Compiler Construction: Introduction and History
INTRODUCTION AND ADMINISTRATION
Administrivia

Instructor: Garrett Morris
Office: Eaton 2028
   Tuesday 12:30-2:30 PM
   Thursday 3:00-5:00 PM

Lab instructor: April Wade
Office: Eaton 3025
   Thursday: 1:00-2:30 PM
   Friday: 12:00-1:30 PM

Web page: http://ittc.ku.edu/~garrett/eecs665s18
The point

*Form* and *function* of programming languages.
The point

Syntax and semantics of programming languages.
The point: syntax

Σύνταξις, orderly or systemic arrangement

<table>
<thead>
<tr>
<th><strong>Theory</strong></th>
<th><strong>Implementation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular languages,</td>
<td>Lexing, lexx, alex</td>
</tr>
<tr>
<td>regular expressions</td>
<td></td>
</tr>
<tr>
<td>Context-free languages, finite automata</td>
<td>Parsers, yacc, happy</td>
</tr>
</tbody>
</table>

I don’t care (very much) about syntax
The point: semantics

Σημαντικός, significant, (something that) shows or signifies

<table>
<thead>
<tr>
<th>Directly</th>
<th>By translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 \downarrow \lambda x. e$</td>
<td>$[e_1 e_2] = <a href="%5Be_2%5D">e_1</a>$</td>
</tr>
<tr>
<td>$e_2 \downarrow v$</td>
<td></td>
</tr>
<tr>
<td>$e[v/x] \downarrow w$</td>
<td></td>
</tr>
<tr>
<td>$e_1 e_2 \downarrow w$</td>
<td></td>
</tr>
<tr>
<td>Interpreters</td>
<td>Compilers</td>
</tr>
<tr>
<td>EECS 662</td>
<td>EECS 665</td>
</tr>
</tbody>
</table>

We all care (implicitly) about semantics
The point: learning

• How to interpret text as (high-level) programs
• How to assure semantic properties of programs
• How high-level programs are implemented in machine language
• (A subset of) Intel X86 architecture
• Deeper understanding of code
• Deeper understanding of common compilation tools (gcc, llvm, &c)
• Manipulating complex, data structures (recursively)
• Programming (functionally, in Haskell)
Not the point: grading

<table>
<thead>
<tr>
<th>Out of class</th>
<th>In class</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labs (~10)</td>
<td>Midterm</td>
<td>15%</td>
</tr>
<tr>
<td>30%</td>
<td>Final</td>
<td>20%</td>
</tr>
<tr>
<td>Homeworks (~4)</td>
<td>Quizzes</td>
<td>5%</td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>Total</td>
<td>40%</td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You must pass both columns to pass the course.
Haskell

1971: Robin Milner starts the LCF project (at Stanford)
1973: Implementation of LCF (at Edinburgh) includes “meta language” (ML)
1987-90: Haskell project aims to standardize multiple dialects of “lazy” ML
1998: Haskell ‘98 report defines (effectively) the current version of the language.
Haskell

Functional & pure
- Programs manipulate values, rather than issue commands
- Functions and computations are first-class entities
- Side effects explicit in terms and types

Strongly & statically typed
- Compiler guarantees that programs meet correctness conditions
- Good support for generic types and type inference
- User-defined “algebraic” data type with pattern matching
Haskell

**Functional & pure**
- Programs manipulate values, rather than issue commands
- Functions and computations are first-class entities
- Side effects explicit in terms and types

**Strongly & statically typed**
- Compiler guarantees that programs meet correctness conditions
- Good support for generic types and type inference
- User-defined “algebraic” data type with pattern matching

*FP languages are force multipliers*
Resources

Recommended:
• Appel, *Modern Compiler Implementation in ML*

Other compiler texts:
• Aho, Lam, Sethi, Ullman, *Compilers-Principles, Techniques & Tools*

Haskell tutorials:
• Lipovača, *Learn you a Haskell...*
• O’Sullivan, Stewart, Goerzen, *Real World Haskell*
• Allen, Moronuki, *Haskell Programming from First Principles*
WHAT IS A COMPILER?
History

1940s: computers programmed in assembly
1951-2: Grace Hopper developed A-0 for the UNIVAC I
1957: FORTRAN compiler developed by team led by John Backus
1960s: development of the first bootstrapping compiler for LISP
Assigning meaning to code

- Single step to give meaning to programs
- More common than you might think
  - JavaScript
  - Ruby / Python / other scripting languages
  - JBC / CIL / other VMs
Source languages

Optimized for understanding

– Expressive: matches human ideas of syntax and meaning
– Redundant: includes information to guide compilation and catch errors
– Abstract: details of computation not fully determined by code

```c
#include <stdio.h>

int factorial(int n) {
    int acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}

int main(int argc, char *argv[]) {
    printf("factorial(6) = %d\n", factorial(6));
}
```
Assigning meaning to code

- Gives meaning to program by translation
- Frequently targeting low-level code
- But doesn’t have to:
  - Source-to-source translations
  - Various compilers target JavaScript
Machine languages

Optimized for execution

- Inexpressive: expressions match hardware operations
- Explicit: very little implicit information about program meaning
- Concrete: abstractions & information about intent is lost

_factorial:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl 8(%ebp), %eax
  movl %eax, -4(%ebp)
  movl $1, -8(%ebp)
LBB0_1:
  cmpl $0, -4(%ebp)
  jle LBB0_3
  movl -8(%ebp), %eax
  imull -4(%ebp), %eax
  movl %eax, -8(%ebp)
  movl -4(%ebp), %eax
  subl $1, %eax
  movl %eax, -4(%ebp)
  jmp LBB0_1
LBB0_3:
  movl -8(%ebp), %eax
  addl $8, %esp
  popl %ebp
  retl
Assigning meaning to code

• Compilation usually divided into stages
• Intermediate representations optimized for different program manipulations
• Key idea: composition of translations

- Source language
- Intermediate languages
- Machine language

- Parsing
- Static analysis
- Code generation
- Execution

Input ➔ Output
Compilers by composition

Source language
- Lexing: Stream of characters $\rightarrow$ stream of tokens
- Parsing: $\rightarrow$ Abstract syntax tree
- Desugaring: $\rightarrow$ Simplified syntax tree
- Type checking: $\rightarrow$ Type-annotated syntax tree
- Control-flow analysis: $\rightarrow$ Control-flow graph
- Data-flow analysis: $\rightarrow$ Interference graph
- Register allocation: $\rightarrow$ Assembly
- Code emission: Assembly

Abstract syntax
Compilers by composition

- Higher level languages may require more steps
- Smaller passes simplify understanding & maintenance
Future directions

• Compiler correctness & certification
• JIT compilation and virtual machines
• Modular and generic programming