CSEP 590 – Programming Systems
University of Washington

Lecture 1: Motivation; Administratrivia;
Overview Part I

Michael Ringenburg
Spring 2017

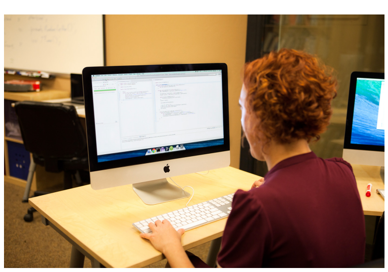
Agenda

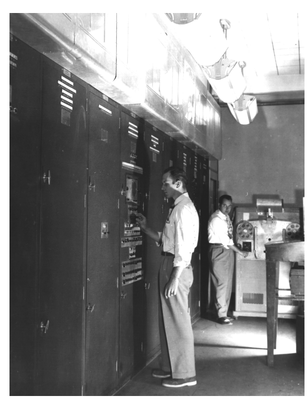
- What are programming systems?
  - Why do we study them?
- About this course
- High level overview of compilers and programming systems
- Fundamentals of Programming Systems, Part I

What are Programming Systems, and why do we care?

What are programming systems?

- Broadly all of the pieces of the software stack that enable a developer's source code to execute.





What are programming systems?

- Broadly all of the pieces of the software stack that enable a developer's source code to execute.
  - Can you think of any examples?

What are programming systems?

- My examples …
  - Compilers
    - High level language compilers, e.g., C++, Java, etc
    - Assemblers (translate assembly to machine code
  - Runtime Systems,
    - Stack and memory management
    - Libraries for interacting with the system (sockets libraries, graphics libraries, etc)
    - Garbage collection
    - Virtual machines (e.g., the JVM)
  - Interpreters
    - Python, ML, etc
  - Programming Frameworks
    - E.g., analytics frameworks like Hadoop/Spark
  - Verification tools
    - Debuggers
    - Profilers
    - Program Analysis tools

Why Study Programming Systems?

- Become a better programmer(!)
  - Insight into interaction between high-level language source and hardware
    - What "really" happens when you run your code
  - Understanding of implementation techniques, how code maps to hardware
  - Better intuition about what your code does
    - Write better, and faster, code
  - Understanding how compilers optimize code helps you write code that is easier to optimize
    - And not waste time making optimization that the compiler would do as well or better.

Why Study Programming Systems?

- Compiler techniques are everywhere
  - Parsing ("little" languages, interpreters, XML)
  - Software tools (verifiers, checkers, …)
  - Database engines, query languages
  - Text processing
    - Tex/LaTex -> dvi -> Postscript -> pdf
  - Hardware: VHDL; model-checking tools
  - Mathematics (Mathematica, Matlab)

Why Study Programming Systems?

- Fascinating blend of theory and engineering
  - Lots of beautiful theory around compilers
  - But also interesting engineering challenges and tradeoffs, particularly in optimization
    - Ordering of optimization phases
    - What's good for some programs may not be good 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/ heuristics

Why Study Programming Systems?

- Draws ideas from many parts of CSE
  - AI: Greedy algorithms, heuristic search
  - Algorithms: graph algorithms, dynamic programming, approximation algorithms
  - Theory: Grammars, DFAs and PDAs, pattern matching, fixed-point algorithms
  - Systems: Interaction with OS, runtimes
  - Architecture: pipelines, instruction set use, memory hierarchy management, locality

This Course

About me

- UW CSE PhD alum – graduated in 2014
  - Thesis research on architectures and programming models for approximate computing (reducing energy consumption by relaxing accuracy/precision guarantees)
  - Previously, research on programming language extensions for transactional memory, and runtime enforcement of security properties
- Worked at Cray since 2006
  - The supercomputer company
    - Building the world's fastest computers since 1972 ☺
  - ~7 years working on an automatically parallelizing compiler
    - Take non-parallel C/C++ code, plus (optional) pragmas, convert to a parallel program via automatic loop parallelization
  - More recently: working on data analytics and machine learning frameworks
    - High-productivity programming systems like Hadoop, Spark, Python Data Stack
    - How do we make them fast/take advantage of Cray hardware?

What am I doing here?

- Give something back to the department
- Enjoy teaching, meeting students
  - Taught undergraduate compilers course in 2013, PMP parallel computing in 2015
  - Both broadly about programming systems
- Programming systems is a broad, fascinating, ever-changing subject – always more to learn
  - Many of you probably have experiences and knowledge that I don't (even if you don't realize it!)
  - I hope to learn as much from you as you learn from me

Overall Course Goals

- Provide basic foundations of programming systems
  - Get everyone on the same page
  - First 2 or 3 weeks will be focused on this
- Explore a selection of topics relevant to modern languages and architectures
  - Topics you might not see in a traditional curriculum
  - I will provide some, but also want this to be partially driven by you…
- Gain experience with important skills for Masters grads:
  - Reading research literature in programming systems
    - Typical class format: short presentation(s) of papers about weekly topic, classroom discussion
    - Before class: Read paper(s), submit summary and discussion questions
- Presenting material, leading discussions
  - Everyone in class will present at some point this quarter ...

Presentations

- Everyone will be expected to prepare a presentation on a programming systems topic. Options:
  - Present a relevant project that you have worked on
  - Present a research paper in programming systems
  - Other ideas you suggest: E.g., implement something we discussed, present results?
  - Some ideas on course web (soon!), but feel free to suggest others
  - Proposal due to me by April 14 – more details to come soon
- We will allocate ~30 minutes each. 20 to present, 10 for questions and discussion.
  - Last 3 or 3.5 class sessions, depending on final enrollment
- Why?
  - Important skill for Masters graduates – career advancing
  - Allows class to share knowledge, learn about more topics than I could cover
  - Best way to learn is by teaching
  - Hopefully generate interesting discussions!

Class Sessions

- Don't worry, I won't lecture for three hours straight…
  - You would fall asleep; I would lose my voice
- Class will be a mix of lectures/presentations and discussion
- First 2 or 3 weeks will be more lecture heavy, as we cover the foundations
- Later classes will more discussion heavy
- Discussion basics
  - Discussion session is for **you** to discuss/debate (***politely***) the papers and related topics
  - Be considerate, polite, respectful of everyone – we all have different backgrounds
  - I am just here to moderate/keep things on track
  - So, please be prepared: do the readings and any homeworks on time
  - Otherwise discussions will not be valuable
- Today's discussion will be short, since the first reading isn't due until next week (maybe we can leave a little early!)s
  - Introduce yourselves, why you are here, what you work on, etc.
- Warning: I have some travel coming up middle of the quarter. Stay tuned…

Your Work

- **Assignments:**
  - Most weeks will include ~2 articles/research papers to read and review (sometimes 1, occasionally more if they are short)
  - May also include a couple short written and/or programming problems, especially in the beginning
- **Review format:**
  - 0.5 - 1 pages (using a "reasonable" font size)
  - Include:
    - Summary of articles key points
    - Do you agree/disagree? Why?
    - 2-3 discussion questions related to the article(s)
- **Late policy:** At most twice during the quarter, you may turn in an assignment late (max 1 week). This is intended for use with work/family emergencies – don't abuse.

Grading

- Don't worry, I'm not here because I want to fail anyone. ☺
- Everyone should be able to get a high grade if you show up, do the work, participate in the discussions as well as you can, and enjoy yourself.

Overview of Programming Systems

Types of Programming Systems

- Compilers
  - Responsible for translating human readable source into machine-executable instructions
- Runtime systems
  - Provides the common infrastructure needed to execute compiled programs
  - E.g., memory management, device access, threading, language features like garbage collection, etc
- Interpreters
  - Combine aspects to compilers and runtimes
  - Directly execute source code
- May also include tools like debuggers, profilers, static checkers, etc, used by developers to improve their programs
- We will focus the first couple lectures on compilation, but touch on other aspects as appropriate
  - Some of our later topics will touch on other types of programming systems more extensively

What do compilers do?

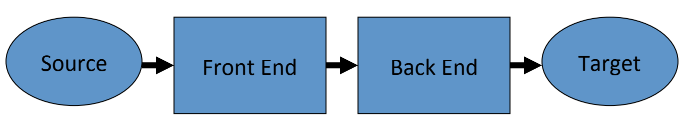
- How do we turn this into something the computer can execute?

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

- The computer only knows 1's & 0's
- Using a compiler (and/or an interpreter)
  - We'll discuss the differences in a few slides

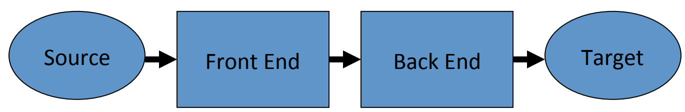
Structure of a Compiler

- At a high level, compilers have two pieces:
  - Front end: read source code
    - Parse the source, understand its structure
  - Back end: produce an executable
    - Generate equivalent target language program. May optimize (improve) code, but must not change behavior.



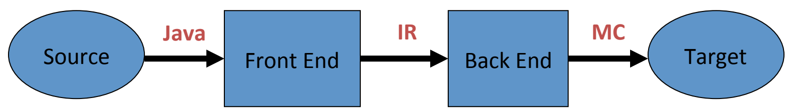
Compiler must…

- recognize legal programs (& complain about illegal ones)
- generate correct code
  - Programmer's favorite pastime is blaming their buggy code on "compiler bugs". ☺
- manage runtime storage of all variables/data
- agree with OS (loader) and linker on target format



How does this happen?

- Phases communicate via Intermediate Representations, a.k.a., "IR".
  - Front end maps source into IR
  - Back end maps IR to target machine code
  - Often multiple IRs produced by different phases of front/ back ends – higher level at first, lower level in later phases



Front End

source → Scanner → tokens → Parser → IR

- Usually split into two main parts
  - Scanner: Responsible for converting character stream to token stream: operation, variable, constant, etc.
    - Also: strips out white space, comments
  - Parser: Reads token stream; generates IR
    - (Semantics analysis can happen here, or immediately afterwards)
- Both of these can be generated automatically
  - Use a formal grammar to specify source language (e.g., Java)
  - Tools read the grammar and generate scanner & parser (e.g., lex and yacc for C, or JFlex and CUP for Java)

Scanner Output Example

- Input text

```
// Look, I wrote a comment!  I'm a good programmer!
if (x >= y) y = 42;
```

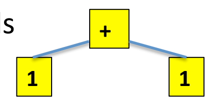
- Token Stream

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

  - Notes: tokens are atomic items, not character strings; comments & whitespace are *not* tokens (in most languages, ahem, FORTRAN)
    - Tokens *may* have associated data, e.g., a value or a variable name.

Parser Output (IR)

- Given token stream from scanner, parser must produce output that conveys meaning of program.
- Most common is an abstract syntax tree ("AST")
  - Essential meaning of program without syntactic noise
  - Nodes are operations, children are operands
    - E.g., 1 + 1 – Parent: +, Child1: 1, Child2: 1



- Many different forms of IR used in compilers
  - Engineering tradeoffs have changed over time
  - Tradeoffs (and IRs) also can vary between different phases of compilation.

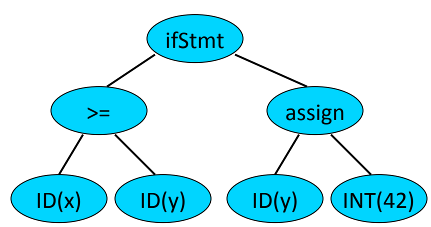
Parser Example

```
// Look, I wrote a comment!  I'm a good programmer!
if (x >= y) y = 42;
```

- Token Stream Input

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

- Abstract Syntax Tree



Static Semantic Analysis

- During and/or after parsing, checks that program is legal, and collects info for back end
  - Type checking
  - Check language requirements like proper declarations/initializations (e.g. Java locals), etc.
  - Collect other information used by back end analysis (e.g., scoping, aliasing restrictions)
- Key data structure: Symbol Table(s)
  - Maps names -> meaning/types/details

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 28

Back End

- Responsibilities
  - Translate IR into target machine code
  - Should produce "good" code
    - "good" = fast, compact, low power (pick some)
    - Optimization phases translate code into semantically equivalent but "better" code.
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 29

Back End Structure

- Typically split into two major parts
  - "Optimization" – code improvements, e.g.,
    - Common subexpression elimination:

      `(x+y) * (x+y)` → `t = x + y; t * t`

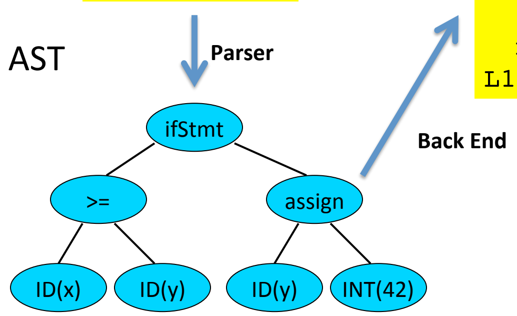
    - Constant folding: `(1+2) * x` → `3 * x`
    - Optimization phases often interleaved with analysis phases to better understand program meaning/know what transformations preserve that meaning
  - Target Code Generation (machine specific)
    - Instruction selection & scheduling, register allocation

The Result

- Input

```
if (x >= y)
  y = 42;
```

- AST



- Output

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

Interpreters & Compilers

- Programs can be compiled or interpreted (or in some cases both)
- Compiler
  - A program that translates a program from one language (the *source*) to another (the *target*)
  - In some cases the source and target can even be 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

w h i l e (k < l e n g t h) { <nl> <tab> i f (a [k] > 0
) <nl> <tab> <tab>{ n P o s + + ; } <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, other programming languages, (La)TeX, SQL (databases), VHDL (a hardware description language), many others
- Particularly appropriate if significant optimization wanted/needed

Interpreter

- Interpreter
  - Typically implemented with "execution engine" model
  - 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, functions)
    - But - hybrid approaches can avoid this …
  - But: immediate execution, good debugging/interaction, etc.

Often implemented with interpreters

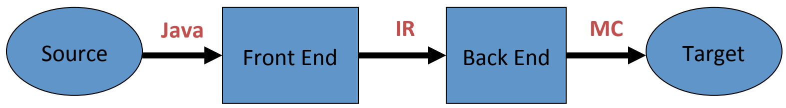
- Javascript, PERL, Python, Ruby, awk, sed, shells (bash), Scheme/Lisp/ML, postscript/pdf, machine simulators
- Particularly efficient if interpreter overhead is low relative to execution cost of individual statements
  - But even if not (machine simulators), flexibility, immediacy, or portability may 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 languages (Haskell, ML, Ruby), C# and Microsoft Common Language Runtime, others

Fundamentals of Compilers and Programming Systems

Front End

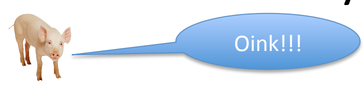


- We'll walk through the compilation process in order. Front end first:
  - Translate source code into compiler intermediate representation (IR)
  - Two parts
    - Scanning: read text, recognize *tokens*
    - Parsing: translate token stream into *Abstract Syntax Tree* (AST)
    - Produce IR (can take many forms)

Programming Language Specifications

- 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)

Review of Formal Languages and Automata Theory

- Starring Mr. Pig 

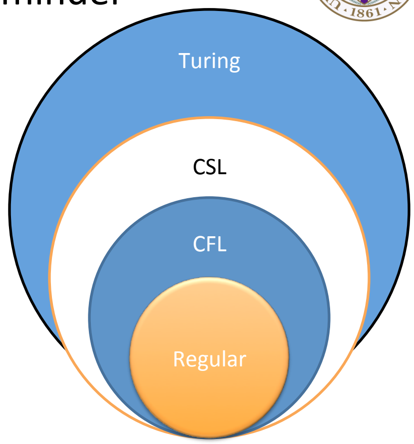
- Alphabet: a finite set of symbols and characters
  - E.g., {'i', 'k', 'n', 'o', ' '}
- String: a finite, possibly empty sequence of symbols from an alphabet
  - E.g., "oink"
- Language: a set of strings (possibly empty or infinite)
  - E.g., {"oink", "oink oink", "oink oink oink", …}

Review of Formal Languages and Automata Theory

- Finite specifications of (possibly infinite) languages:
  - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
    - E.g., a pig detector: accepts all sequences of oinks, rejects "moo"s or "baa"s (or anything else)
  - Grammar – a generator; a system for producing all strings in the language (and no other strings)
    - Unfortunately, we can't use a pig as our grammar – no pig (that I've met) can generate an infinite amount of "oink" sequences.
    - Instead we use formal (aka mathematical) grammars.
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Language (Chomsky) hierarchy: quick reminder

- Regular (Type-3) languages are specified by regular expressions/ grammars and finite automata (FAs)
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
- Context-sensitive (Type-1) languages … aren't too important
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



Example: Grammar for Pig-ish (or Pig-ese?)

- A formal grammar for our pig language could be:

PigTalk ::=	oink *PigTalk*	(rule 1)
|	oink	(rule 2)

- This can generate, for example:

PigTalk	::= oink	(Rule 2)
PigTalk	::= oink *PigTalk*	(Rule 1)
	::= oink oink	(Rule 2)
PigTalk	::= oink *PigTalk*	(Rule 1)
	::= oink oink *PigTalk*	(Rule 1)
	::= oink oink oink	(Rule 2)

Example: Grammar for a Tiny Language

- A more realistic (but still small) 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

Example: Derive a one line 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

program ::=
statement ::=
???

```
if (x) y = 1 + y ;
```

Example: Derive a one line 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

```
if (x) y = 1 + y ;
```

This is just one possible derivation. Many others are possible.

program ::=
statement ::=
ifStmt ::=
if (*expr*) *statement* ::=
if (*id*) *statement* ::=
if (x) *statement* ::=
if (x) *assignStmt* ::=
if (x) *id* = *expr* ; ::=
if (x) y = *expr* ; ::=
if (x) y = *expr* + *expr* ; ::=
if (x) y = *int* + *expr* ; ::=
if (x) y = 1 + *expr* ; ::=
if (x) y = 1 + *id* ; ::=
if (x) y = 1 + y ;

Example 2: A multiline 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

program ::=
???

```
if (x) y = 1 + y ; x = 1 ;
```

Your solution may reference your previous derivation.

Example 2: A multiline 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

```
if (x) y = 1 + y ; x = 1 ;
```

program **::=**
program statement **::=**
program assignStmt **::=**
program id = *expr* ; **::=**
program x = *expr* ; **::=**
program x = *int* ; **::=**
program x = 1 ; **::=**

Then derive *program* as in the previous example.

Once again, others are possible.

Alternative Notations

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

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

  ifStmt ⟶ if (*expr*) *statement*

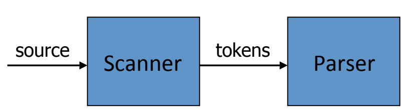
  <ifStmt> ::= if (<expr>) <statement>

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 source code to tokens
    - Reports *lexical* errors like illegal characters and illegal symbols.
  - Parser: read token stream and reconstruct the derivation
    - Reports *parsing* errors – i.e., source that is not derivable from the grammar. E.g., mismatched parens/braces, nonsensical statements (x = 1 +;)



*Not always quite this clean of a separation – but true at a high level.

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) / narrow interface
- Efficiency
  – Tokens can be defined by regular expressions, and recognized by finite automata.
    - (But still often consumes a surprising amount of the compiler's total execution time)
  – Parsing requires context-free grammars, and thus pushdown automata.
  – Can build automatic DFA generators for scanning (Jflex) and automatic PDA generators for parsing (CUP) .

File I/O!

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 54

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 …
- Things are even uglier in Fortran 77
  – E.g., `myvar`, `my var`, and `my  var` are all the same identifier, keywords are not reserved, etc. Tokenizing requires context…
- So we hack around it somehow…
  – Either use simpler grammar and disambiguate later, or communicate between scanner & parser (with some semantic analysis mixed in).
  – Real world: Often ends up very complex and hard to follow. Compiler front ends are sometimes referred to as "black magic".

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 55

Regular Expressions and Finite Automate (FAs)

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  - (Sometimes a little cheating is needed)
- Therefore, tokens can be recognized by a deterministic finite automaton
  - Can be either table-driven (automated tools like lex/flex) or built by hand based on lexical grammar

Fundamental REs

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

These are the basic building blocks that other regular expressions are built from.

Operations on REs

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

Precedence: * (highest), concatenation, | (lowest)

Parentheses can be used to group REs as needed

Examples

re	Meaning
+	single + character
!	single ! character
!=	2 character sequence
xyzzy	5 character sequence
(1|0)*	Zero or more binary digits
(1|0)(1|0)*	Binary constant
0|1(1|0)*	Binary constant without leading 0s

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

Examples

re	Meaning
[abc]+	Sequence of one or more a's, b's and c's
[abc]*	Zero or more a's, b's, and c's
[0-9]+	Integer (possibly with leading 0s)
[1-9][0-9]*	Integer (no leading 0s)
[a-zA-Z][a-zA-Z0-9_]*	One or more letters or digits, must start with a letter.

Example

- Possible syntax for numeric constants

digit ::= [0-9]
digits ::= *digit*+
number ::= *digits* (. *digits*)?
([eE] (+ | -)? *digits*) ?

- Notice that this allows (unnecessary) leading 0s, e.g., 00045.6. (0, or 0.14 would be necessary 0s.)
- How would you prevent that?

Example

- Possible syntax for numeric constants

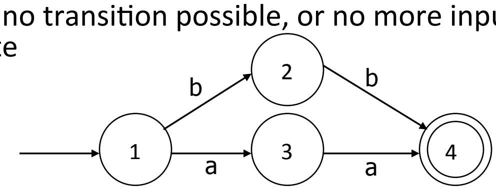
digit ::= [0-9]
nonzero_digit ::= [1-9]
digits ::= *digit*+
number ::= (0 | *nonzero_digit digits?)*
(. *digits*)?
([eE] (+ | -)? *digits*) ?

Recognizing REs

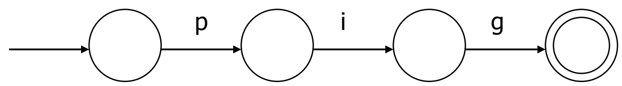
- Recall from your undergrad CS theory course … finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex (for compilers written in C++), and JFlex (for compilers written in Java) do this automatically, given a set of REs

Finite State Automaton

- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - Deterministic (DFA): Always one or zero possible transitions
  - Nondeterministic Finite Automata (NFA): May have multiple transitions. May also have ε-transitions that can be taken on any input.
  - Can convert to NFA -> DFA (recall your CS theory class).
- Accept when final state reached and no more input
  - Slightly different in a scanner, where the FSA is used as a subroutine to find the longest input string that matches a token RE.
- Reject if no transition possible, or no more input and not in final state

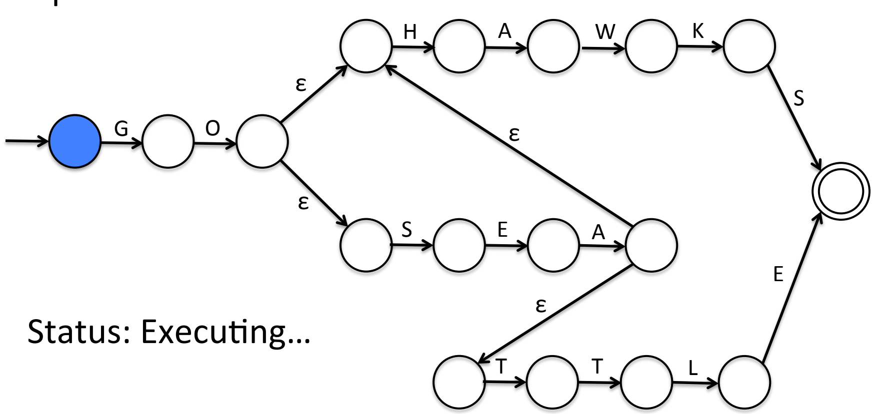


Example: DFA for "pig"



Example NFA: Seahawks Cheer token

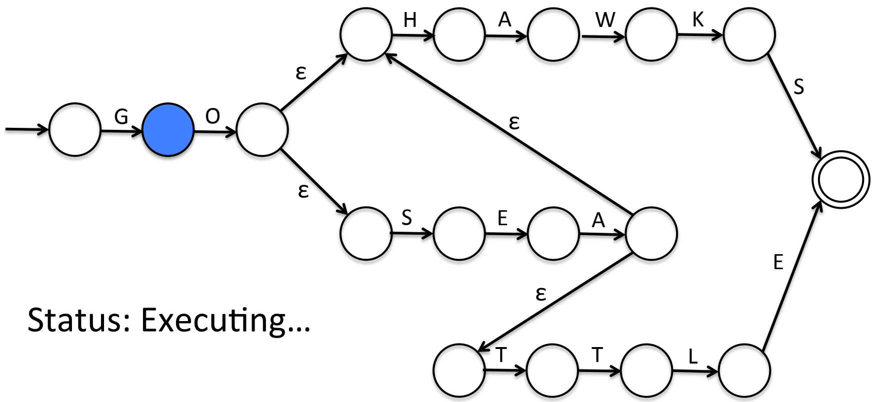
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

Input 1: GOSEAHAWKS

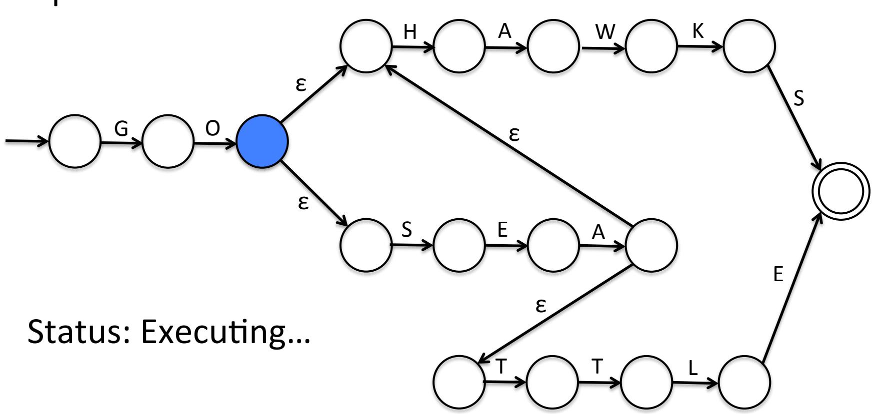


Status: Executing…

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 68

Example NFA: Seahawks Cheer token

Input 1: GOSEAHAWKS

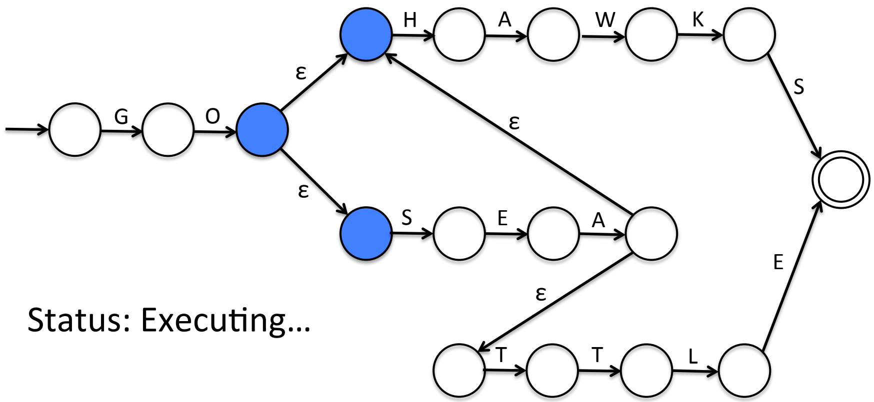


Status: Executing…

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 69

Example NFA: Seahawks Cheer token

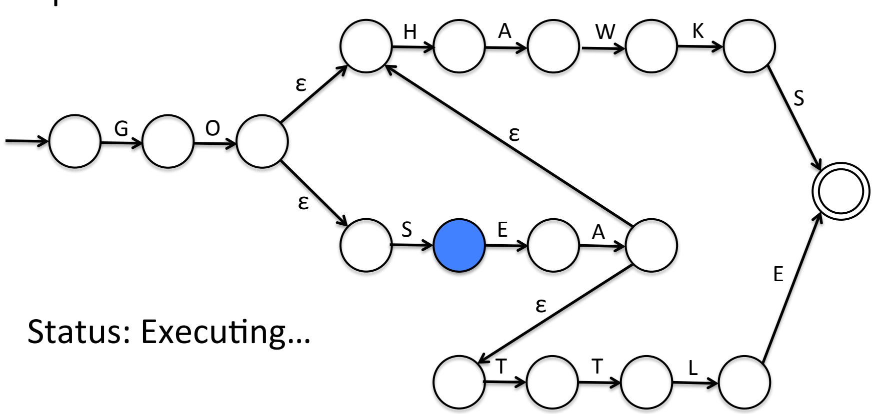
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

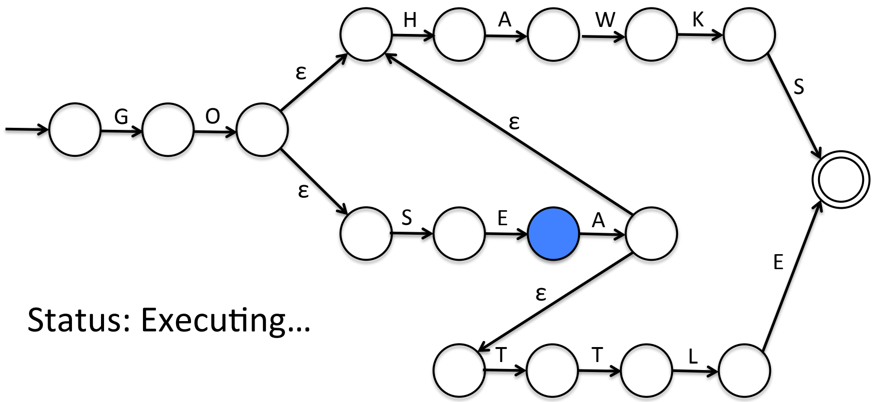
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

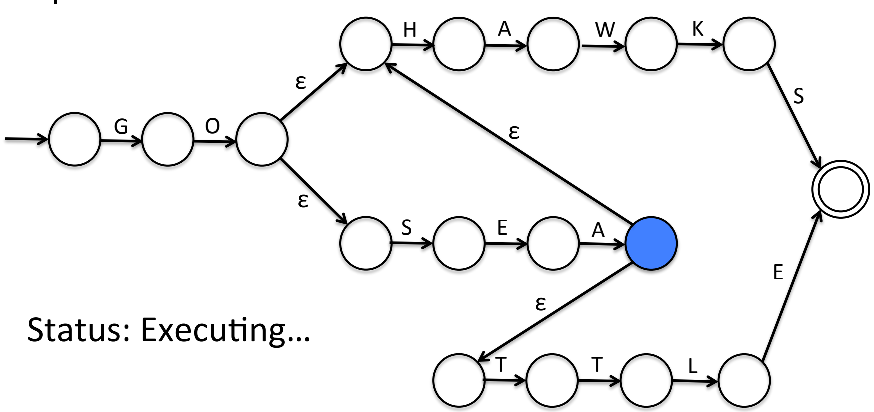
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

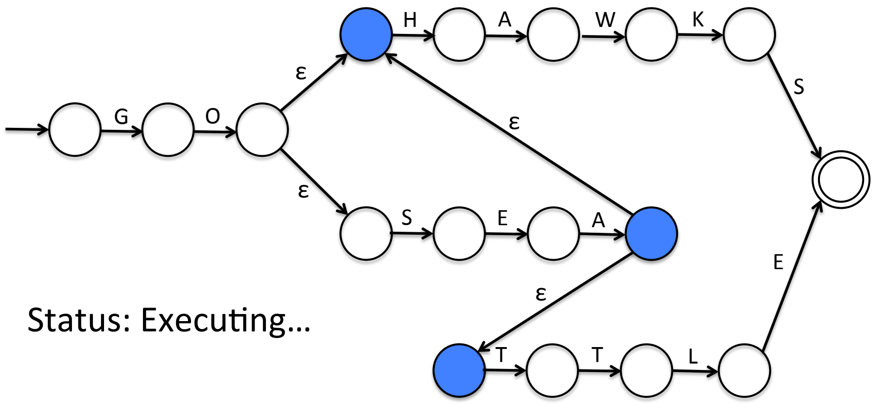
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

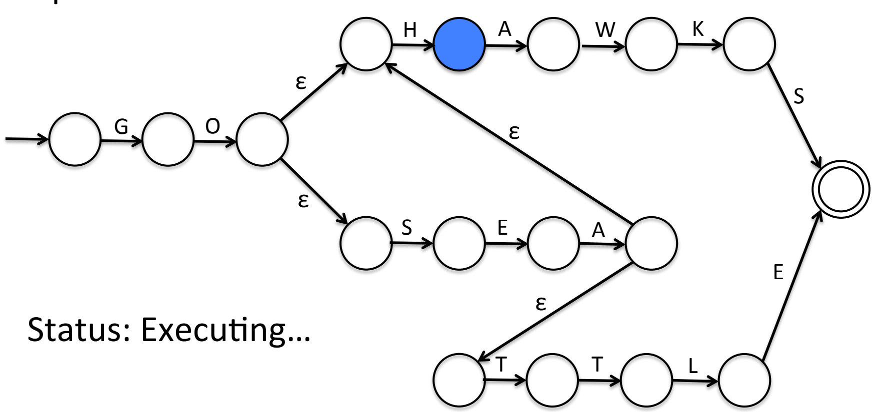
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

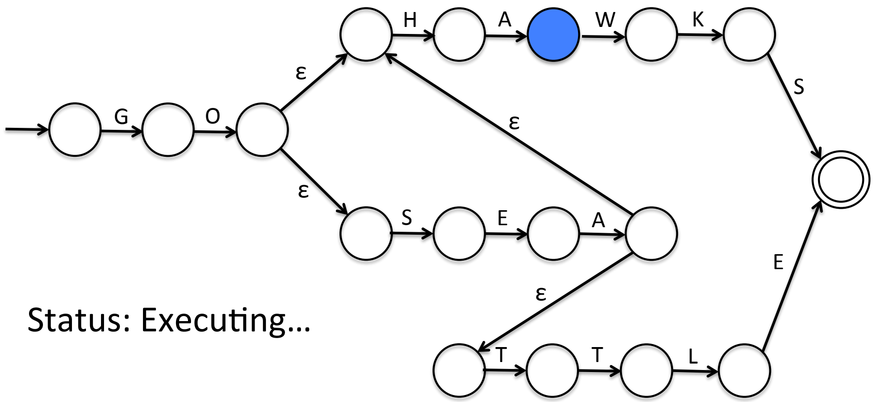
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

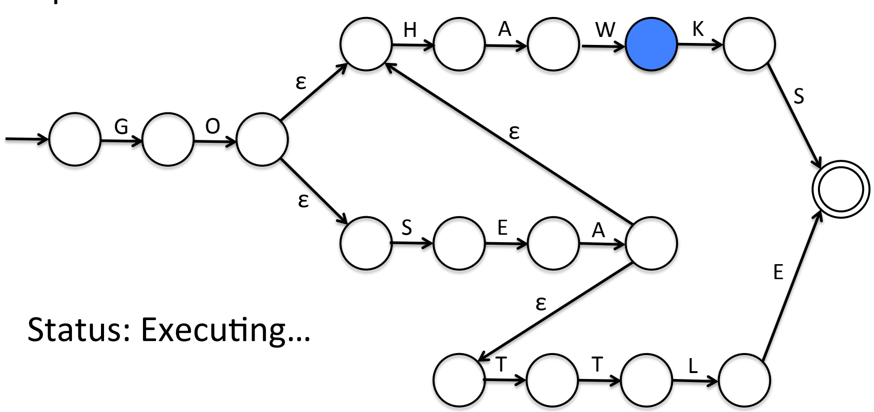
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

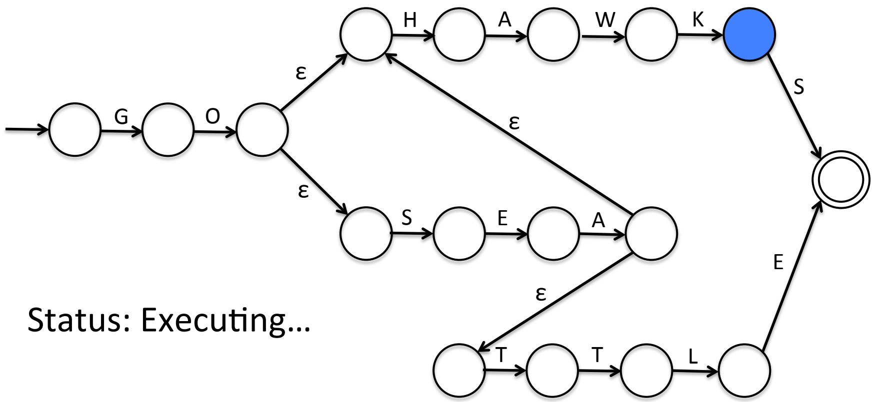
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

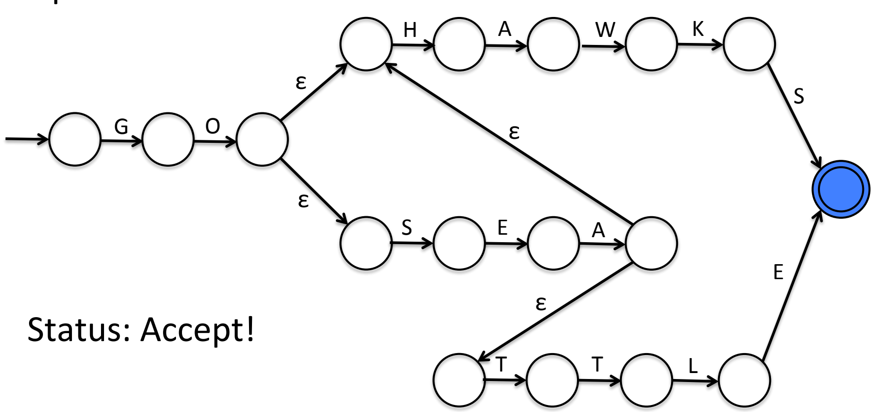
Input 1: GOSEAHAWKS



Status: Executing…

Example NFA: Seahawks Cheer token

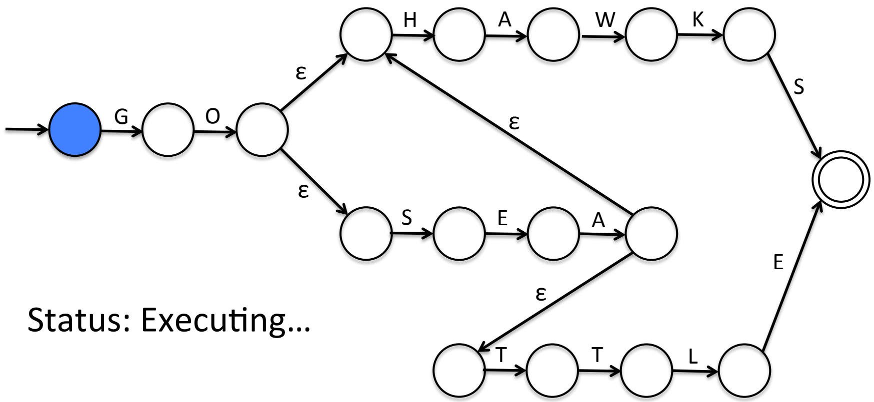
Input 1: GOSEAHAWKS



Status: Accept!

Example NFA: Seahawks Cheer token

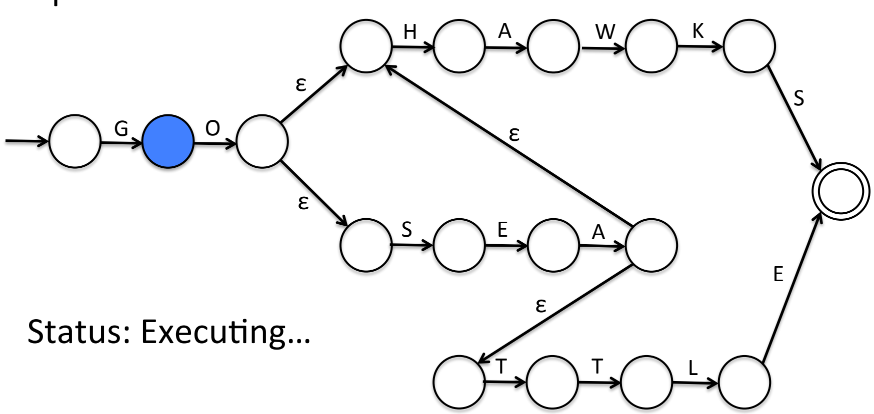
Input 2: GOPACKERS



Status: Executing…

Example NFA: Seahawks Cheer token

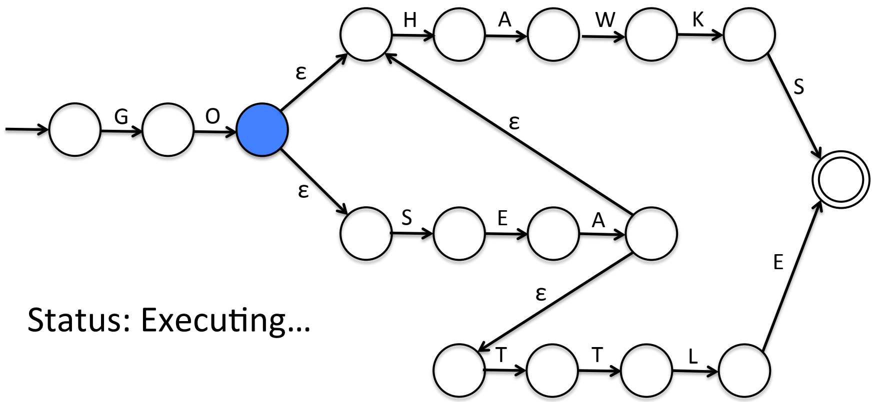
Input 2: GOPACKERS



Status: Executing…

Example NFA: Seahawks Cheer token

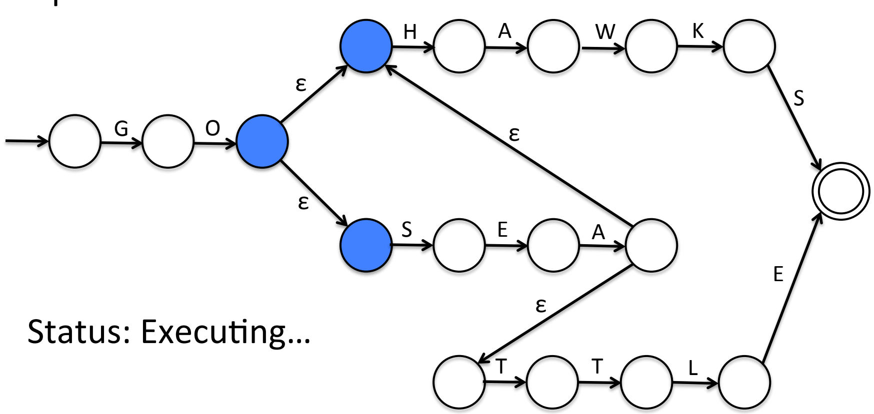
Input 2: GOPACKERS



Status: Executing…

Example NFA: Seahawks Cheer token

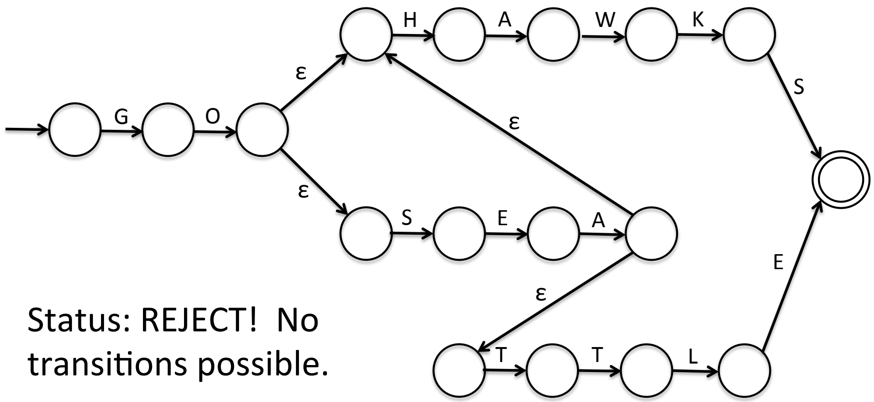
Input 2: GOPACKERS



Status: Executing…

Example NFA: Seahawks Cheer token

Input 2: GOPACKERS



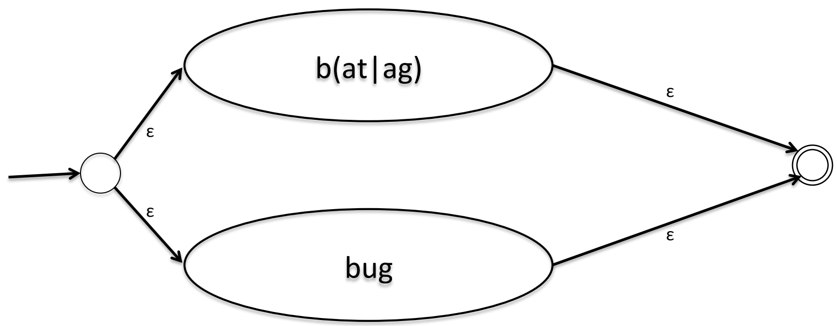
Status: REJECT! No transitions possible.

Example

- Draw the NFA for: b(at|ag) | bug

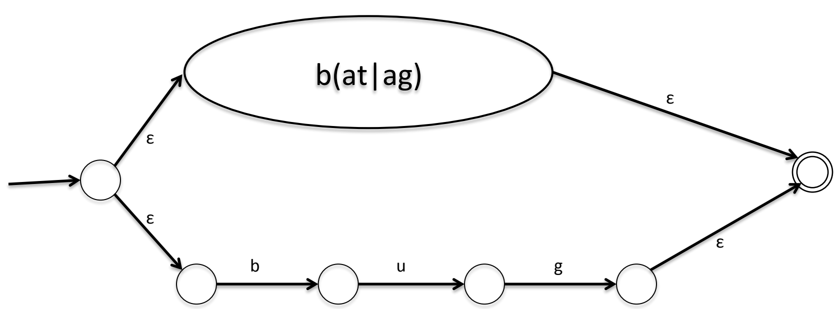
Example

- Draw the NFA for: b(at|ag) | bug



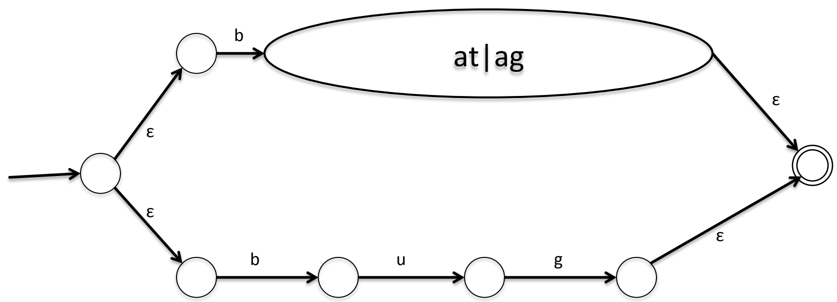
Example

- Draw the NFA for: b(at|ag) | bug



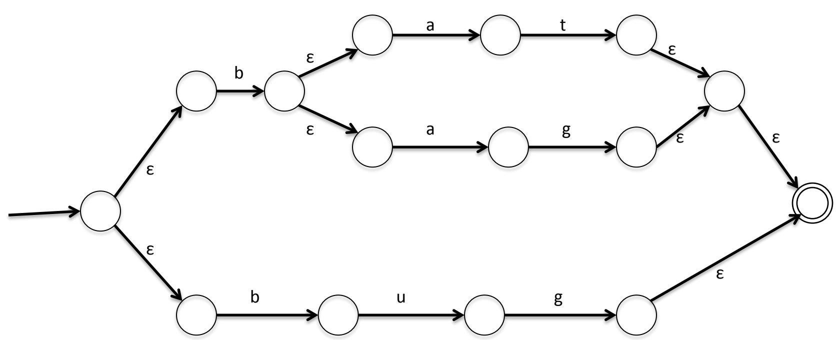
Example

- Draw the NFA for: b(at|ag) | bug



Example

- Draw the NFA for: b(at|ag) | bug



To Tokens

- A scanner is a DFA that finds the next token each time it is called
  - Slight modification: always try to find the longest token
- Every "final" state of a DFA emits (returns) a token
- Tokens are the internal compiler names for the lexemes

  == becomes equal

  (becomes leftParen

  private becomes private
- You choose the names

DFA => Code

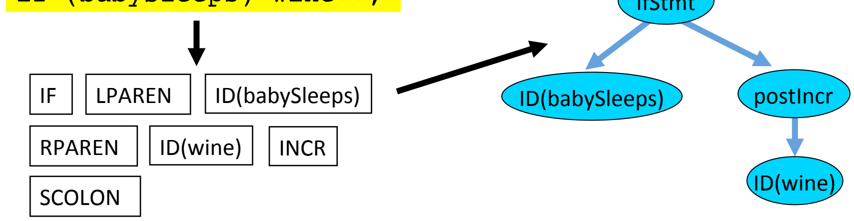
- Option 1: hand written
  - Pros
    - If written well, can be faster than auto-generated scanners
    - Handles weird language corner cases that don't map perfectly to the RE/ FA model
    - Readable code
  - Cons:
    - A lot of tedious work – thus, error prone
- Option 2: use a tool to generate a scanner
  - Pros
    - Convenient – just feed it the token regular expressions
    - Exactly matches specification you give it, if tool correct
  - Cons
    - Sometimes language constructs don't map perfectly to FA model
  - Table driven: Rows are states of DFA, columns are input characters, entries are action (go to next state, accept, error)
  - Direct-coded auto-generated scanner: transitions embedded in the code
    - Faster than table-driven, but generated code is very hard to follow

The Real World

- In commercial settings (and most gcc front ends) hand written scanners used more often than not.
  - Especially for larger languages, e.g., C++/Java.
  - Can purchase, e.g., EDG C/C++ front end.
- Auto-generated used for simpler languages, parsing "other things" (e.g., queries).
- Why hand written?
  - Fastest
  - Can handle language corner cases – C++ especially bad.
  - Readable/debugable code.

Parsing

```
if (babySleeps) wine++;
```

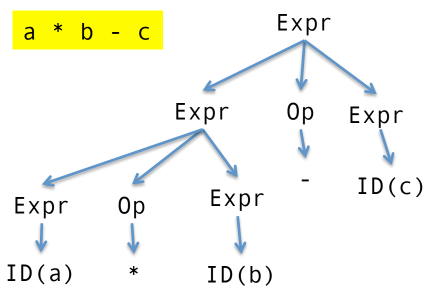


- We have: a scanner that generates a token stream
- We want an abstract syntax tree (AST)
  - A data structure that encodes the *meaning* of the program, and captures its structural features (loops, conditionals, etc.)
  - Primary data structure for next phases of compilation

How is this done?

- A grammar specifies the syntax of a language
- Parsing algorithms build *parse trees* based on a grammar and a stream of tokens
  - Parse trees represent how a string can be derived from a grammar, and *encode meaning*
    - E.g., multiply *a* by *b*, then subtract *c* from result.
  - Can build AST by traversing parse tree (parsers may do this implicitly).
- Do you see a problem here?

```
Expr  -> Expr Op Expr
       | ID
Op    -> -
       | *
```



Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CGF)
- Compromise between
  - REs: can't nest or specify recursive structure
  - General grammars: too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; but also allow restrictions for speed
- Not perfect
  - Cannot capture semantics, as in "variable must be declared" – requires later semantic pass
  - **Can be ambiguous**

What about ambiguity?

expr ::= *expr* + *expr* | *expr* – *expr*
| *expr* * *expr* | *expr* / *expr*
| INTEGER | ID | (*expr*)

- Need to construct unambiguous grammars for parsing
  - Otherwise nondeterminstic results of parsing and compilation!
- Classic example – order of operations
  - How do we ensure that * and / have higher precedence in our AST than + and - ???
  - Another common ambiguity: nested if-then-else

What about ambiguity?

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= INTEGER | ID | (*expr*)

- Need to construct unambiguous grammars for parsing
  - Otherwise nondeterminstic results of parsing and compilation!
- Classic example – order of operations
  - How do we ensure that * and / have higher precedence in our AST than + and - ???
  - Another common ambiguity: nested if-then-else

Examples

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= INTEGER | ID | (*expr*)

`a * b - c`

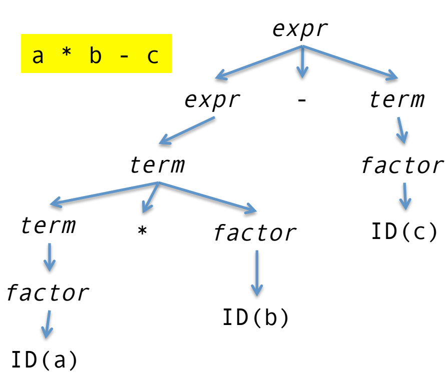
`a + b + c`

Examples

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= INTEGER | ID | (*expr*)

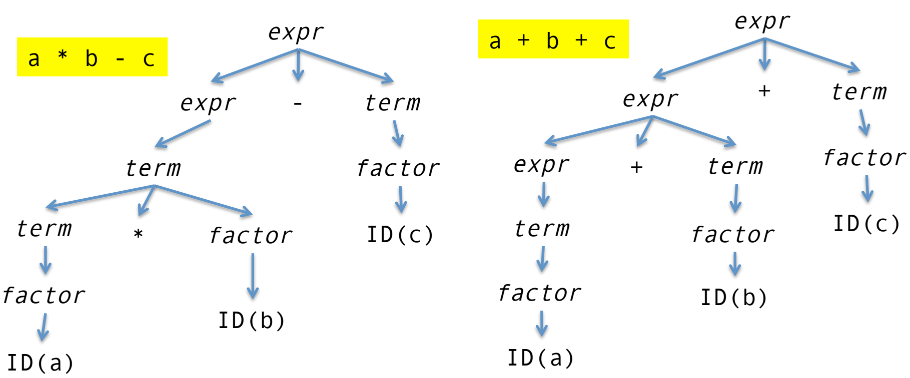
`a * b - c`

`a + b + c`



Examples

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= INTEGER | ID | (*expr*)



Shift-Reduce Parsing

- Most common parsing algorithms are shift-reduce bottom-up parsers
  - Bottom-up: Start with tokens, derive grammar starting symbol
  - Shift: Read tokens left to right, push them onto a stack
  - Reduce: Whenever the set of topmost tokens on the stack matches the right-hand side of a production, replace them with the appropriate non-terminal and add that non-terminal to the parse tree.

Shift-Reduce Example

Stack	Input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	Reduce A=>b
$aA	bcde$	shift
$aAb	cde$	shift
$aAbc	de$	reduce A=>Abc
$aA	de$	shift
$aAd	e$	reduce B=>d
$aAB	e$	shift
$aABe	$	reduce S=>aABe
$S	$	accept

S ::= aABe
A ::= Abc | b
B ::= d

Tables

- What if multiple choices possible (shift? reduce by rule 1? reduce by rule 2?)
  - Parsing algorithms generate a DFA based on the grammar that tells you what to do in each state
    - DFA + stack = PDA … which is how we recognize a CFG
    - DFA converted to table for efficiency
  - May use *lookahead* (peek at future symbols) to avoid backtracking
  - If table generation leads to conflict (shift-reduce or shift-shft), grammar is not parsable by that algorithm.

More Details

- Large amount of literature on parsing algorithms, but this is mostly a solved problem now
  - We will could spend the next few lectures going over this – but will instead refer the curious to any compiler textbook
  - And will have a short reading and homework problem to let you try it out
- Parser generators like yacc/bison (C) and CUP (Java) work well in many cases.
  - Specify grammar, actions to take to build AST
  - Will detect ambiguities, problems
  - Make it easy to specify precedence (so don't need to build more complicated grammars to encode)

Discussion

- Today will be short (we can go home early!), since you haven't read any papers yet.
- Briefly introduce yourself:
  - Name
  - Where you work
  - What you do
  - Why you are interested in this course
  - Any other interesting facts about yourself/ relevant background you bring/jokes/etc.