CSCI-GA.2130-001
Compiler Construction

Lecture 1: Introduction

Hubertus Franke
(frankeh@cs.nyu.edu)
Who Am I?
What am I doing here?

• Hubertus Franke
  – Ph.D. EE Vanderbilt University 1992
  – Diplom Informatik, Technical University Karlsruhe, Germany, 1987
  – Research Staff Member and Manager Operating Systems at
    IBM T.J. Watson Research Center in Yorktown Heights, NY (since 1993)

• Research:
  – Operating Systems:
    • Linux, AIX, object oriented OS (K42)
    • Scheduling, memory management, ..
  – Computer Architecture:
    • Multicore processors and Systems on a chip
  – High Performance Computing:
    • MPI (Message Passing Interfaces), Gang Scheduling
  – Cloud Computing
  – Software Engineering, Compilers and Robotics.
  – ~100 publications in these areas, ~35 Patents

• Classes Taught at NYU:
  – Operating Systems (3x)
  – Computer Architecture (1x)
  – Computer Games (1x)
  – Compilers (0x)
Logistics

• **Class Website**
  – Short term announcement will be on the web (top website)

• **Class List Setup:** (subscribe and use please)
  – csci_ga_2130_001_sp13@cs.nyu.edu

• **Office hours:**
  – Tuesdays 6:00-7:00pm or after class
  – Room: WWH 328 or WWH 320 (@Z)

• **Teaching Assistant:**
  – Ravi Chotrani ([ravi.chotrani@nyu.edu](mailto:ravi.chotrani@nyu.edu))
  – More likely to be added
Informal Goals of This Course

• To get more than an A
• To learn compilers and enjoy it
• To use what you have learned in MANY different contexts
Informal Goals

• Think, Learn
Informal Goals

• Think, Learn, Apply ....

• And have some Fun (all at the same time)
Formal Goals of This Course

• What exactly is this thing called compiler?

• How does the compiler interact with the hardware and programming languages?

• Different phases of a compiler

• Develop a simple compiler
Grading

• The project (labs): 60%
  – Due several lectures later
  – Several parts
  – Mostly programming and dealing with tools
  – Can do on your own machine or NYU machines
  – Must be sure that they work on NYU machines
    • This is YOUR RESPONSIBILITY:
      – energon1, energon2
  – Getting help: office hours and mailing list

• Final Exam: 40%

• Homework (occasionally): 0%
The Course Web Page

• Lecture slides
• Info about mailing list, labs, … .
• Useful links (manuals, tools, book errata, … ).
• Short term notices
  – (e.g. room change, … etc )
The Book

• The classic definitive compiler technology text
• It is known as the Dragon Book
• A knight and a dragon in battle, a metaphor for conquering complexity
• We will cover mostly chapters 1 - 8
What Is A Compiler?

- Programming languages are *notations* for describing computations to people and to machines.
- Machines do not understand programming languages.
- So a software system is needed to do the translation.
- This is the *compiler*.

---

BEAR FACTS by Burke

He always liked to open the Annual Programmer's Conference with a joke in binary.

0100010110110010
1101 1110 10 10 1110
Compiler writers have to track new language features.

Compiler writers must take advantage of new hardware features.

• Optimizing compilers are hard to build
• Excellent software engineering case study
• Theory meets practice
Why Compilers Are So Important?

- Compiler writers have influence over all programs that their compilers compile
- Compiler study trains good developers
- We learn not only how to build compilers but the general methodology of solving complex and open-ended problems
- Compilation technology can be applied in many different areas
  - Binary translation
  - Hardware synthesis
  - DB query interpreters
  - Compiled simulation
So What Is An Interpreter?

- Better error diagnostics than compiler
- Slower than machine language code directly executed on the machine
source program

Translator

Compilation

intermediate program

Virtual Machine

Interpretation

output

input
High Level Language → Compiler (translator) → Assembly Language → Assembler (translator) → Machine Language → Control Unit (Interpreter) → Microarchitecture → Microsequencer (Interpreter) → Logic Level

Device Level → Semiconductors → Quantum
Compiler Is Not A One-Man-Show!

Why not letting the compiler generate machine code directly?
Let's Have A Closer Look: Phases of A Compiler

Front-end (Analysis Phase)

Back-End (Synthesis Phase)
Lexical Analysis

• Reads stream of characters: your program
• Groups the characters into meaningful sequences: lexemes
• For each lexeme, it produces a token 
  <token-name, attribute value>
• Blanks are just separators and are discarded
• Filters comments
• Recognizes: keywords, identifier, numbers, …
position = initial + rate * 60

Lexical Analyzer

(id, 1) (id, 2) (id, 3) (+) (*) (60)

Token stream

token name

Entry into the symbol table

<table>
<thead>
<tr>
<th></th>
<th>position</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td>...</td>
</tr>
</tbody>
</table>

SYMBOL TABLE
Syntax Analysis (Parsing)

• Uses tokens to build a tree
• The tree shows the grammatical structure of the token stream
• A node is usually an operation
• Node’s children are arguments
This is usually called a **syntax tree**
Semantic Analysis

- Uses the syntax tree and symbol tables
- Gathers type information
- Checks for semantic consistency errors
Intermediate Code Generation

• Code for an abstract machine
• Must have two properties
  – Easy to produce
  – Easy to translate to target language
• Called three address code
• One operation per instruction at most
• Compiler must generate temporary names to hold values
Intermediate Code Optimization (Optional)

- Machine independent
- Optimization so that better target code will result

Instead of `inttofloat (60)`

Instead of `inttofloat` we can use `60.0` directly

Do we really need `t2`?

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```
Code Generation

- Input: the intermediate representation
- Output: target language
- This is the backend, or synthesis phase
- Machine dependent

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generator

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```
Qualities of a Good Compiler

- **Correct**: the meaning of sentences must be preserved
- **Robust**: wrong input is the common case
  - compilers and interpreters can’t just crash on wrong input
  - they need to diagnose all kinds of errors safely and reliably
- **Efficient**: resource usage should be minimal in two ways
  - the process of compilation or interpretation itself is efficient
  - the generated code is efficient when interpreted
- **Usable**: integrate with environment, accurate feedback
  - work well with other tools (editors, linkers, debuggers, . . .)
  - descriptive error messages, relating accurately to source
Compilers Optimize Code For:

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging
Compiler Construction Tools

- **Scanner generators**: produce lexical analyzer
- **Parser generators**: automatically produce syntax analyzer
- **Syntax-directed translation engines**: collection of routines for walking a parse tree and generate intermediate code
- **and many more ...**
A Little Bit of History

Eckert and Mauchly

- 1\textsuperscript{st} working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft\textsuperscript{3}
A Little Bit of History

- Maurice Wilkes

EDSAC 1 (1949)

http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/

1st stored program computer
650 instructions/sec
1,400 ft³
A Little Bit of History

- 1954 IBM developed 704
- All programming done in assembly
- Software costs exceed hardware costs!
A Little Bit of History

- Fortran I (project 1954-57)
- The main idea is to translate high level language to assembly
- Many thought this was impossible!
- In 1958 more than 50% of software in assembly!
- Development time halved!

John Backus
(December 3, 1924 – March 17, 2007)
A Glimpse At Programming Language Basics

• Static/Dynamic distinction

• Environments and states
  – Environment: mapping names to locations
  – States: mapping from locations to values

• Procedures vs Functions

• Scope
main() {

    int a = 1;
    int b = 1;

    {
        int b = 2;

        {
            int a = 3;
            cout << a << b;

        }

        {
            int b = 4;
            cout << a << b;

        }

        cout << a << b;

    }

    cout << a << b;
}

}
Roadmap

• Today we have mostly discussed chap 1
• Chapter 2 gives an overview of the different phases of a compiler by building the front-end of a simple compiler
• Chapters 3-8 fill the gaps