What is a Compiler?

Compiler ≡ A program that translates code in one language (source code) to code in another language (target code).

Usually, target code is semantically equivalent to source code, but not always!

Examples

- C++ to Sparc assembly
- C++ to C (some C++ compilers work this way)
- Java to JVM bytecode
- High Performance Fortran (HPF: a parallel Fortran language) to Fortran: a parallelizing compiler
- C to C (or any language to itself):
  Why? Make code faster, or smaller, or instrument for performance . . .
Uses of Compiler Technology

- **Code generation**: To translate a program in a high-level language to machine code for a particular processor
- **Optimization**: Improve program performance for a given target machine
- **Text formatters**: translate TeX to dvi, dvi to postscript, etc.
- **Interpreters**: “on-the-fly” translation of code, e.g., Java, Perl, csh, Postscript
- **Automatic parallelization or vectorization**
- **Debugging aids**: e.g., purify for debugging memory access errors
- **Performance instrumentation**: e.g., -pg option of cc or gcc for profiling
- **Security**: JavaVM uses compiler analysis to prove safety of Java code
- **Many more cool uses!** Power management, code compression, fast simulation of architectures, transparent fault-tolerance, global distributed computing, . . .

**Key**: *Ability to extract properties of a program (analysis), and optionally transform it (synthesis)*
A Code Optimization Example

What machine-independent optimizations are applicable to the following C example? When are they safe?

```c
/* A, B, C are double arrays; X, Y are double scalars; rest are int scalars. */
int main(int argc, char** argv) {
    /* Declare and initialize variables. */
    X = ...
    N = 1; i = 1;
    while (i <= 100) {
        j = i * 4;
        N = j * N;
        Y = X * 2.0;
        A[i] = X * 4.0;
        B[j] = Y * N;
        C[j] = N * Y * C[j];
        i = i + 1;
    }
    printArray(B, 400);
    printArray(C, 400);
}
```
A Code Optimization Example: Result

```c
X = ...  
N = 1;   // Induction Variable Substitution (SUBST),  
j = 4;   // Strength Reduction  
Y = X * 2.0;  // Loop-Invariant Code Motion (LICM)  
while (j <= 400) {  // Linear Function Test Replacement (LFTR)  
    // Dead Code Elimination (DCE) for i * 4  
    N = j * N;  
    tmp = Y * N;  // DCE of A, since A not aliased to B or C  
    B[j] = tmp;  
    C[j] = tmp * C[j];  // Common Subexpression Elimination (CSE)  
    j = j + 4;  // Induction Variable Substitution,  
}  // Strength Reduction  
printArray(B, 400);  
printArray(C, 400);```

General Structure of a Compiler
Topical Outline

1. The structure of a compiler
2. Intermediate representations
3. Runtime storage management (excluding garbage collection)
4. Intermediate code generation
5. Code Optimization
   - Peephole optimizations
   - Control flow graphs and analysis
   - Static Single Assignment (SSA) form
   - Introduction to iterative dataflow analysis
   - SSA and iterative dataflow optimizations
6. Global Register allocation
7. Global Instruction Scheduling (if time permits)
Programming Projects

An Optimizing Compiler for DECAF using C++

Source Language: DECAF

- 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

Project phases

MP1: **Scanning and Parsing**: DECAF 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 DECAF

MP4: **Dataflow (SSA) Optimizations**: ADCE, LICM

Unit Project (Teams of 2): **Write a graph-coloring register allocator for LLVM on X86**
Getting Started on the Programming Projects

1. Login and set up your account on the EWS machines.
2. Print and read the DECAF manual, Chapters 1-11 (through syntax) at least. The manual is on the class web site under the Project/ link.
3. Download and read the DECAF examples from the Resource section of the class website. Write a DECAF program to get familiar with the syntax.
4. *DON’T* download or install LLVM! We will release a reduced version for your use in this class.
“Education is what survives when what has been learned has been forgotten.” —B. F. Skinner, New Scientist, May 21, 1964.

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
Getting The Most Out Of This Class

“Sir, I can give you an explanation but not an understanding!”

–British parliamentarian

- Do the exercises given in class (more on it later)
- Start the assignment the day it’s handed out, not the day it’s due
- “Come” to class.
Getting Started: Summary

- Read the CS 426 Web site — all pages
- Register for Piazza (or contact me ASAP if you have concerns)
- Log in and set up your EWS account
- Download and read the DECAF manual and examples
- Write a few simple DECAF programs
- Buy/Borrow the text books. Some exercises will be from the Aho... book.