Compilers and Programming Languages

IE 496 Lecture 4
Readings for This Lecture

- “Basics of Compiler Design”, Torben Ægidius 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.
Machine Language

- **Data handling and Memory operations**
  - set a register to a fixed constant value
  - move data from a memory location to a register, or vice versa.
  - read and write data from hardware devices

- **Arithmetic and Logic**
  - add, subtract, multiply, or divide the values of two registers, placing the result in a register
  - perform bitwise operations
  - compare two values in registers

Machine Language (cont.)

- **Control flow**
  - branch to another location in the program and execute instructions there
  - conditionally branch to another location if a certain condition holds
  - indirectly branch to another location, but save the location of the next instruction as a point to return to (a call)

Steps in Compilation

1. **Lexical analysis**: Divides the text into *tokens*

2. **Syntax analysis**: Generates the *syntax tree*

3. **Type checking**: Checks for type consistency

4. **Intermediate code generation**: Translates code to machine independent intermediate language

5. **Register allocation**: Assigns variables to registers

6. **Machine code generation**: Generates machines code (textual)

7. **Assembly and linking**: Creates final binaries
Lexical and Syntax Analysis

Parse tree (pruned)
Python add5() function

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.

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

• Python is an example of an interpreted language.

• You can call C/C++ from Python, so a good strategy is to use C++ for functions for which speed matters.
Intermediate Code Generation

- Intermediate code is similar to machine code except that it is machine independent.
- For example, such code assumes 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
  
  Example: $a[i] := a[i]+2$

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

  ```java
  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.

<table>
<thead>
<tr>
<th>Normal loop</th>
<th>After loop unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>int x;</td>
</tr>
<tr>
<td>for (x = 0; x &lt; 100; x++)</td>
<td>for (x = 0; x &lt; 100; x += 5)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>delete(x);</td>
<td>delete(x);</td>
</tr>
<tr>
<td>}</td>
<td>delete(x+1);</td>
</tr>
<tr>
<td>//.</td>
<td>delete(x+2);</td>
</tr>
<tr>
<td>//.</td>
<td>delete(x+3);</td>
</tr>
<tr>
<td>//.</td>
<td>delete(x+4);</td>
</tr>
<tr>
<td>//.</td>
<td>}</td>
</tr>
</tbody>
</table>

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 here to write 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;
    exit(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.