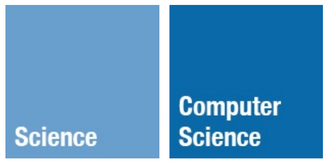
Formal Languages



CS 100: Introduction to the Profession

Matthew Bauer & Michael Saelee

ILLINOIS INSTITUTE OF TECHNOLOGY
College of Science

Some languages

- “Natural” languages: English, Chinese, Thai
- Programming languages: Java, Lisp, Lambda calculus
- Domain specific languages: SQL, HTML/CSS, UML
- Axiomatic systems: Propositional calculus, Set theory

Languages: what for?

- Socializing
- Artistic expression
- Communicating thoughts
- Representing problems
- Formalizing ideas

Who cares?

- Linguists: how to describe/categorize natural languages?
- Philosophers: what kinds of (valid) thoughts can we express?
- Mathematicians: how can we manipulate axiomatic systems?
- Computer scientists: how do we use languages to reason about, specify, and perform computational tasks?

Formally …

- A *language* consists of all *well-formed*, finite-length *strings* of *symbols* drawn from some *alphabet*.
  - "well-formed" according to some rules/constraints
  - strings ≈ words, sentences, formulae
  - symbols ≈ letters, tokens, terminals

e.g. language over { I, love }*

- Constraint: sentences begin with “I” and can’t be empty
- Valid sentences (infinite in number!):
  - I
  - I I I love
  - I love I love I love love love

Syntax vs. Semantics

- A formal language is strictly a *syntactic* specification
  - i.e., no ascription of semantics/meaning
- "Colorless green ideas sleep furiously" (Chomsky, 1957) is a well-formed but nonsensical English sentence
- Most applications of formal languages also require semantic interpretation to be useful (but not all!)

Applications in CS

- Data validation and recognition
- Parsing / Syntax-checking; e.g., vis-a-vis compiling
- Programming language specification
- Complexity theory; e.g., how much computational power is needed to recognize all strings of a given language?

Working with languages

- Formal grammars *generate* languages
- Automatons *accept* strings of a language
- Regular expressions *match* strings of a language
- Parsers *analyze/deconstruct* strings of a language

Formal Grammars

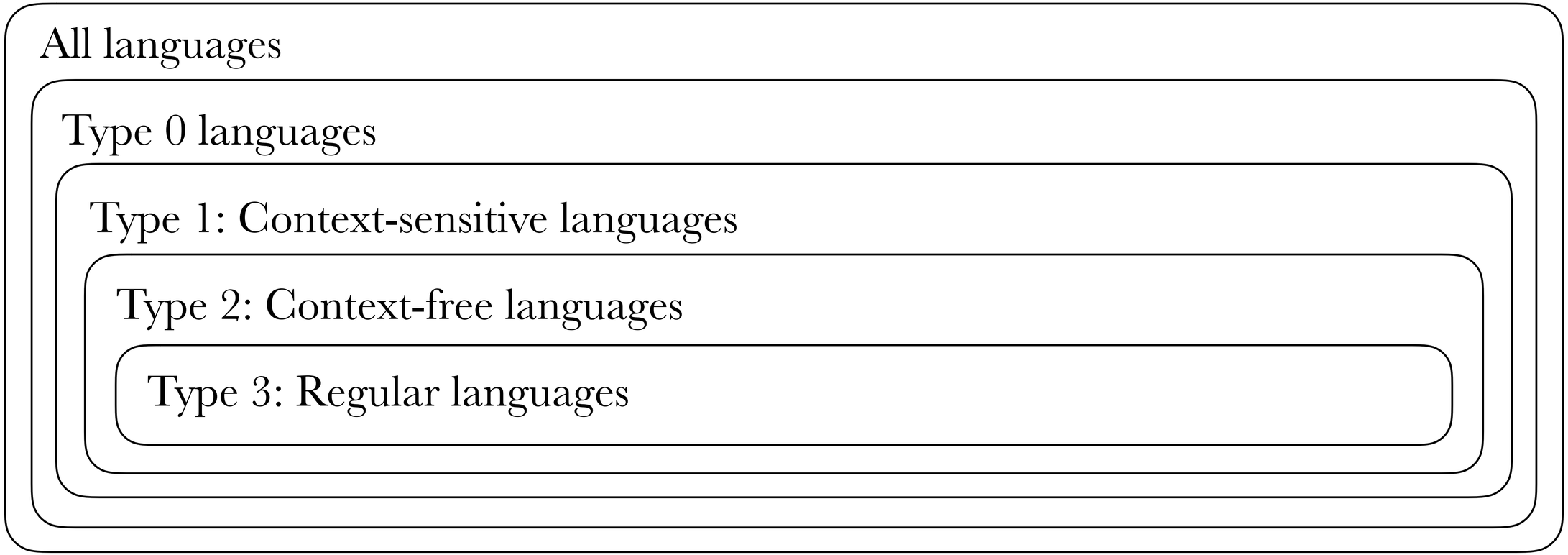
A formal grammar consists of:

1. a set of *terminal symbols* Σ; i.e., the alphabet
2. a set of *non-terminal symbols* N; aka variables
3. a set of *productions* P of the form *symbol(s)* $\rightarrow$ *symbol(s)*
   - left hand side must contain at least one non-terminal
4. a *start symbol* S

Chomsky Hierarchy

- Grammars are categorized by the Chomsky Hierarchy
  - *Type 0*: no extra constraints
  - *Type 1, aka "Context-Sensitive"*: # symbols on left hand side of each production must be ≤ # symbols on right hand side
  - *Type 2, aka "Context-Free"*: left hand side of each production can only have one symbol (a non-terminal)
  - *Type 3, aka "Regular"*: each production can only be of the form `A` → `a` or `A` → `aB`, where `A` and `B` are non-terminals, and `a` is a terminal

Chomsky Hierarchy



Grammars & Languages

- The language *generated* by a given grammar is the set of all strings we can *derive* from the start symbol
- Recall: grammars are just one way of specifying languages
  - Not all languages can be described by grammars!

e.g. CFG (Matched parentheses)

- $\Sigma = \{ (,) \}$; $N = \{ S \}$, $S = S$
- Productions:
  - $S \rightarrow SS$
  - $S \rightarrow (S)$
  - $S \rightarrow \varepsilon$

e.g. CFG (Matched parentheses)

- $\Sigma = \{ (,) \}$; $N = \{ S \}$, $S = S$
- Productions (using alternation):
  - $S \rightarrow SS \mid (S) \mid \varepsilon$
- e.g. deriving the string (( )( ))
  - $S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)(S)) \Rightarrow (( )( ))$

Derivation strategies

- If we have a string of multiple non-terminals during the derivation process, we have to decide which to expand first
- Two common strategies:
  - Leftmost derivation: expand the leftmost non-terminal
  - Rightmost derivation: expand the rightmost non-terminal

$S \rightarrow SS \mid (S) \mid \varepsilon$

- Using leftmost derivation, derive:
  - ()()()
  - (())()(())

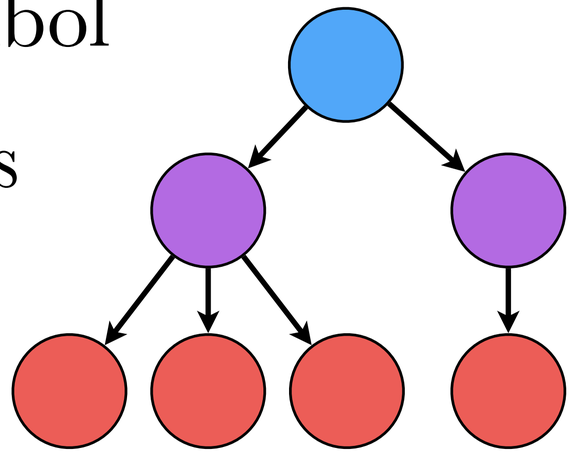
e.g. CFG (Simple arithmetic)

`Expr` → `Expr + Expr` | `Expr × Expr`
| `0` | `1` | `2` | ... | `9`

- Derivation for `5 + 2 × 3`?

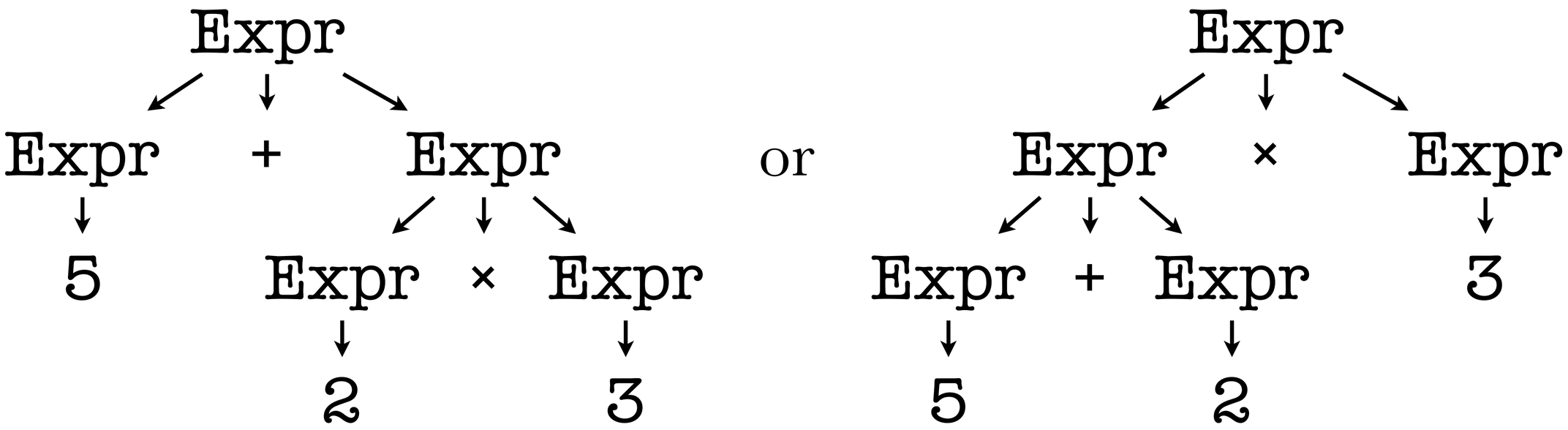
Parse trees

- Describe how a string is derived from some non-terminal
  - The root node represents the start symbol
  - Internal nodes represent non-terminals
  - Leaf nodes represent terminals



Expr → Expr + Expr | Expr × Expr | 0 | 1 | 2 | ... | 9

- Parse tree for 5 + 2 × 3?



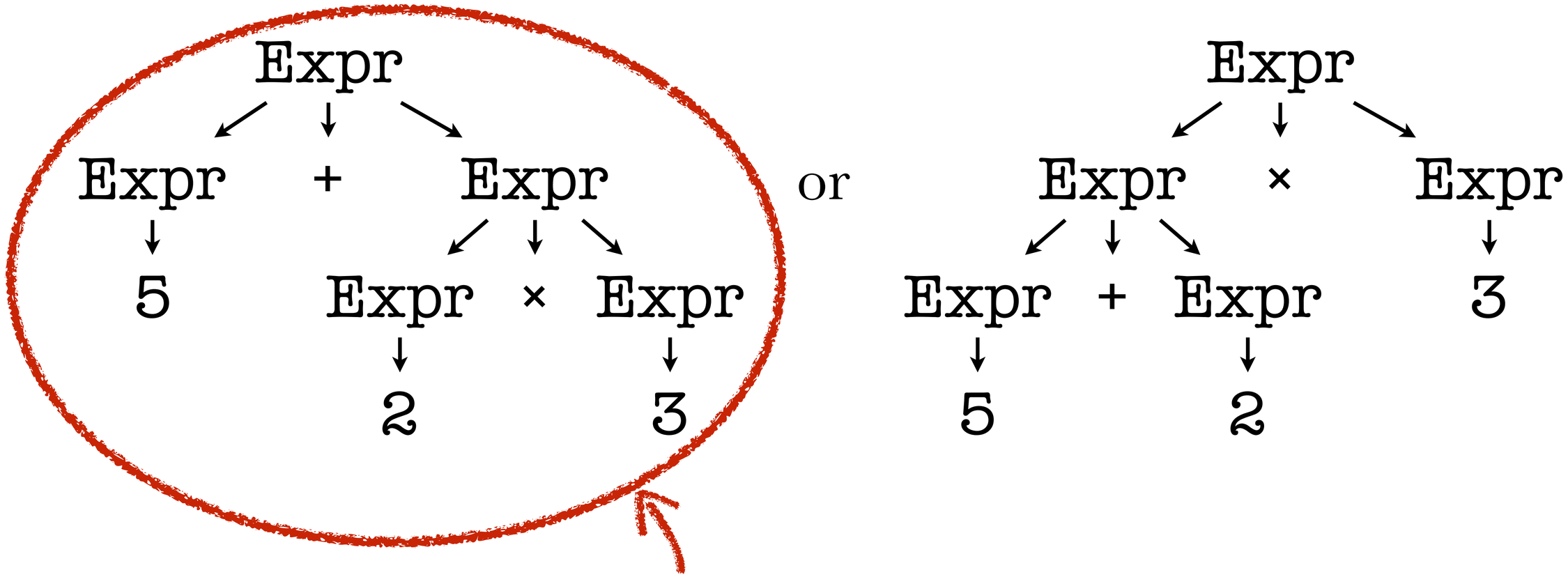
- This grammar is *ambiguous*; i.e., it may produce multiple parse trees for a given string

Ambiguous grammars

- May be problematic, especially if semantics are ascribed to substructures of the parse tree
  - E.g., arithmetic precedence, control structure nesting

$$\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} \times \text{Expr} \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

- Parse tree for $5 + 2 \times 3$?



this is the desired parse tree! (why?)

“Fixing” ambiguous grammars

- Rewrite grammar so it is no longer ambiguous but generates the same language (can be hard/impossible!)
  - May result in different parse trees
- Add disambiguating productions to force the desired parse trees to be generated

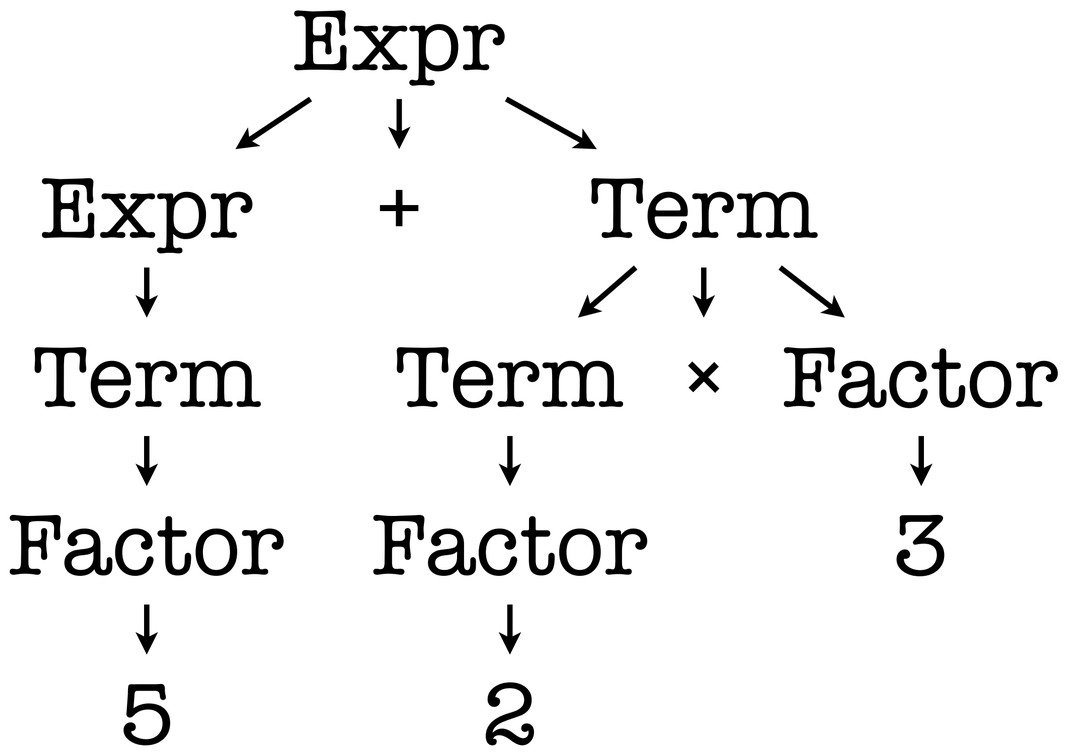
e.g. CFG (Simple arithmetic)

Expr → Term | Expr + Term

Term → Factor | Term × Factor

Factor → 0 | 1 | 2 | ... | 9

- Parse tree for 5 + 2 × 3?



e.g. CFG (Simple arithmetic)

We can update our grammar to allow for parentheses:

`Expr` → `Term` | `Expr` + `Term`

`Term` → `Factor` | `Term` × `Factor`

`Factor` → `0` | `1` | `2` | ... | `9` | (`Expr`)

Expr → Term | Expr + Term

Term → Factor | Term × Factor

Factor → 0 | 1 | 2 | ... | 9 | (Expr)

- Using leftmost derivation, show the parse trees for:
  - 1 + 2 + 3
  - 1 + 2 × 3 + 4
  - (1 + 2) × (3 + 4)

e.g. CFG (Java)

- http://cs.au.dk/~amoeller/RegAut/JavaBNF.html

Regular Grammars

- Recall, productions must take the form $A \rightarrow a$ or $A \rightarrow aB$, where A and B are non-terminals, and a is a terminal
  - Technically, this describes a *right-regular* grammar; left-regular grammars also exist (what would they look like?)

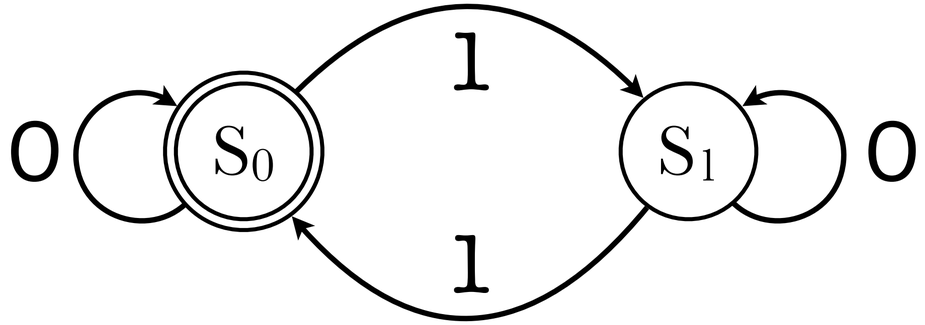
e.g. Regular Grammar

- `A` → `0A` | `1B` | ε
- `B` → `0B` | `1A`
- Derive some strings based on this grammar. What characteristic do they share?
  - All strings have an even number of `1`s; aka *even parity*

Limitation & Simplicity

- Because regular expressions only expand to the right (or left), they cannot generate languages with nested/recursive substructures (e.g., matching parentheses)
  - Due to this simplicity, recognizing regular languages requires limited computing power and memory
- *Finite-state machines* can be used to recognize regular languages!

e.g. FSM acceptor (even parity)



- Candidate strings are scanned left to right; each token follows the appropriate state transition (start from state S_0)
- FSM fails to accept a string if a valid state transition is not available or it fails to terminate on a final (circled) state

Ubiquity of Regular languages

- Despite (due to?) their relative simplicity, regular languages are incredibly important and commonplace
  - Vast majority of simple data formats are regular languages
    - e.g., URLs, e-mail addresses, dates, numerical data, etc.
    - Even when not, useful subsets of data often are

Regular Expressions

- Regular expressions are another way of describing how to match strings corresponding to regular languages
  - Can also be used to extract data from and manipulate strings being matched

Some Regexp Elements

- Most characters match themselves (aka literals)
- Metacharacters may match a set of characters (e.g., '.' matches any character, '`\d`' matches a digit)
- Quantifiers indicate how many of the preceding character to match (e.g., '`*`' = 0 or more, '`+`' = 1 or more, '`?`' = 0 or 1)
- `|` for alternation, `()` for grouping, `[]` for character classes

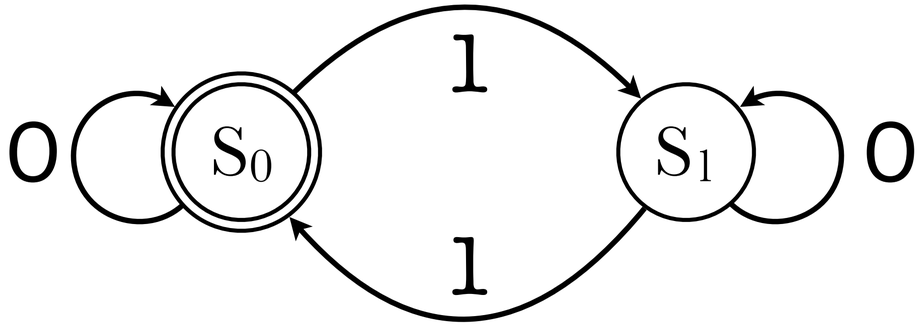
e.g. Regexps

- `mic.*` matches mic, michael, mic_9c, …
- `m+ike` matches mike, mmike, mmmike, …
- `r(at)+` matches rat, ratatatatat
- `(m|n)+emonic` matches mnemonic, mnmnnmnemonic, ...
- `CS.?\d{3}` matches CS_100, CS200, CS 351, …

Regexp = FSM = Reg. Grammar

- All can be used interchangeably to specify a regular language!
  - Regexps are just algebraic notation for regular grammars
  - FSMs can be designed to accept precisely the language generated by a regular grammar

e.g. Even parity Regexp?



Demo

- https://regexr.com