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

Theory	*Implementation*
Regular languages, regular expressions	Lexing, `lexx`, `alex`
Context-free languages, finite automata	Parsers, `yacc`, happy

I don’t care (very much) about syntax

The point: semantics

Σημαντικός, significant, (something that) shows or signifies

Directly	*By translation*
$$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow w}{e_1 e_2 \Downarrow w}$$	$$[\![e_1 e_2]\!] = [\![e_1]\!]([\![e_2]\!])$$
Interpreters	Compilers
EECS 662	EECS 665

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

Out of class		*In class*	
Labs (~10)	30%	Midterm	15%
Homeworks (~4)	30%	Final	20%
		Quizzes	5%
Total	60%	Total	40%

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?

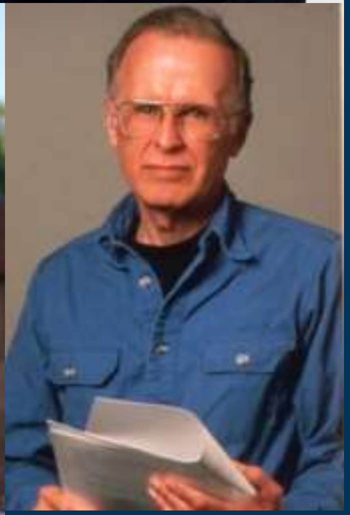
History

1940s: computers programmed in assembly

1951-2: Grace Hopper developed A-0 for the UNIVAC I

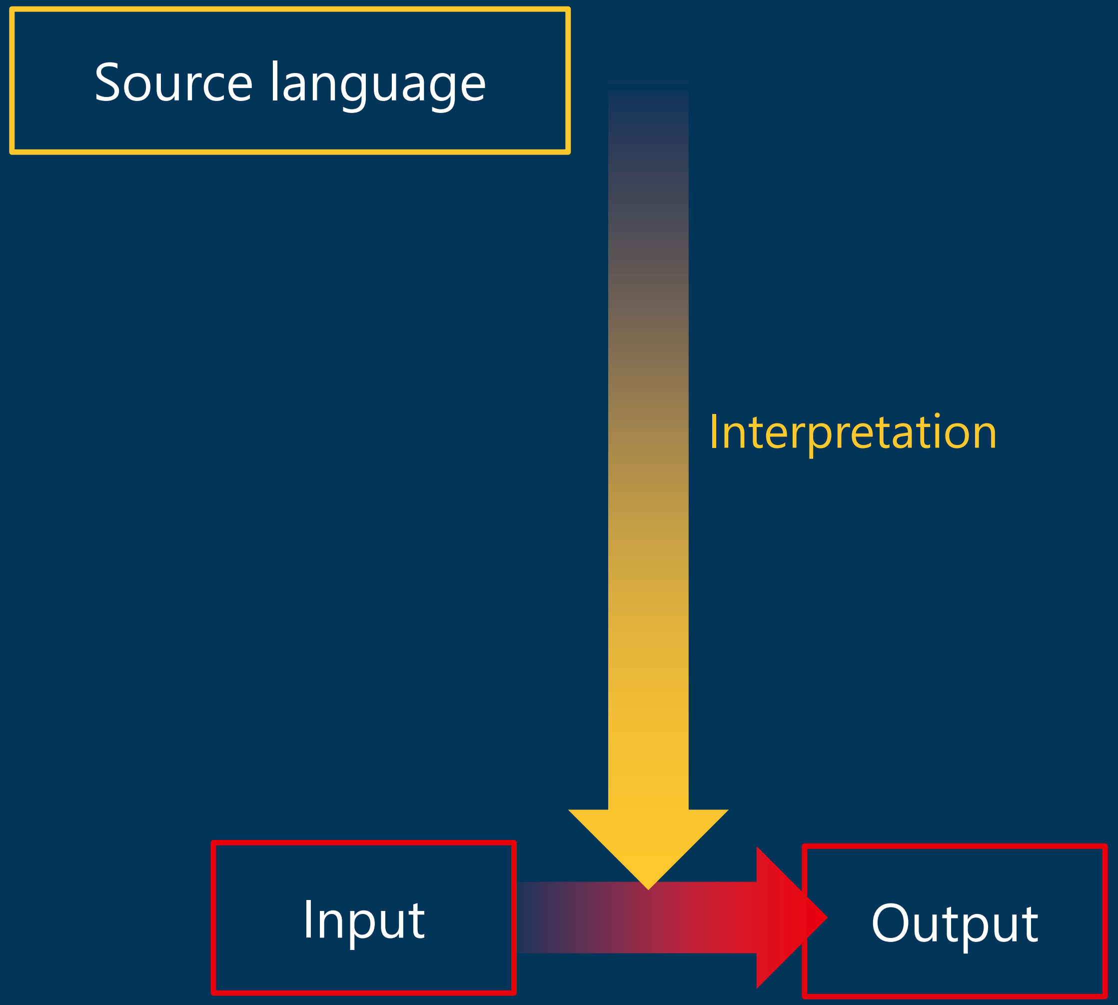
1957: FORTRAN compiler developed by team led by John Backus

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

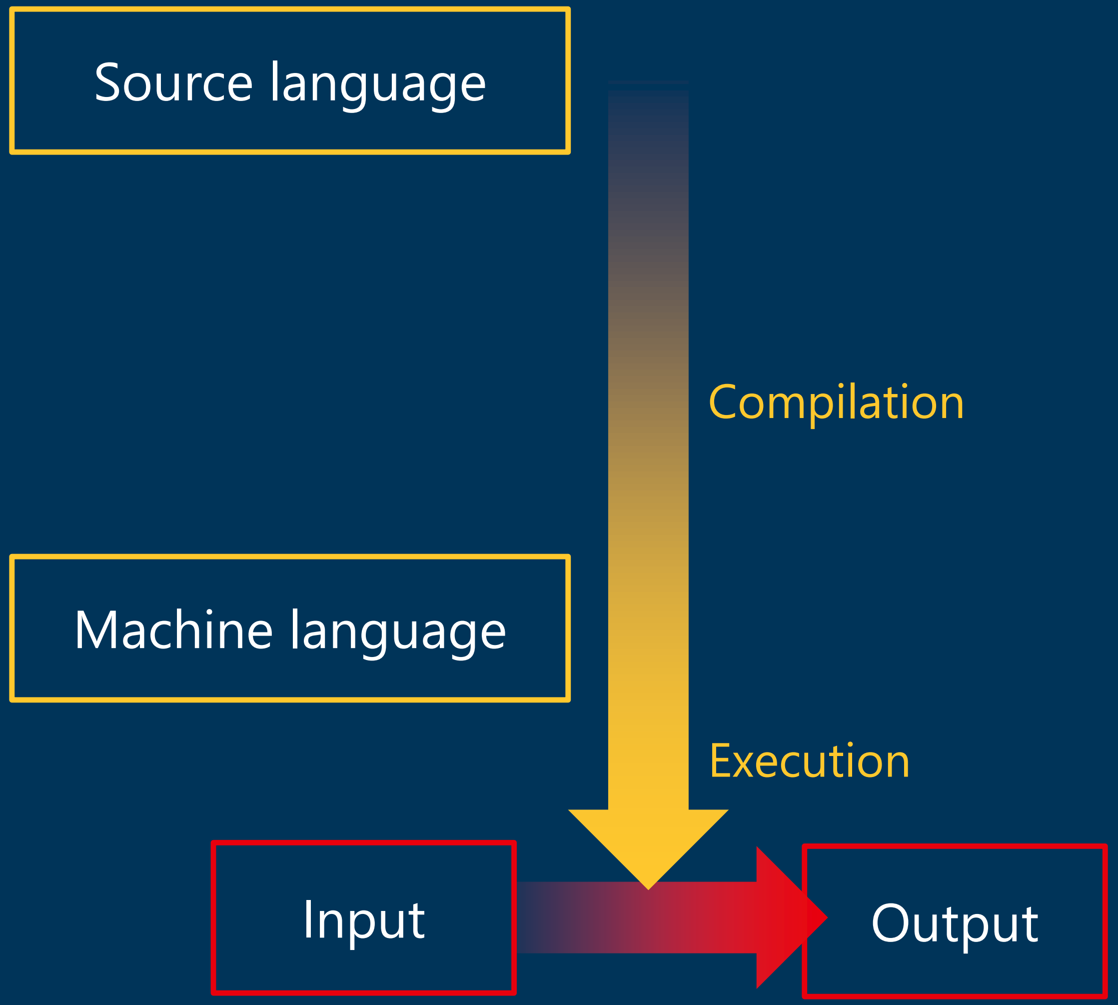
```
#include <stdio.h>

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

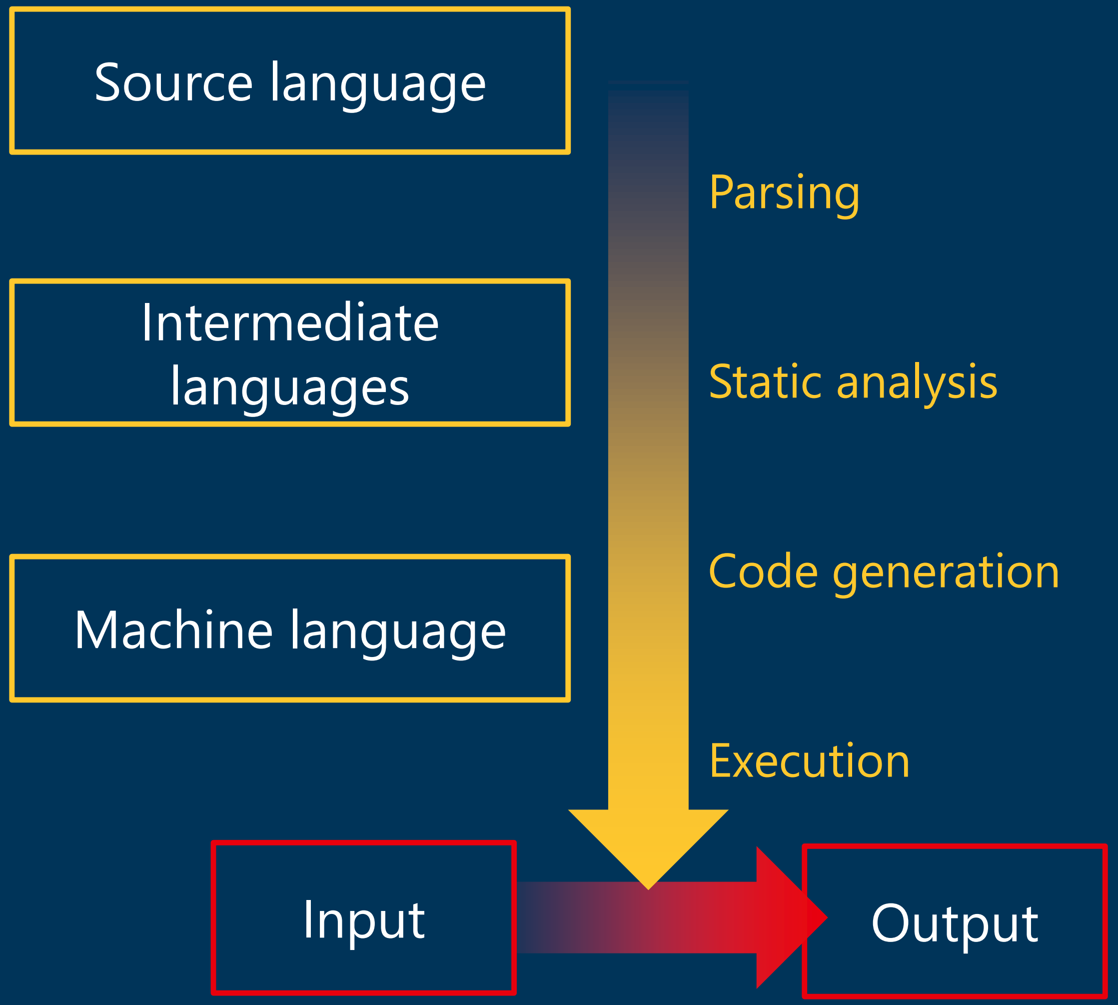
Optimized for execution

- 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



Compilers by composition

Source language

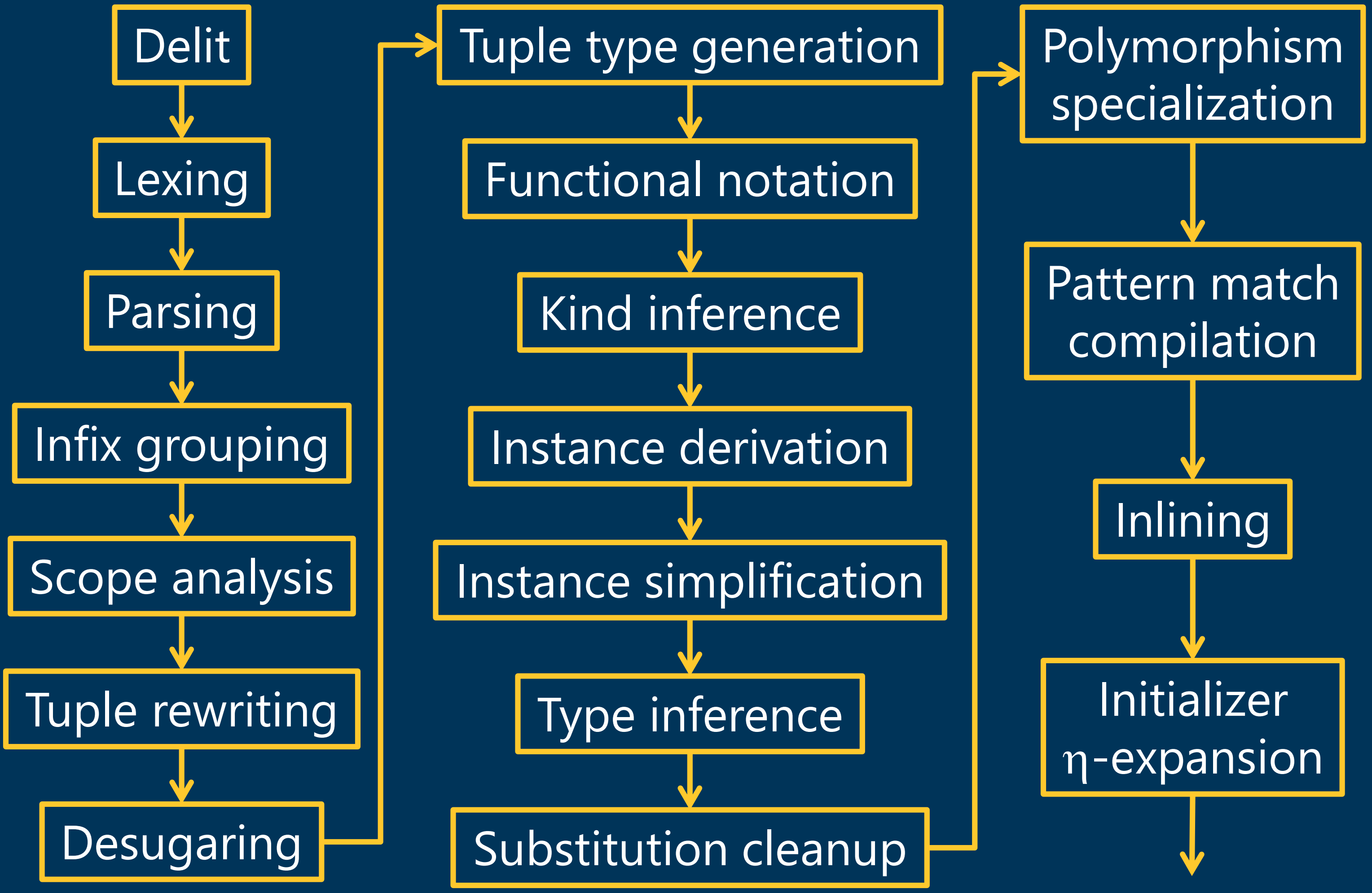
Abstract syntax

Assembly

Lexing	Stream of characters → stream of tokens
Parsing	→ Abstract syntax tree
Desugaring	→ Simplified syntax tree
Type checking	→ Type-annotated syntax tree
Control-flow analysis	→ Control-flow graph
Data-flow analysis	→ Interference graph
Register allocation	→ Assembly
Code emission	

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