Lecture 15: Putting it all together

From parsing to code generation

How to make the computer understand?

• Write a program using a programming language
• Microprocessors talk in assembly language

Example (input program)

```c
int expr(int n)
{
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

Example (Output assembly code)

<table>
<thead>
<tr>
<th>Unoptimized Code</th>
<th>Optimized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushq %rbp</code></td>
<td><code>push %rbp</code></td>
</tr>
<tr>
<td><code>movq %rsp, %rbp</code></td>
<td><code>mov %edi, %eax</code></td>
</tr>
<tr>
<td><code>imull %edi, %eax</code></td>
<td><code>imull %edi, %eax</code></td>
</tr>
<tr>
<td><code>incl %edi</code></td>
<td><code>incl %edi</code></td>
</tr>
<tr>
<td><code>imull %edi, %eax</code></td>
<td><code>imull %edi, %eax</code></td>
</tr>
<tr>
<td><code>movl -4(%rbp), %eax</code></td>
<td><code>movl -4(%rbp), %eax</code></td>
</tr>
<tr>
<td><code>sall $2, %eax</code></td>
<td><code>sall $2, %eax</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>

Anatomy of a Computer

Lexical Analysis

• Lexical analyzer create tokens out of a text stream
• Tokens are defined using regular expressions
Examples of Regular Expressions

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Strings matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>“a”</td>
</tr>
<tr>
<td>a · b</td>
<td>“ab”</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>ε</td>
<td>“ε”</td>
</tr>
<tr>
<td>a*</td>
<td>“a” “aa” “aaa”…</td>
</tr>
<tr>
<td>(a</td>
<td>ε) · b</td>
</tr>
<tr>
<td>num = 0</td>
<td>1</td>
</tr>
<tr>
<td>posint = num · num*</td>
<td>“8” “6035”…</td>
</tr>
<tr>
<td>int = (ε</td>
<td>·) · posint</td>
</tr>
<tr>
<td>real = int · (ε</td>
<td>( · posint))</td>
</tr>
</tbody>
</table>

Lexical Analysis

- Lexical analyzer create tokens out of a text stream
- Tokens are defined using regular expressions
- Regular expressions can be mapped to Nondeterministic Finite Automatons (NFA)
  - by simple construction
- NFA is transformed to a DFA
  - Transformation algorithm
  - Executing a DFA is straightforward

Syntax Analysis (parsing)

- Defining a language using context-free grammars

Example: A CFG for expressions

```latex
<expr> → <expr> <op> <expr>
<expr> → ( <expr> )
<expr> → - <expr>
<expr> → num
<op> → +
<op> → *
```

Parse Tree Example

```
num * ( ( num + num ) )
```

Syntax Analysis (parsing)

- Defining a language using context-free grammars
- Classification of Grammars

- LR(0)
- SLR(1)
- LR(1)
- LR(k)
- regular
- LALR
- unambiguous
- Context free
Semantic Analysis

- Building a symbol table
- Static Checking
  - Flow-of-control checks
  - Uniqueness checks
  - Type checking
- Dynamic Checking
  - Array bounds check
  - Null pointer dereference check

Translation to Intermediate Format

- Goal: Remain Largely Machine Independent
  But Move Closer to Standard Machine Model
  - From high-level IR to a low-level IR
- Eliminate Structured Flow of Control
- Convert to a Flat Address Space

Code Optimizations

- Generate code as good as hand-crafted by a good assembly programmer
- Have stable, robust performance
- Abstract the architecture away from the programmer
  - Exploit architectural strengths
  - Hide architectural weaknesses
Compiler Project!
- You guys build a full-blown compiler from the ground up!!
- From decaf to working code

Compiler Derby
- Who has the fastest compiler in the east???
- Will give you the program 12 hours in advance
  - Test and make all the optimizations work
  - DO NOT ADD PROGRAM SPECIFIC HACKS!
- Wednesday, December 14th at 11:00AM
  - location TBA
    - refreshments provided

How will you use 6.035 knowledge?
- As an informed programmer
- As a designer of simple languages to aid other programming tasks
- As an engineer dealing with new computer architectures
- As a true compiler hacker

1. Informed Programmer
- Now you know what the compiler is doing
  - don’t treat it as a black box
  - don’t trust it to do the right thing!
- Implications
  - performance
  - debugging
  - correctness

2. Language Extensions
- In many applications and systems, you may need to:
  - implement a simple language
    - handle input
    - define an interface
    - command and control
  - extend a language
    - add new functionality
    - modify semantics
    - help with optimizations
2. Language Extensions

- What you learned in 6.035
  - define tokens and languages using regular expressions and CFGs
  - use tools such as jlex, lex, javacup, yacc
  - build intermediate representations
  - perform simple transformations on the IR

3. Computer Architectures

- Many special purpose processors
  - in your cell phone, car engine, watch, etc. etc.

- Designing new architectures
  - Adapting compiler back-ends for new architectures

3. Designing New Architectures

- Great advances in VLSI technology
  - very fast and small transistors
  - scaling up to billion transistors
  - but, slow and limited wires and I/O

- A computer architecture is a combination of hardware and compiler
  - need to know what a compiler can do and what hardware need to do
  - If compiler can do it don’t waste hardware resources.

3. Back-end support

- Every new architecture need a new backend

- Instruction scheduling
  - Even if the ISA is the same, different resource constrains
  - How to handle new features

- What do you learned in 6.035
  - Intermediate representations
  - Transforming/optimizing the IR
  - Process of generating assembly from a high-level IR
  - Assembly interface issues (eg: calling conventions)
  - Register allocation issues
  - Code scheduling issues
4. Compiler Hacking

- Theory
- Algorithms
- Implementation

Theory:
- Develop general, abstract concepts
- Prove correctness, optimality etc.

Examples:
- parser theory
- lattices and data-flow
- abstract interpretation
- The language ML

Algorithms:
- Design a solution to a given problem
  (Mostly new optimizations)
- Use many techniques such as graph theory, number
  theory, etc.
- May have to limit the scope and find good
  heuristics

Examples
- partial redundancy elimination
- register allocation by graph coloring
- using multi-granular (MMX) operations

Implementation:
- Develop a new compiler
- Issues of designing a very complex software
- Putting theory and algorithms into practice

Examples
- A JIT for Java
- A query optimization engine for SQL
- A rasterizer for postscript

Where to Look for Current Research?

- PLDI – Programming Languages Design and Implementation
  Conference
- Code Generation / Machine specific
  – Micro Conference
  – ASPLOS – Architecture Support for Programming Languages and
    Operating Systems
  – CGO – Code Generation and Optimization
- Language Theory
  – POPL – Principles of Programming Languages
  – OOPSLA – Object Oriented Programming Systems Languages and
    Applications
- Program Analysis
  – SAS – Static Analysis Symposium
  – PPOPP – Principles and Practice of Parallel Programming