Readings for This Lecture

- “Basics of Compiler Design”, Torben Aegidius Mogensen
- “Source Code Optimization”, Felix von Leitner
What is a Compiler?

• A compiler is a computer program that translates a programming language into a machine language.

• Programming languages are meant to be human-readable.

• Machine languages are the native instructions understood by the CPU.

• To write effective code, it helps to understand how compilers work.
Compiler Optimization

• Most machine languages are actually fairly simple.
• However, this makes them tedious to work with directly.
• The purpose of a programming language is to hide details so programs can be expressed more naturally.
• The instructions are at a higher level of abstraction.
• Unfortunately, this makes it easy to write inefficient code.
• Besides performing the translation to machine code, a compiler also tries to optimize the resulting code.
• The programmer can sometimes help this along.
Steps in Compilation

- **Lexical analysis**: Divides the text into *tokens*
- **Syntax analysis**: Generates the *syntax tree*
- **Type checking**: Checks for type consistency
- **Intermediate code generation**: Translates code to machine independent intermediate language
- **Register allocation**: Assigns variables to registers
- **Machine code generation**: Generates machine code (textual)
- **Assembly and linking**: Creates final binaries
Lexical and Syntax Analysis

Parse tree (pruned)
Python add5() function

Source: en.wikipedia.org/wiki/File:Python_add5_parse.png
**Type Checking**

- One of the major differences between programming languages is whether and how they do type checking.

- Type checking ensures that the program doesn’t perform operations that are undefined for a given type.

- Machine language is an untyped language, as it regards all data as just strings of bits.

- Most other languages have some kind of typing
  - Strong/weak
  - Static/dynamic
  - Explicit/implicit
  - Safe/unsafe
Interpreted Languages

• In an interpreted language, the program is executed directly using the syntax tree.

• A *virtual machine* with native instructions from the intermediate language executes the program as if it were machine code.

• The translation to the machine code of the computer itself essentially occurs in real time.

• This results in a number of inefficiencies, so interpreted languages tend to be slower than compiled ones.

• Python and Matlab are examples of interpreted language.

• You can write extensions to both Python and Matlab directly in C/C++, so a good strategy is to use C/C++ for functions for which speed matters.
Intermediate Code Generation

- Intermediate code is similar to machine code except that it is machine independent.

- Such code must make certain assumptions to be platform-independent, e.g., assuming an infinite number of registers.

- To make translation easier, most instructions are atomic.

- Translation of intermediate code to machine code is then independent of programming language.

- This makes it easier to port existing compilers to new architectures and to create compilers for new languages.
Machine Code Generation

- The final step is translating the intermediate code to machine code.
- This encompasses a number of different issues.
- Some instruction sets include nonatomic instructions of which we should take advantage.
- There are also a large number of variants on conditional jumps.
- Perhaps the most interesting step, however, is register allocation.
Register Allocation

• The *register allocation problem* is to map a large number of variables to a small number of registers.

• This is done by assigning multiple variables to the same register using *liveness* analysis.

• By analyzing when different variables are alive and dead, i.e., are never used again, we create a conflict graph.

• The nodes are variables and the edges indicate a conflict.

• The problem of finding the minimum number of registers is equivalent to *graph coloring*.

• If there aren’t enough registers, then we have *spilling*. 
Optimizations

• Common subexpression elimination: Evaluate common subexpressions once.

\[ a[i] := a[i]+2 \]

• Code hoisting: Remove code from loops that doesn't need to be executed every time.

```c
while (j < k){
    sum = sum + a[i][j];
    j++;
}
```

A large part of the calculation of \( a[i][j] \) is repeated each time

• Constant propagation: Replace constants with the actual value.
Loop Unrolling

- Although loops make code easier to read, they can be very inefficient.
- This is due to the overhead in incrementing the loop counter, loss of efficiency from combined writes, etc.

Normal loop

```c
int x;
for (x = 0; x < 100; x++){
    delete(x);
}
```

After loop unrolling

```c
int x;
for (x = 0; x < 100; x+=5){
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
```

Source: en.wikipedia.org/wiki/Loop_unwinding
Tail Recursion

• Recursion is a very effective tool for expressing algorithms succinctly.
• However it can generate inefficient code due to the overhead of function calls.
• If the last call in a function is recursive, it is easy to replace the recursion with iteration.

```c
long fact(long x){
    if (x <= 0) return 1;
    return x*fact(x-1);
}
```
Vectorization

• Because it's more efficient to write entire words to memory at once, compilers may “vectorize.”

```c
int zero(char *array){
    unsigned long i;
    for (i = 0; i < 1024; ++i){
        array[i] = 23;
    }
}
```

• On a 32-bit machine, it’s more efficient to write \texttt{0x23232323} 256 times.
Dirty Tricks

- Casting to unsigned makes negative values wrap to be very large.

```c
int regular(int i){
    if (i > 5 && i < 100){
        return 1;
    }
    return 0;
}

int clever(int i) {
    return (((unsigned)i) - 6 > 93);
}
```

- Comparison against zero is also more efficient than comparison against nonzero (count down instead of up!).

Bit Operations

- Because computers store numbers in binary, operations that involve flipping or shifting bits are very efficient.

- For example, multiplying by 2 is just shifting all bits to the left and adding a zero.

- This can be taken advantage of in many situations.

- Modern compilers will do this automatically.