CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins for all the help and inspiration in preparing these course materials and assignments.

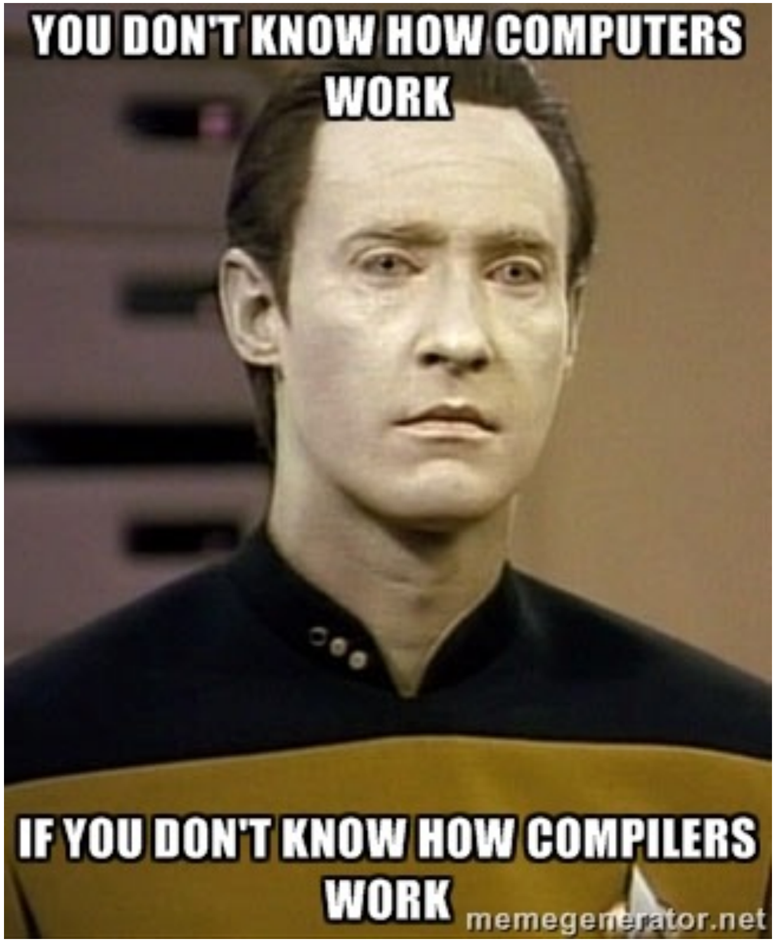
Credits For Course Material

- **Big thank you to UW CSE faculty member, Hal Perkins**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Agenda

- Why do we care about compilers?
- A Structure of a Compiler
- Front End
  - Scanner
  - Parser
- Back End
- Interpreters and Compilers
- Why Study Compilers
- Some History
- Upcoming Attractions and Activities

Why Do We Care About Compilers?



[Meme credit:https://cdn.meme.am]

Why Do We Care About Compilers?

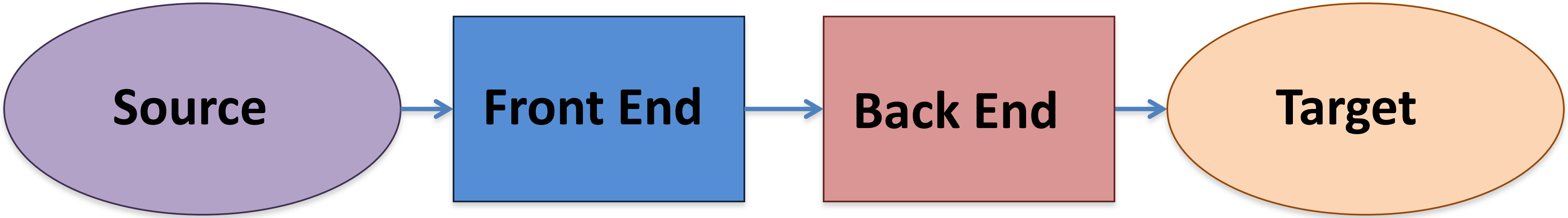
- **Question** - how do you execute something like this:

```
int numFives = 0;
int k = 0;
while (k < length) {
if (a[k] == 5) {
numFives++; }
}
```

- **Difficulty: computer knows only zeros and ones (the encodings of data and instructions)**

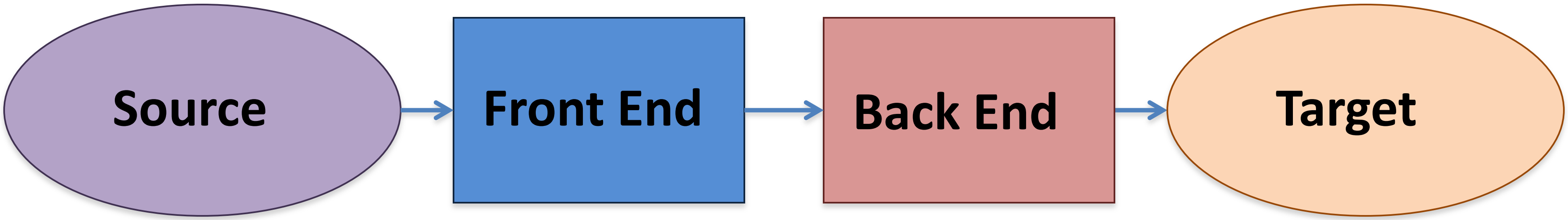
A Structure of a Compiler

- At a high level, a compiler has two pieces:
  - **Front end – analysis**
    - Read source program, and discover its structure and meaning
  - **Back end – synthesis**
    - Generate equivalent target language program



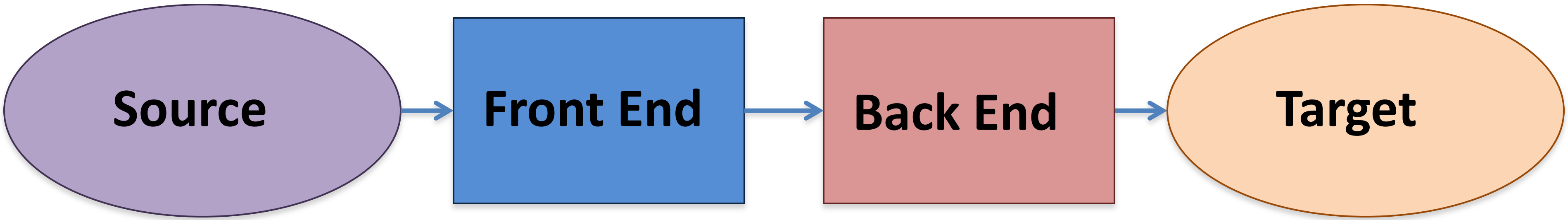
Compiler Must...

- Recognize legal programs (& complain about illegal ones)
- Generate correct code
  - ***Compiler can attempt to improve ("optimize") code, but cannot change behavior (meaning)***
- Manage runtime storage of all variables/data
- Agree with OS & linker on target format

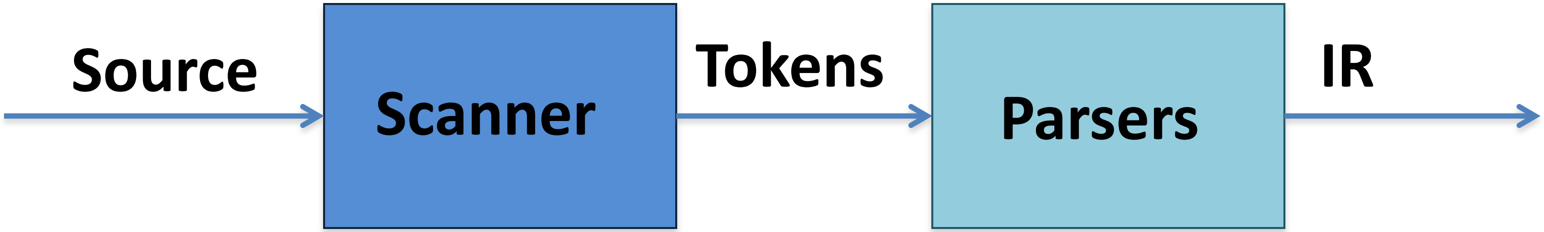


Implications

- Phases communicate using some sort of **Intermediate Representation(s) (IR)**
  - Front end maps source into IR
  - Back end maps IR to target machine code
- **Often multiple IRs** – higher level at first, lower level in later phases

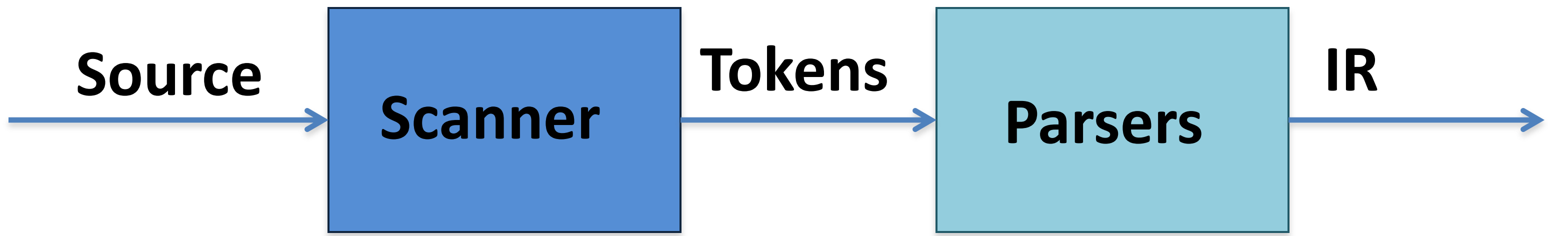


Compiler: Front End



- **Front end is usually split into two parts:**
  1. **Scanner** – responsible for converting character stream to token stream: keywords, operators, variables, constants
     - Also: strips out white space, comments
  2. **Parser** - reads token stream; generates IR
     - Either here or shortly after, perform semantics analysis to check for things like type errors

Compiler: Front End



- Front end is usually split into two parts:
  - **Scanner** – responsible for converting character stream to token stream
  - **Parser** – reads token stream; generates IR
- Both of these can be generated automatically
  - Use a formal grammar to specify the source language
  - Tools read the grammar and generate scanner & parser (lex/yacc or flex/bison for C/C++, JFlex/CUP for Java)

Scanner Example

- **Input Text:**

```
// this statement does very little
if (x >= y) y = 42;
```

- **Token Stream:**

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

- Tokens are atomic items, not character strings
- Comments & whitespace are *not* tokens (in most languages)
  - ***Counterexamples: Python indenting, Ruby newlines***
- Tokens may carry associated data (e.g., int value, variable name)

Parser Output - IR

- Given a token stream from a scanner, the parser must produce output that captures the meaning of the program
- Most common output from a parser is an **Abstract Syntax Tree (AST)**
  - Represents the essential meaning of program without syntactic noise
  - Nodes are operations, children are operands
- **Many different forms**
  - Engineering tradeoffs have changed over time
  - Tradeoffs (and IRs) can also vary between different phases of a single compiler

Parser Example

- **Input Text:**

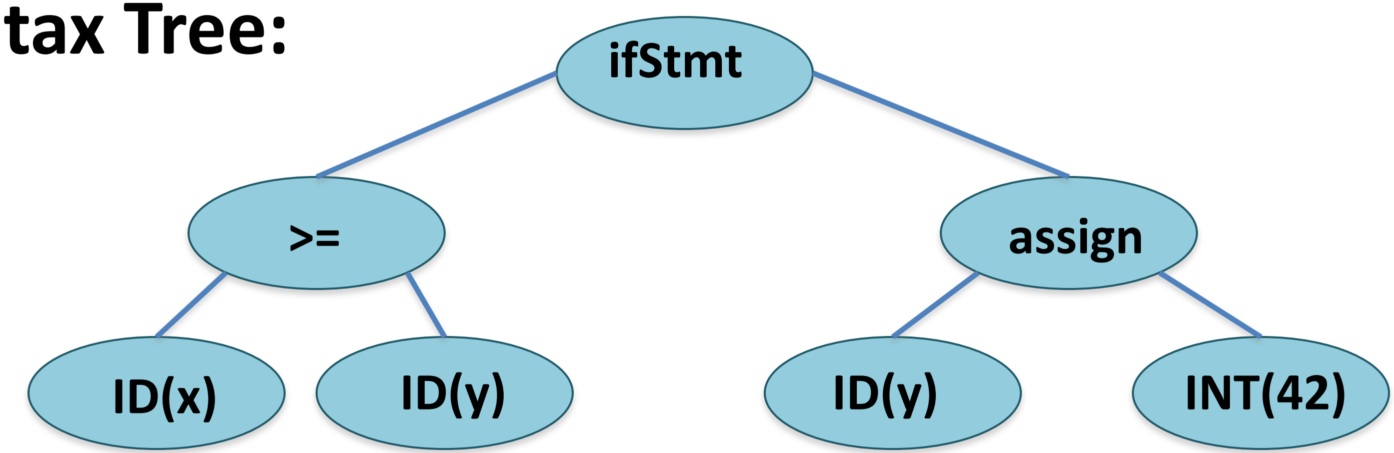
```
// this statement does very little
if (x >= y) y = 42;
```

- **Token Stream:**

IF | LPAREN | ID(x) | GEQ | ID(y)

RPAREN | ID(y) | BECOMES | INT(42) | SCOLON

- **Abstract Syntax Tree:**



Static Semantic Analysis (SSA)

- During or after parsing, **check that the program is legal and collect info for the back end**
  - Type checking
  - Check language requirements like proper declarations
  - Preliminary resource allocation
- Collect other information needed by back end analysis and code generation
- **Key data structure: Symbol Table(s)**
  - Maps names -> meaning/types/details

Compiler: Back End

- On the back end, compiler:
  - Translates IR into target machine code
  - Should produce **"good" code**
    - **"good"** = fast, compact, low power (pick some)
  - Optimization phase translates correct code into semantically equivalent "better" code
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

Back End Structure

- Typically split into two major parts
  - **“Optimization” – code improvement**
    - Examples: common sub-expression elimination, constant folding, code motion (move invariant computations outside of loops)
    - Optimization phases often interleaved with analysis
  - **Target Code Generation (machine specific)**
    - Instruction selection & scheduling, register allocation
- Usually walks the AST to generate lower-level intermediate code before optimization

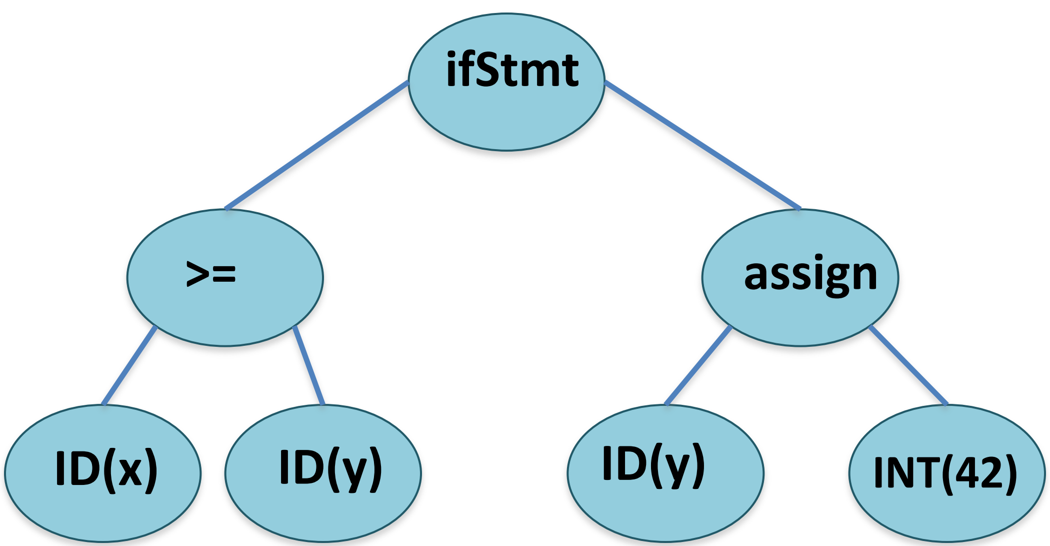
The Result

- **Input Text:**

```
// this statement does very little
if (x >= y) y = 42;
```

- **Token Stream:** IF | LPAREN | ID(x) | GEQ | ID(y) | RPAREN | ID(y) | BECOMES | INT(42) | SCOLON

- **Abstract Syntax Tree:**



- **Output:**

```
mov   eax,[ebp+16]
cmp   eax,[ebp-8]
jl    L17
mov   [ebp-8],42
L17:
```

Compilers and Interpreters

- Programs can be compiled or interpreted (or sometimes both)
- **Compiler**
  - A program that translates a program from one language (the *source*) to another (the *target*)
    - ***Languages are sometimes even the same(!)***
- **Interpreter**
  - A program that reads a source program and produces the results of executing that program on some input

Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and “understand” it

→ ***front-end analysis phase***

```
While (k < length) { <nl> <tab> if (a
  [k] == 5) <nl> <tab> <tab>{numFives+
  +;} <nl> <tab>}
```

Compiler

- Read and analyze **entire program**
- Translate to semantically equivalent program in another language
    - Presumably easier or more efficient to execute
- **Offline process**
- Tradeoff: compile time overhead
    - (preprocessing) vs execution performance

Typically Implemented with Compilers

- FORTRAN, C, C++, COBOL, and many other programming languages
- (La)TeX, SQL (databases), VHDL, many others
- Particularly appropriate if significant optimization wanted/needed

Interpreter

- Typically implemented as an **“execution engine”**
- Program analysis interleaved with execution:

```
running = true;
while (running) {
analyze next statement;
execute that statement; }
```

- Usually requires repeated analysis of individual statements (particularly in loops and functions)
  - But hybrid approaches can avoid some of this overhead
- But: immediate execution, good debugging/interaction…

Often Implemented with Interpreters

- Javascript, PERL, Python, Ruby, awk, sed
- Shells (bash),
- Scheme/Lisp/ML/OCaml,
- postscript/pdf,
- machine simulators
- **Particularly efficient if interpreter overhead is low relative to execution cost of individual statements**
  - But even if not (machine simulators), flexibility, immediacy, or portability may be worth it

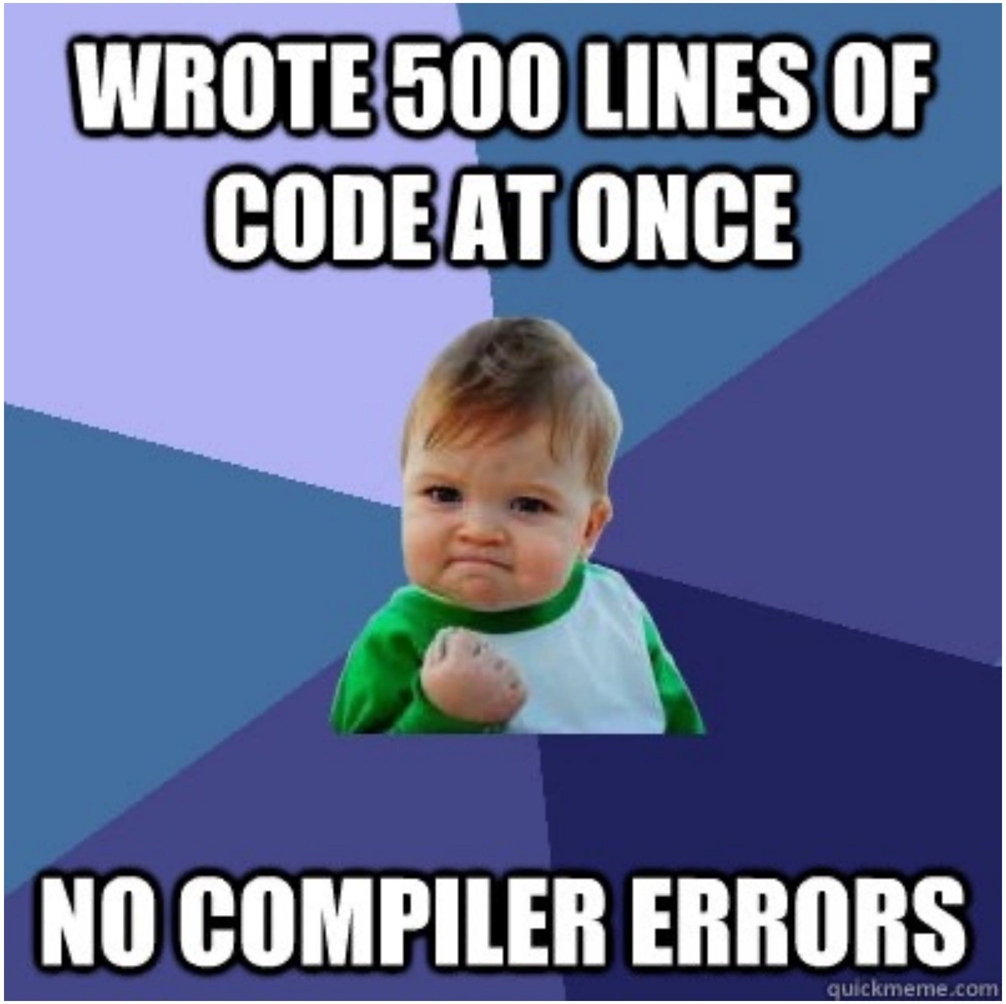
Hybrid Approaches

- Compiler generates byte code intermediate language, e.g., compile Java source to Java Virtual Machine .class files, then
- Interpret byte codes directly, or
- Compile some or all byte codes to native code
  - **Variation: Just-In-Time compiler (JIT)** – detect hot spots & compile on the fly to native code
- Also widely use for Javascript, many functional and other languages (Haskell, ML, Racket, Ruby), C# and Microsoft Common Language Runtime

Why Study Compilers?

[Meme credit: quickmeme.com]

Why Study Compilers?



[Meme credit: quickmeme.com]

Why Study Compilers - 1

- **Become a better programmer(!)**
- Insight into interaction between languages, compilers, and hardware
- Understanding of implementation techniques, and how code maps to hardware
- Better intuition about what your code does
- Understanding how compilers optimize code helps you write code that is easier to optimize
  - And avoid wasting time doing “optimizations” that the compiler will do as well or better (particularly if you don’t try to get too clever)

Why Study Compilers - 2

- **Compiler techniques are everywhere**
- Parsing (“little” languages, interpreters, XML)
- Software tools (verifiers, checkers, ...)
- Database engines, query languages
- AI, etc.: domain-specific languages
- Text processing
  - Tex/LaTex -> dvi -> Postscript -> pdf
- Hardware: VHDL; model-checking tools
- Mathematics (Mathematica, Matlab, SAGE)

Why Study Compilers - 3

- **Cool blend of theory and engineering**
  - Lots of beautiful theory around compilers
    - Parsing, scanning, static analysis
  - Interesting engineering challenges and tradeoffs, particularly in optimization (code improvement)
    - Ordering of optimization phases
    - What works for some programs can be bad for others
  - Plus some very difficult problems (NP-hard or worse)
    - E.g., register allocation is equivalent to graph coloring
    - Need to come up with good-enough approximations and heuristics

Why Study Compilers - 4

- **Draws ideas from many parts of CS**
  - **AI:** Greedy algorithms, heuristic search
  - **Algorithms:** graphs, dynamic programming, approximation
  - **Theory:** Grammars, DFAs and PDAs, pattern matching, fixed-point algorithms
  - **Systems:** Allocation & naming, synchronization, locality
  - **Architecture:** pipelines, instruction set use, memory
  - Hierarchy management, locality

Some History

[Meme credit: liucs.net]

Some History - 1

- **1950’s. Existence proof**
  - FORTRAN I (1954) – competitive with hand-optimized code
- **1960’s**
  - New languages: ALGOL, LISP, COBOL, SIMULA
  - Formal notations for syntax, esp. BNF
  - Fundamental implementation techniques
    - Stack frames, recursive procedures, etc.

Some History - 2

- **1970's**
  - Syntax: formal methods for producing compiler front-ends; many theorems
- **Late 1970's, 1980's**
  - New languages (functional; object-oriented - Smalltalk)
  - New architectures (RISC machines, parallel machines, memory hierarchy issues)
  - More attention to back-end issues

Some History - 3

- **1990s**
  - Techniques for compiling objects and classes, efficiency in the presence of dynamic dispatch and small methods (Self, Smalltalk – now common in JVMs, etc.)
  - Just-in-Time compilers (JITs)
  - Compiler technology critical to effective use of new hardware (RISC, parallel machines, complex memory hierarchies)

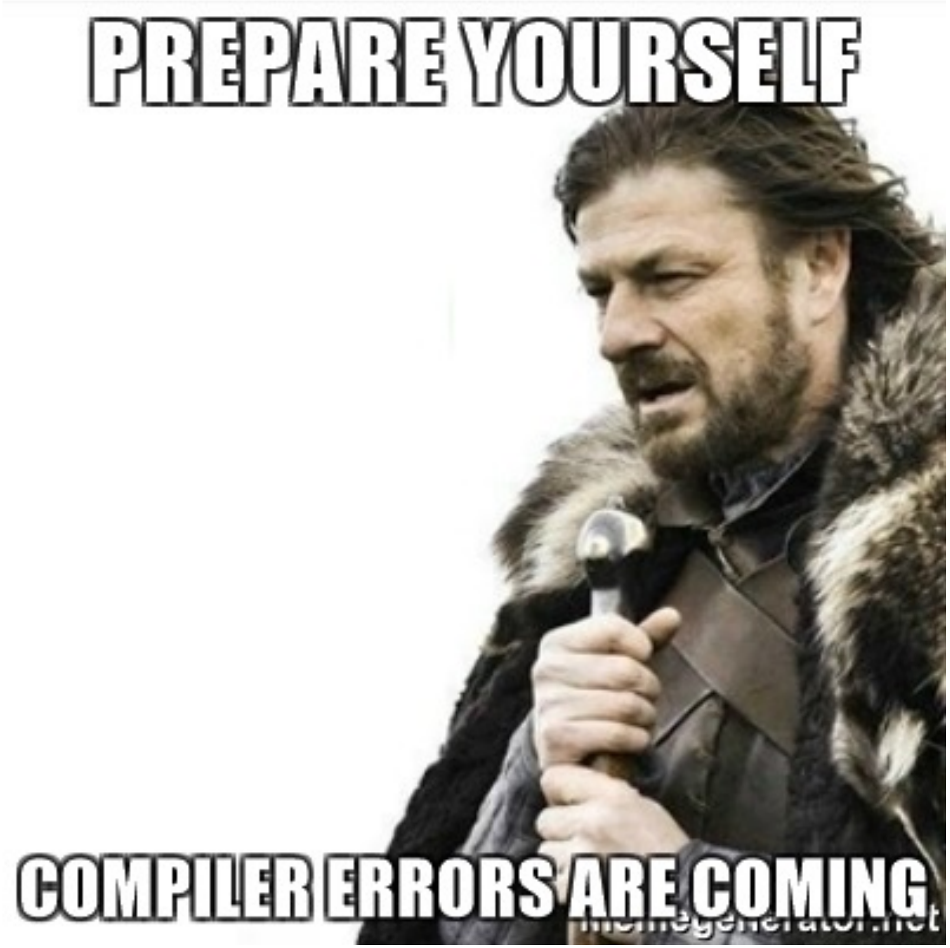
Some History - 4

- **Last decade**
  - Compilation techniques in many new places
    - Software analysis, verification, security
  - ***Phased compilation*** – blurring the lines between “compile time” and “runtime”
  - Using machine learning techniques to control optimizations(!)
  - Dynamic languages – e.g., JavaScript, ...
  - The new 800 lb gorilla - multicore

Compilers (and Related) Turing Awards

- 1966 Alan Perlis
- 1972 Edsger Dijkstra
- 1974 Donald Knuth
- 1976 Michael Rabin and Dana Scott
- 1977 John Backus
- 1978 Bob Floyd
- 1979 Ken Iverson
- 1980 Tony Hoare
- 1984 Niklaus Wirth
- 1987 John Cocke
- 1991 Robin Milner
- 2001 Ole-Johan Dahl and Kristen Nygaard
- 2003 Alan Kay
- 2005 Peter Naur
- 2006 Fran Allen
- 2008 Barbara Liskov

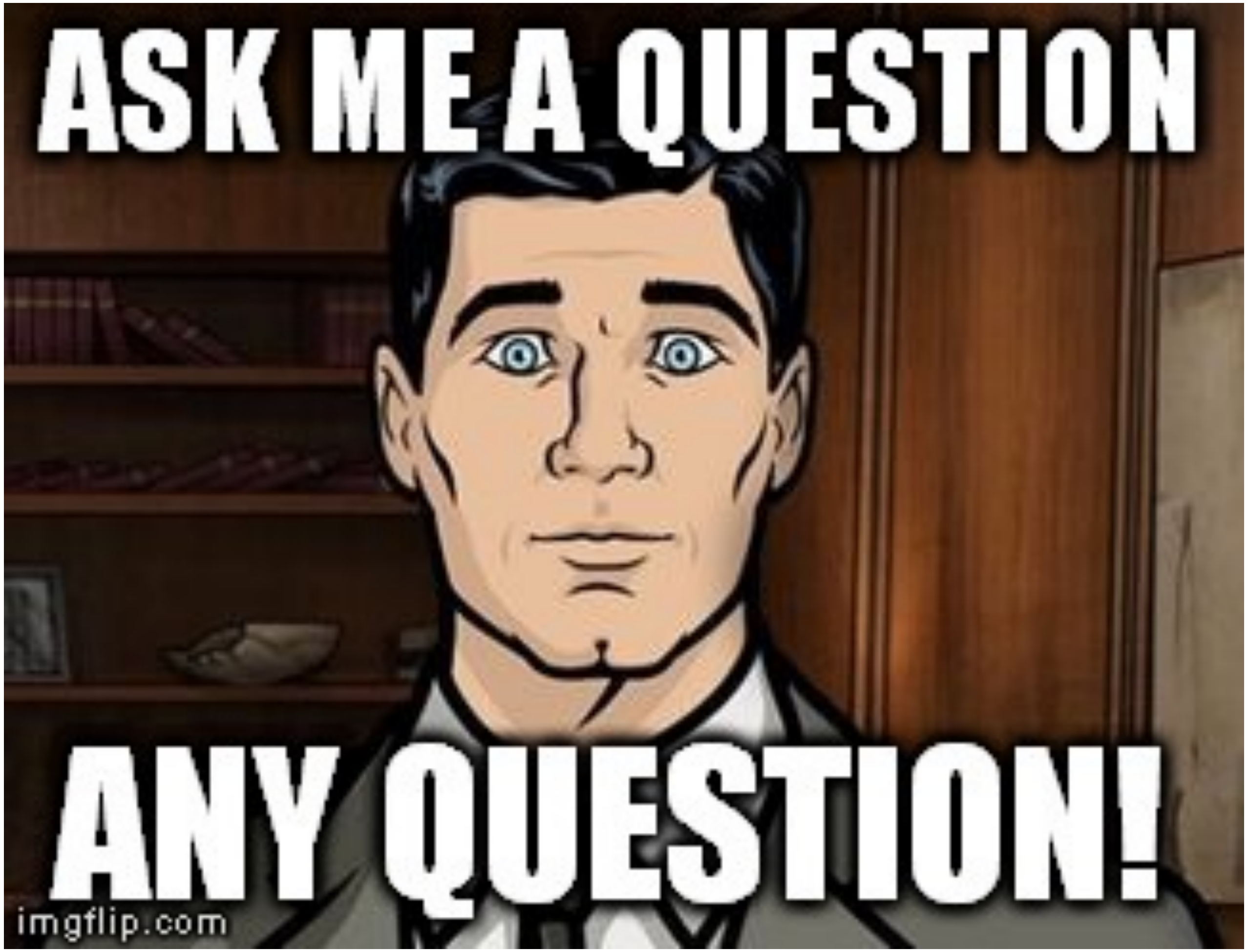
Upcoming Attractions and Activities



[Meme credit: cdn.meme.am]

Upcoming Attractions and Activities

- Quick review of formal grammars
- Lexical analysis – scanning & regular expressions
- Followed by parsing ...



[Meme credit: imgflip.com]