Northeastern University

CS 6410: Compilers

Fall 2023

Lecture 2

Tamara Bonaci

t.bonaci@northeastern.edu

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

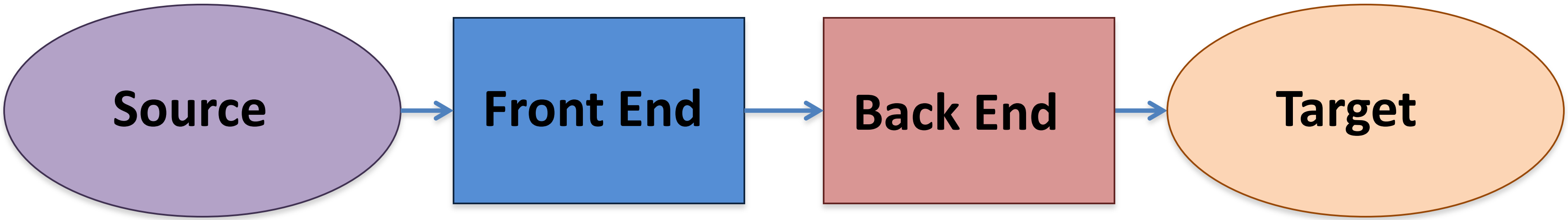
- Review – introduction to compilers
  - Front end
  - Back end
- Quick review of basic concepts of formal grammars
- Regular expressions
- Lexical specification of programming languages
- Using finite automata to recognize regular expressions
- Scanners and Tokens

CS 6410, Compilers – Fall 2023

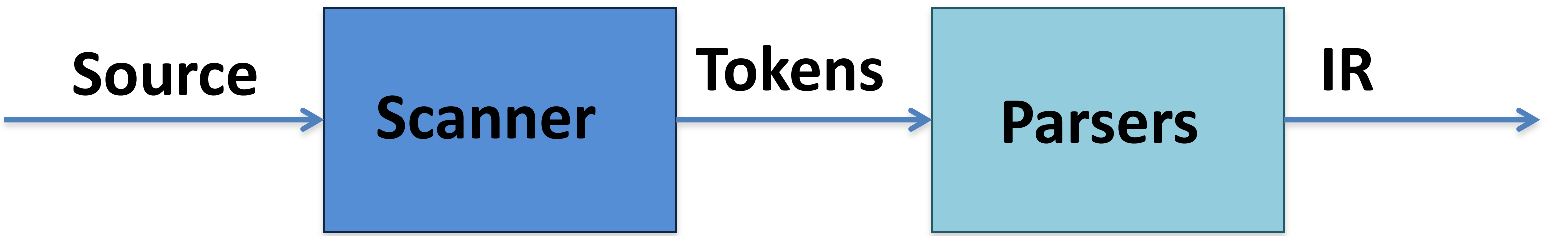
Introduction To Compilers - Review

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

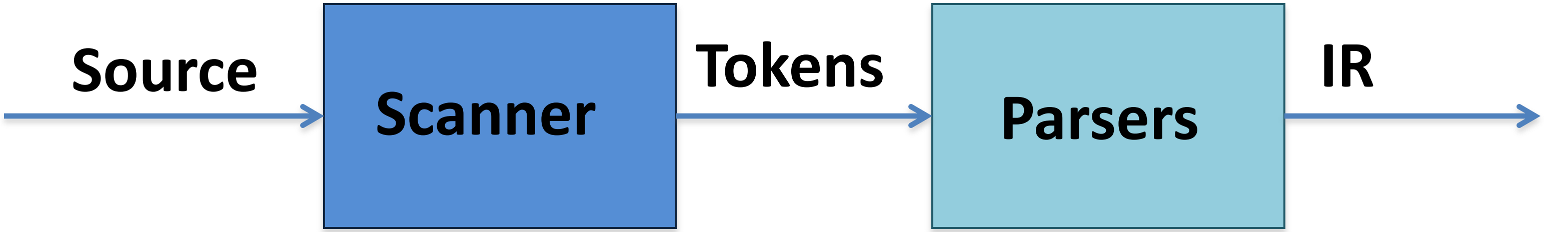


Compiler: Front End



- **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)

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

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

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

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

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

Programming Language Specs

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - First done in 1959 with BNF (Backus-Naur Form), used to specify ALGOL 60 syntax
  - Borrowed from the linguistics community (Chomsky)

CS 6410, Compilers – Fall 2023

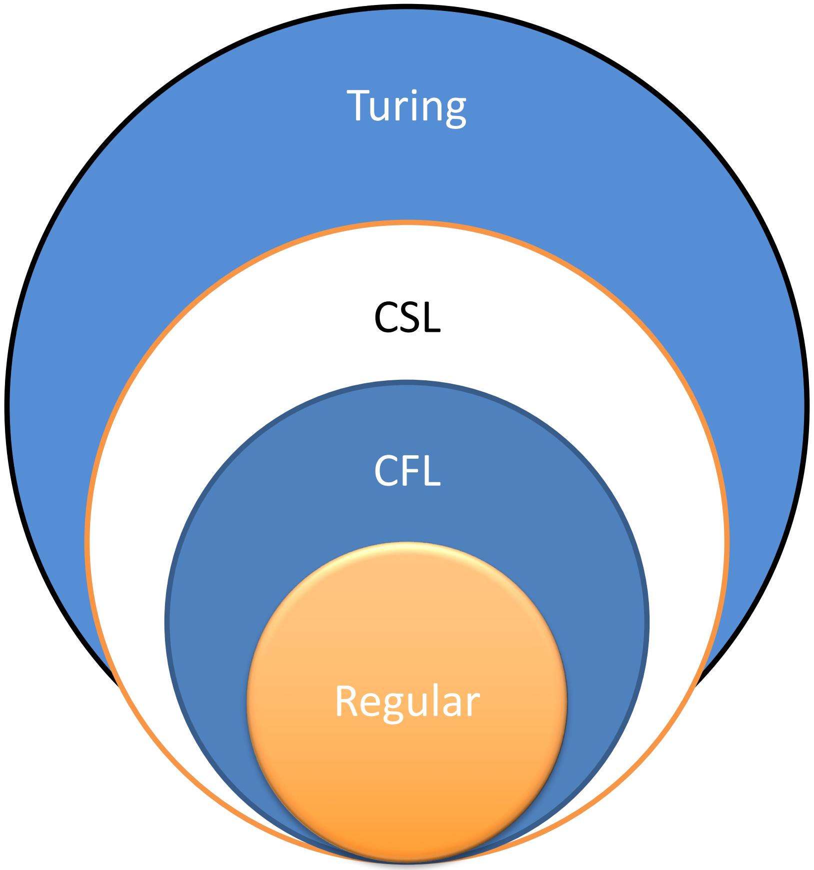
Formal Languages and Automata

Formal Languages & Automata Theory (One slide review)

- Alphabet: a finite set of symbols and characters
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set of strings (possibly empty or infinite)
- Finite specifications of (possibly infinite) languages
  - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  - Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Chomsky's Language Hierarchy

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
  - Specs and implementation of scanners
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
  - Specs and implementation of parsers
- Context-sensitive (Type-1) languages … aren't too important (at least for us)
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



Backus-Naur Form (BNF)

- Backus-Naur Form (BNF): a syntax for describing language grammars in terms of transformation *rules*, of the form:

  <symbol> ::= <expression> | <expression> ... | <expression>

  – Terminal: a fundamental symbol of the language
  – Non-terminal: a high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.
  – Developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

An Example BNF Grammar

```
<s>::=<n> <v>
<n>::=Tamara | Emily | Daniel |
<v>::=laughed | cooked | slept
```

- Some sentences that could be generated from this grammar:

```
Tamara cooked
Emily laughed
Daniel slept
```

Another Example BNF Grammar

```
<s>::=<np> <v>
<np>::=<pn> | <dp> <n>
<pn>::=Tamara | Emily | Daniel |
<dp>::=a | the
<n>::=ball | hamster | carrot | computer
<v>::=laughed | cooked | slept
```

- Some sentences that could be generated from this grammar:

```
the hamster cooked
Tamara slept
A computer laughed
```

Yet Another Example BNF Grammar

```
<s>::=<np> <v>
<np>::=<pn> | <dp> <adj> <n>
<pn>::=Tamara | Daniel | Emily | Jessica
<dp>::=a | the
<adj>::=silly | invisible | loud | romantic
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | belched
```

- Some sentences that could be generated from this grammar:

```
the invisible carrot cried
Jessica belched
a computer slept
a romantic ball belched
```

BNF Grammar and Recursion

```
<s>::=<np> <v>
<np>::=<pn> | <dp> <adjp> <n>
<pn>::=Tamara | Daniel | Emily | Jessica
<dp>::=a | the
<adjp>::=<adj> <adjp> | <adj>
<adj>::=silly | invisible | loud | romantic
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | belched
```

- Grammar rules can be defined *recursively*, so that the expansion of a symbol can contain that same symbol.
  - There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.

Example: Grammar for a Tiny Programming Language

program ::= *statement* | *program statement*

statement ::= *assignStmt* | *ifStmt*

assignStmt ::= *id* = *expr* ;

ifStmt ::= if (*expr*) *statement*

expr ::= *id* | *int* | *expr* + *expr*

id ::= a | b | c | i | j | k | n | x | y | z

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Exercise 1: Derive a Simple Program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Exercise 2: Derive Another Simple Program

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Productions

- The rules of a grammar are called productions
- Rules contain:
  - Non-terminal symbols: grammar variables (program, statement, id, etc.)
  - Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (,), …
- Meaning of

  nonterminal ::= <sequence of terminals and non-terminals>
    - In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and non-terminals on the right of the production
- Often there are several productions for a non-terminal – can choose any in different parts of derivation

Alternative Notations

- There are several notations for productions in common use; all mean the same thing

  ifStmt ::= if (*expr*) *statement*

  ifStmt → if (*expr*) *statement*

  <ifStmt> ::= if (<expr>) <statement>

CS 6410, Compilers – Fall 2023

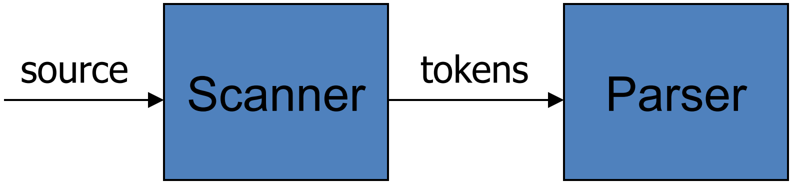
Parsing

Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from a concrete, character-by-character grammar
- In practice this is never done

Parsing & Scanning

- In real compilers, the recognizer is split into two phases
  - Scanner: translate input characters to tokens
    - Also, report lexical errors like illegal characters and illegal symbols
  - Parser: read token stream and reconstruct the derivation



Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
  - Scanner hides details from parser (comments, whitespace, input files, etc.)
  - Parser is easier to build; has simpler input stream (tokens) and simpler interface for input
- Efficiency
  - Scanner recognizes regular expressions – proper subset of context free grammars
    - (But still often consumes a surprising amount of the compiler's total execution time)

But …

- Not always possible to separate cleanly
- Example: C/C++/Java *type* vs *identifier*
  - Parser would like to know which names are types and which are identifiers, but…
  - Scanner doesn't know how things are declared
- So, we hack around it somehow…
  - Either use simpler grammar and disambiguate later, or communicate between scanner & parser
  - Engineering issue: try to keep interfaces as simple & clean as possible

Typical Tokens in Programming Languages

- Operators & Punctuation
  - + - * / ( ) { } [ ] ; : :: < <= == = != ! …
  - Each of these is a distinct lexical class
- Keywords
  - if while for goto return switch void …
  - Each of these is also a distinct lexical class (*not* a string)
- Identifiers
  - A single ID lexical class, but parameterized by actual id
- Integer constants
  - A single INT lexical class, but parameterized by int value
- Other constants, etc.

Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example

  return maybe != iffy;

  should be recognized as 5 tokens

RETURN	ID(maybe)	NEQ	ID(iffy)	SCOLON

  i.e., != is one token, not two; “iffy” is an ID, not IF followed by ID(fy)

Lexical Complications

- Most modern languages are free-form
  - Layout doesn't matter
  - Whitespace separates tokens
- Alternatives
  - Fortran – line oriented
  - Haskell, Python – indentation and layout can imply grouping
- And other confusions
  - In C++ or Java, is >> a shift operator or the end of two nested templates or generic classes?

CS 6410, Compilers – Fall 2023

Regular Expressions

Regular Expressions

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  - (Sometimes a little cheating is needed)
- Tokens can be recognized by a deterministic finite automaton
  - Can be either table-driven or built by hand based on lexical grammar

Regular Expressions

- Defined over some alphabet Σ
  - For programming languages, alphabet is usually ASCII or Unicode
- If *re* is a regular expression, *L*(*re*) is the language (set of strings) generated by *re*

Fundamental REs

re	*L*(*re*)	Notes
a	{ a }	Singleton set, for each a in Σ
ε	{ ε }	Empty string
∅	{ }	Empty language

Operations on REs

re	*L*(*re*)	Notes
rs	L(r)L(s)	Concatenation
r|s	L(r) ∪ L(s)	Combination (union)
r*	L(r)*	0 or more occurrences (Kleene closure)

- Precedence: * (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed

Examples

re	Meaning
+	single + character
!	single ! character
=	single = character
!=	2 character sequence "!="
xyzzy	5 character sequence ”xyzzy”
(1|0)*	0 or more binary digits
(1|0)(1|0)*	1 or more binary digits
0|1(0|1)*	sequence of binary digits with no leading 0’s, except for 0 itself

Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Some examples:

Abbr.	Meaning	Notes
r+	(rr*)	1 or more occurrences
r?	(r | ε)	0 or 1 occurrence
[a-z]	(a|b|…|z)	1 character in given range
[abxyz]	(a|b|x|y|z)	1 of the given characters

More Examples

re	Meaning
[abc]+	
[abc]*	
[0-9]+	
[1-9][0-9]*	
[a-zA-Z][a-zA-Z0-9_]*	

Abbreviations

- Many systems allow abbreviations to make writing and reading definitions or specifications easier

  name ::= *re*

  – Restriction: abbreviations may not be circular (recursive) either directly or indirectly (else would be non-regular)

Example

- Possible syntax for numeric constants

  digit ::= [0-9]

  digits ::= *digit*+

  number ::= *digits* (. *digits*)?
  ([eE] (+ | -)? *digits*) ?

- How would you describe this set in English?
- What are some examples of legal constants (strings) generated by *number* ?
  - What are the differences between these and numeric constants in YFPL? (Your Favorite Programming Language)

CS 6410, Compilers – Fall 2023

Recoginizing Res and Finite Automata

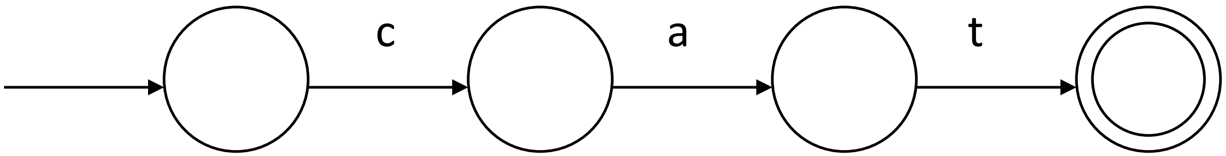
Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex, JFlex et seq do this automatically, given a set of REs

Finite State Automaton

- A finite set of states
  - One marked as initial state
  - One or more marked as accepting (final) states
  - States sometimes labeled or numbered
- A set of transitions from state to state
  - Each labeled with symbol from Σ, or ε
  - Common to allow multiple labels (symbols) on one edge to simplify diagrams
- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - ε-transition can be taken at any time
- Accept when final state reached & no more input
  - Slightly different in a scanner where the FSA is a subroutine that accepts the longest input string matching a token regular expression, starting at the current location in the input
- Reject if no transition possible, or no more input and not in final state (DFA)
  - Some versions require an explicit “error” state and transitions to it on all “no legal transition possible” input. OK to omit that for CSE 401

Example: FSA for “cat”

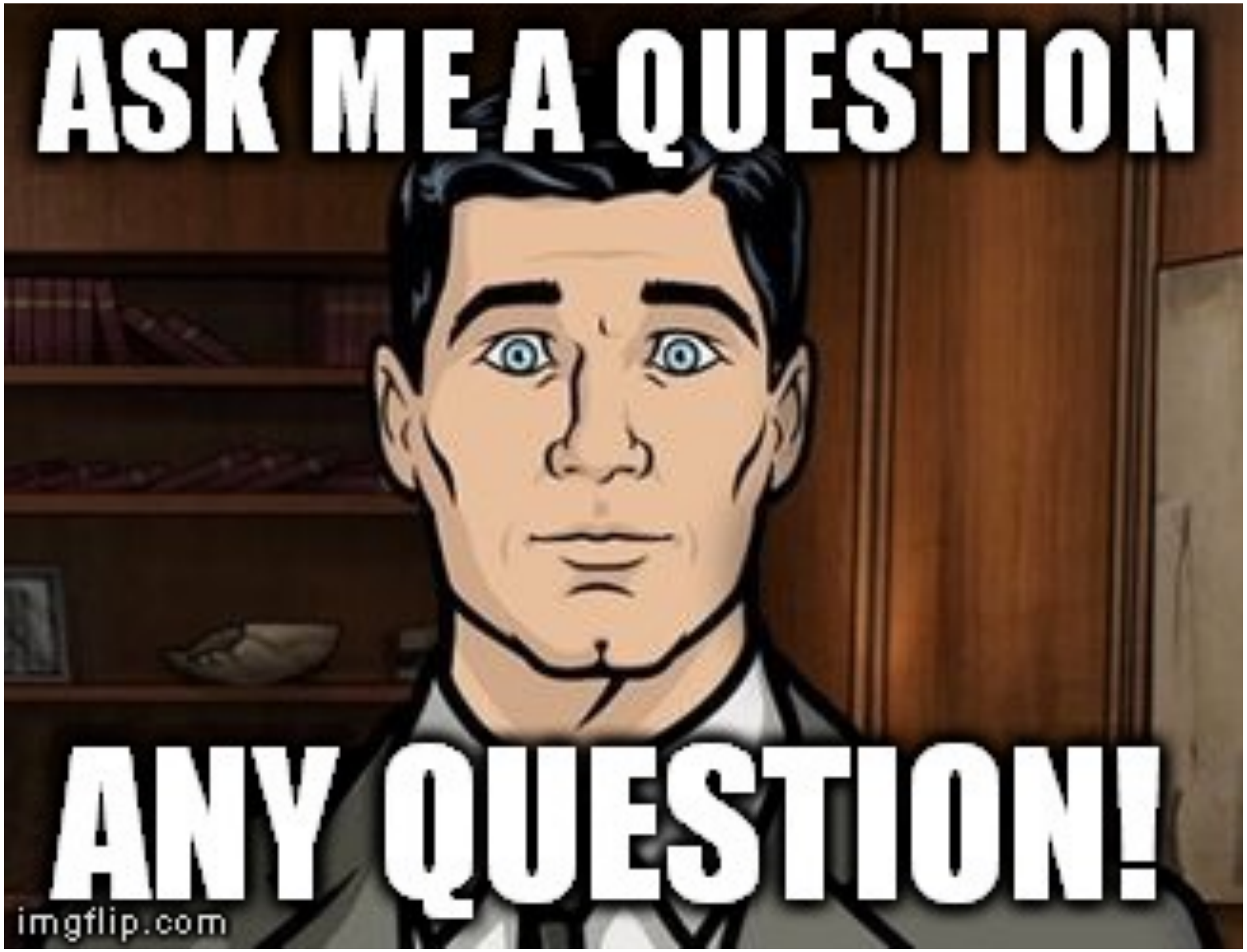


DFA vs NFA

- Deterministic Finite Automata (DFA)
  - No choice of which transition to take under any condition
  - No ε transitions (arcs)
- Non-deterministic Finite Automata (NFA)
  - Choice of transition in at least one case
  - Accept if some way to reach a final state on given input
  - Reject if no possible way to final state
  - i.e., may need to guess right path or backtrack

Coming Attractions

- First homework: paper exercises on regular expressions, automata, etc.
- Then: first part of the compiler assignment – the scanner
- Next topic: more scanning



[Meme credit: imgflip.com]