction selection, interlocks and latencies, parallelism

**Security:** Detection of and Protection against vulnerabilities

**Software Engineering:** Software development environments, debugging

**Verification:** Ensuring correctness of translation

**Artificial Intelligence:** Heuristic based search for best optimizations
CS 426 SM

COURSE PROJECTS
Programming Projects

An Optimizing Compiler for COOL using C++

Source Language: COOL
• Object-oriented language similar to Java
• But small and very well-defined: syntax and semantics

Target Language: LLVM Virtual Instruction Set
• Both intermediate representation and assembly language
• Designed for effective language-independent optimization
• State-of-the-art today
```plaintext
class Main inherits IO {
    pal(s : String) : Bool
        if s.length() = 0
            then true
        else if s.length() = 1
            then true
        else if s.substr(0, 1) = s.substr(s.length() - 1, 1)
            then pal(s.substr(1, s.length() - 2))
        else false
    fi fi fi
};

i : Int;

//... Continue left
```

```plaintext
//...continued
main() : SELF_TYPE {
    i <- ~1;  // assignment
    out_string("enter a string\n");
    if pal(in_string())
        then out_string("palindrome: YES\n")
    else out_string("palindrome NO\n")
    fi;
    self;
};  // end function main
};  // end class Main
```
Cool Grammar

```
program ::= class;+
class ::= class TYPE [inherits TYPE] { feature;* }
feature ::= ID(formal,*):TYPE { expr }
| ID:TYPE [ <- expr ]
formal ::= ID:TYPE
expr ::= ID <- expr
| expr[@TYPE].ID(expr,*)
| ID(expr,*)
| if expr then expr else expr fi
| while expr loop expr pool
| { expr;* }
| let [ID : TYPE [ <- expr ],]++ in expr
| case expr of [ID : TYPE => expr;]++esac
| new TYPE
| isvoid expr
```

https://courses.engr.illinois.edu/cs426/fa2022/Resources/cool-manual.pdf
Infrastructure

LLVM: Low Level Virtual Machine [http://llvm.org](http://llvm.org)

Chris Lattner

Vikram Adve
Infrastructure

**LLVM: Low Level Virtual Machine** [http://llvm.org](http://llvm.org)

- Virtual instruction set: RISC-like, SSA-form
- Powerful link-time (interprocedural) optimization system
- Many front-ends: C/C++, D, Fortran, Julia, Haskell, Objective-C, OpenMP, OpenCL, Python, Swift, ...
- Software: 1.3M+ lines of C++
- Open source: In use at many universities and major companies
LLVM Example

#include <stdio.h>

int main (int argc, char** argv){
    printf("Hello World!");
    return 0;
}

declare i32 @printf(i8* nocapture, ...) nounwind

@.str = private unnamed_addr constant [13 x i8] c"Hello World!\00", align 1

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
    %1 = tail call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds ([13 x i8]* @.str, i64 0, i64 0))
    ret i32 0
}
LLVM Example

```c
int abs(int val) {
    if (val >= 0) return val;
    else return -val;
}
```

```llvm
define i32 @abs(int)(i32 %val) nounwind uwtable readnone {
    %1 = icmp sgt i32 %val, -1
    %2 = sub nsw i32 0, %val
    %.0 = select i1 %1, i32 %val, i32 %2
    ret i32 %.0
}
```

```llvm
define dso_local noundef i32 @abs(int)(i32 noundef %0) local_unnamed_addr #0 {
    %2 = tail call i32 @llvm.abs.i32(i32 %0, i1 true), !dbg !25
    ret i32 %2, !dbg !26
}
```
LLVM Example

```c
int abs(int val) {
    if (val >= 0) return val;
    else return -val;
}
```

```
define i32 @abs(i32 %val) nounwind uwtable {
    %1 = alloca i32, align 4                ;; defines local variables
    %2 = alloca i32, align 4
    store i32 %val, i32* %2, align 4       ;; store val to local register %2
    %3 = load i32* %2, align 4             ;; load the value of local register %2 to register %3
    %4 = icmp sge i32 %3, 0                ;; compare to see is greater to 0, store to register %4
    br i1 %4, label %5, label %7          ;; go to the true or false branch based on value of %4

    5:%6 = load i32* %2, align 4           ;; true branch
    store i32 %6, i32* %1
    br label %10

    7:%8 = load i32* %2, align 4           ;; false branch
    %9 = sub nsw i32 0, %8
    store i32 %9, i32* %1
    br label %10

    10:%11 = load i32* %1                  ;; land here after executing either branch
    ret i32 %11, !dbg !21                  ;; return the final result
}
```
LLVM Doxygen is your friend

https://llvm.org/doxygen/

Basic Block
LLVM C Interface to LLVM » Core

A basic block represents a single entry single exit section of code. More…

Collaboration diagram for Basic Block:

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVMValueRef</td>
<td>LLVMBasicBlockAsValue (LLVMBasicBlockRef BB) Convert a basic block instance to a value type.</td>
</tr>
<tr>
<td>LLVMBool</td>
<td>LLVMValuesBasicBlock (LLVMValueRef Val) Determine whether an LLVMValueRef is itself a basic block.</td>
</tr>
<tr>
<td>LLVMBasicBlockRef</td>
<td>LLVMValueAsBasicBlock (LLVMValueRef Val) Convert an LLVMValueRef to an LLVMBasicBlockRef instance.</td>
</tr>
<tr>
<td>const char *</td>
<td>LLVMGetBasicBlockName (LLVMBasicBlockRef BB) Obtain the string name of a basic block.</td>
</tr>
<tr>
<td>LLVMValueRef</td>
<td>LLVMGetBasicBlockParent (LLVMBasicBlockRef BB) Obtain the function to which a basic block belongs.</td>
</tr>
<tr>
<td>LLVMValueRef</td>
<td>LLVMGetBasicBlockTerminator (LLVMBasicBlockRef BB) Obtain the terminator instruction for a basic block.</td>
</tr>
<tr>
<td>unsigned</td>
<td>LLVMCountBasicBlocks (LLVMValueRef Fn) Obtain the number of basic blocks in a function.</td>
</tr>
<tr>
<td>void</td>
<td>LLVMGetBasicBlocks (LLVMValueRef Fn, LLVMBasicBlockRef *BasicBlocks) Obtain all of the basic blocks in a function.</td>
</tr>
<tr>
<td>LLVMBasicBlockRef</td>
<td>LLVMGetFirstBasicBlock (LLVMValueRef Fn) Obtain the first basic block in a function.</td>
</tr>
<tr>
<td>LLVMBasicBlockRef</td>
<td>LLVMGetLastBasicBlock (LLVMValueRef Fn) Obtain the last basic block in a function.</td>
</tr>
<tr>
<td>LLVMBasicBlockRef</td>
<td>LLVMGetNextBasicBlock (LLVMBasicBlockRef BB) Advance a basic block iterator.</td>
</tr>
<tr>
<td>LLVMBasicBlockRef</td>
<td>LLVMGetPreviousBasicBlock (LLVMBasicBlockRef BB) Go backwards in a basic block iterator.</td>
</tr>
</tbody>
</table>
LLVM Doxygen is your friend

https://llvm.org/doxygen/classllvm_1_1Instruction.html
Exercise: Write Some LLVM IR Code

Use in-browser compiler at https://godbolt.org/

Can also do C++ to LLVM IR (use -emit-llvm option):
Programming Projects

Project Phases

• **MP1** – Scanning and Parsing: COOL to Abstract Syntax Tree (AST)

• **MP2** – Intermediate code gen., Part 1: AST to LLVM, local expressions only

• **MP3** – Intermediate code gen., Part 2: AST to LLVM, all of COOL

• **MP4** – Native code gen: Basic graph-coloring regalloc for LLVM to X86-64

• **Unit Project** – A mid-level dataflow optimization pipeline using SSA: e.g., ADCE, ConstProp, LICM, DomTreeCSE, plus optional passes (e.g., store hoisting, SROA, inlining).
Programming Projects

Project phases (tentative plan):

• **MP1** – September 1 to September 15

• **MP2** – September 15 to October 6

• **MP3** – October 6 to November 4

• **MP4** – November 4 to December 1

• **Unit Project** – October 20 to December 6
Project: Getting Started

1. Login and set up your account on the EWS machines. [we will send you directions early next week]

2. Read the COOL Reference manual (CoolAid), Chapters 1-11 (through syntax) at least. The manual will be available on Piazza and https://courses.engr.illinois.edu/cs426/fa2022/Resources/cool-manual.pdf

3. Download and read the COOL examples from the file cool examples.tar.gz. Link will be available on Piazza and https://courses.engr.illinois.edu/cs426/fa2022/Resources/cool-examples.tar.gz

4. Write a COOL program to get familiar with the syntax.

5. Optional: download and install LLVM on a Linux platform! Not necessary: We will set up a shared version on EWS.
4-credit students: DON’T read source code for scalar transforms.

All students: DON’T read source code for any LLVM register allocators.

We will allow a limited use of LLM-based Code Assistants.

• Will have to keep and submit your chat logs (they become your program’s documentation!)
• We may limit their use for specific MPs
• More Details to Follow!
CS 426 SM

COURSE LOGISTICS
Prerequisites

Necessary:
Good mastery of C++ (including templates)

Helpful:
Basic programming languages course (e.g., CS 421)
• Mainly for lexer/parser and codegen design
• Highly recommended for undergraduates
• But many topics are complementary between courses
Basic computer architecture (e.g., CS 233)
Basic models of computation (e.g., CS 374)

Most important: commitment to learn as you go
• You will realize you missed something!
Grading 3-credit

<table>
<thead>
<tr>
<th>Component</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP1-MP4</td>
<td>50%</td>
</tr>
<tr>
<td>Midterm Quiz</td>
<td>20%</td>
</tr>
<tr>
<td>Final Quiz</td>
<td>30%</td>
</tr>
<tr>
<td>Extra Credit</td>
<td>10%</td>
</tr>
</tbody>
</table>
Grading 3-credit

MP1-MP4  37.5%

MP5 (unit project)  20%

Midterm Quiz  15%

Final Quiz  22.5%

Extra Credit  10%
Quizzes

First
• Mid-October, in class
• Focuses on Parsing and Code Generation
• 75 minutes

Second
• Last day of classes, in class
• Focuses on Register allocation, instruction scheduling and code optimization
• Also includes the materials from the first one
• 75 minutes
More Details

Course Website (see the top bar):
https://courses.grainger.illinois.edu/cs426/fa2023/

Grading policies, Project grading details, ...

Various campus, college, and department services for student well-being
Books
No official book, but many times you will need to look into one of these:

Available online via Illinois University Library
And More Books

Helpful for advanced concepts (and CS526):

Available online via Publisher
And More ...

We will point out several classical papers that introduced the analysis and/or optimization techniques.

To access the papers from ACM/IEE prepend the link with the following:

```
http://www.library.illinois.edu.proxy/go.php?url=
```

For instance, for LLVM Paper:

```
```
Get Most of the Class

Get the big picture: Why are we doing this? Why is it important?

Understand the basic principles: If you know how to apply them, you can work out the details

Learn why things work a certain way: Automatic vs. manual, elegant vs. ad hoc, solved problem vs. open

Think about the cost-benefit trade-offs:
Performance vs. correctness, compile-time vs. payoff

Be active in class:
• Do the exercises in class; read the books and notes
• Start each project the day it’s handed out, not when it’s due
• Pay attention to the discussions, ask questions, and participate
QUESTIONS SO FAR?