LING/C SC/PSYC 438/538

Lecture 23

Sandiway Fong

Adminstration

538 Presentations

- **Reminder**: send me your top 3 choices
  - give section numbers and title of section!
  - preferred date of your presentation: Dec 5th or 7th

- No Homework today:
  - **there'll be one next time, so make sure you understand the Prolog**
  - *haven't graded the most recent homework yet* …

538 Presentations

Name	Presentation	Date
	Sections	5 or 7
Alkuraydis,Ahmed		
Barner,Jacob Ryan		
Bejarano,Cielo S		
Bell,Jack T		
Converse,Amber Charlotte	8.4 HMM Part-of-Speech Tagging	7
Cox,Samantha Ann	18 Word Senses and WordNet, 18.1-18.3	7
Davis,Katherine Nicole		
Dharmala,Bayu	12.3 Some Grammar Rules for English	7
Hopper,Ashlyn Danielle		
Jain,Varshit Chirag	23 Question Answering, 23.4-23.6	
Kankia,Kevin Pinakin		
Kleczewski,Alison	15.4 Event and State Representations, 15.4, 15.4.1, 15.4.2	7
LaScola Ochoa,Logan Michelle		
Logan,Haley Brooke		
Maibach,Marcus Wile		
Mangkang,Tinnawit		
Mangla,Sourav	3 N-gram Language Models, 3.1,3.4	5
McLaughlin,Matthew		
Mehta,Deep Paresh	17 Extracting Times and Events, 17.3-17.4	5
Mendoza,Freddy		
Murphy III,Michael LaMotte		
Pinto,Aayush Bernard	20 Lexicons for Sentiment, Affect, and Connotation, 20.1-20.3	
Pipatanangkura,Leighanna D	23.2 IR-based Factoid Question Answering	
Raju,Anish		
Reeve,Keegan Austin		
Ruparel,Deep Anil	6 : Vector Semantics and Embeddings, 6.3-6.5	5
Shakyam Shreya Nupur	24.2 Chatbots	7
Shu,Qiyu		
Shukla,Kartikey	23.1 Information Retrieval	5

Last Time

- SWI-Prolog introduced: a logic-based programming language
- **Key Concepts so far**:
  - facts: *what is true* – example: bird
  - rules: *logical inference* – example: canfly if bird
  - recursive rules – examples: factorial and Σ*
  - infinite loop (*recursion*) – example: factorial definition without n > 0 guard
  - enumeration (*language*) – example: Σ*
  - backtracking: *explore multiple possible paths of execution*
  - control of backtracking using `fail` (initiate backtracking) and `!` (*cut*: i.e. stop)

Regular Languages

- Three formalisms, same expressive power
  1. Regular expressions
  2. Finite State Automata
  3. **Regular Grammars**

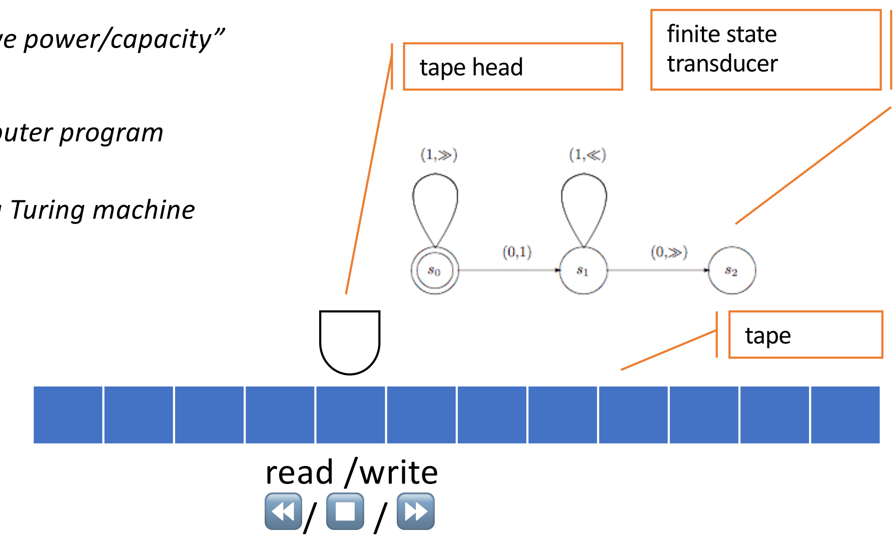
We'll look at this case
using Prolog

Chomsky Hierarchy

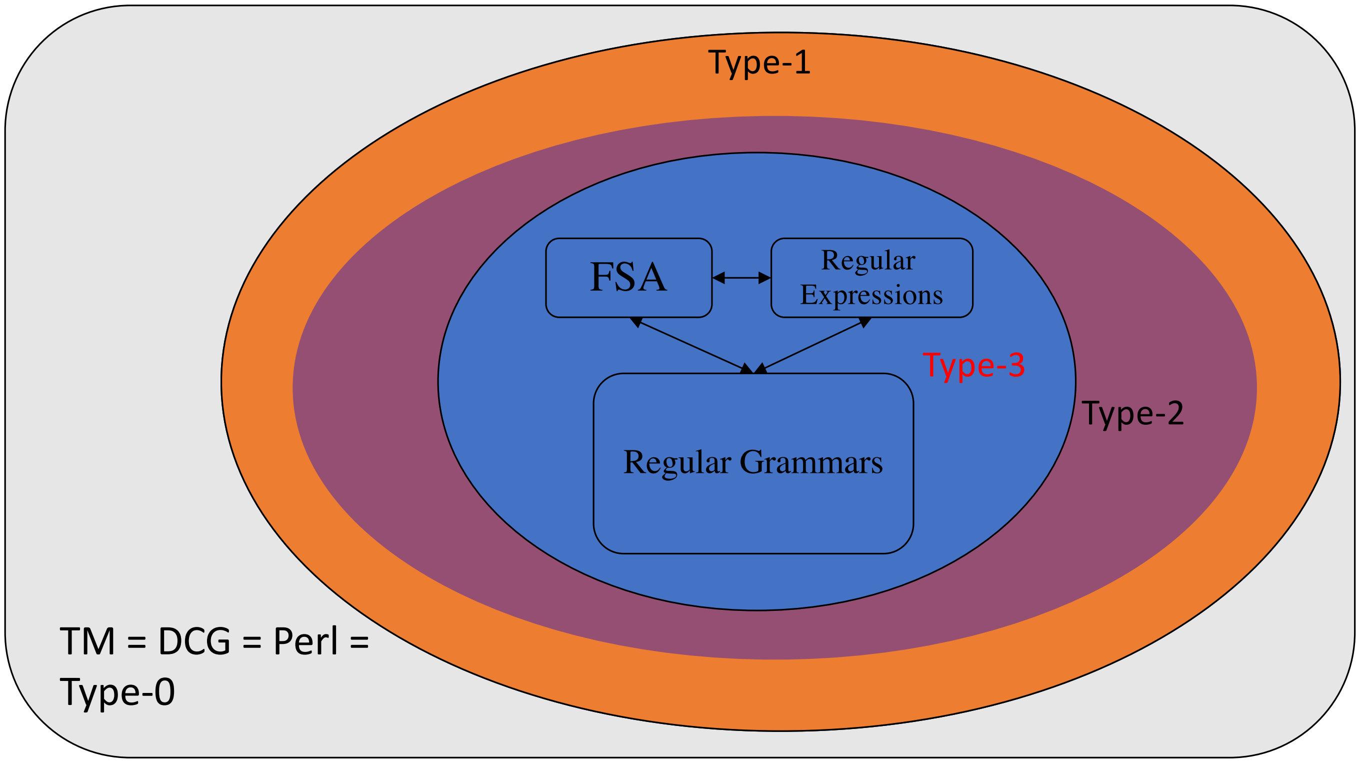
Natural languages: do they even fit here?

Chomsky Hierarchy

- *division of grammar into subclasses partitioned by “generative power/capacity”*
- **Type-0 General rewrite rules**
  - *Turing-complete, powerful enough to encode any computer program*
  - *can simulate a Turing machine*
  - *anything that’s “computable” can be simulated using a Turing machine*
  - **Type-1 Context-sensitive rules**
  - *weaker, but still very powerful*
  - $a^n b^n c^n$
    - **Type-2 Context-free rules**
    - *weaker still*
    - $a^n b^n$ *Pushdown Automata* (PDA)
      - **Type-3 Regular grammar rules**
      - *very restricted*
      - *Regular Expressions* a^+b^+
      - *Finite State Automata* (FSA)



Chomsky Hierarchy



Prolog Grammar Rule System

- known as “Definite Clause Grammars” (DCG)
  - based on type-2 restrictions (context-free grammars)
  - but with extensions
  - (powerful enough to encode the hierarchy all the way up to type-0)
  - Prolog was originally designed (1970s) to also support natural language processing

  - we’ll start with the bottom of the hierarchy
    - *i.e. the least powerful*
    - regular grammars (type-3)

Definite Clause Grammars (DCG)

- **Background**
  - a “typical” formal grammar contains 4 things
  - <N,T,P,S>
    - **a set of non-terminal symbols (N)**
      - *these symbols will be expanded or rewritten by the rules*
    - **a set of terminal symbols (T)**
      - *these symbols cannot be expanded*
    - **production rules (P) of the form**
      - LHS $\rightarrow$ RHS
      - In regular and CF grammars, LHS must be a single non-terminal symbol
      - RHS: a sequence of terminal and non-terminal symbols: possibly with restrictions, e.g. for regular grammars
    - **a designated start symbol (S)**
      - a non-terminal to start the derivation

- **Language**
  - set of terminal strings generated by <N,T,P,S>
  - e.g. through a top-down derivation

Definite Clause Grammars (DCG)

Background

- a "typical" formal grammar contains 4 things
- <N,T,P,S>
  - a set of non-terminal symbols (**N**)
  - a set of terminal symbols (**T**)
  - production rules (**P**) of the form LHS $\rightarrow$ RHS
  - a designated start symbol (**S**)

Example grammar (regular):

$S \rightarrow aB$
$B \rightarrow aB$
$B \rightarrow bC$
$B \rightarrow b$
$C \rightarrow bC$
$C \rightarrow b$

Notes:

- Start symbol: S
- Non-terminals: {S,B,C} (*uppercase letters*)
- Terminals: {a,b} (*lowercase letters*)

DefiniteClause Grammars (DCG)

- **Example**
  - **Formal grammar** — **DCG format**
  - $S \rightarrow aB$ — `s --> [a],b.`
  - $B \rightarrow aB$ — `b --> [a],b.`
  - $B \rightarrow bC$ — `b --> [b],c.`
  - $B \rightarrow b$ — `b --> [b].`
  - $C \rightarrow bC$ — `c --> [b],c.`
  - $C \rightarrow b$ — `c --> [b].`

- **Notes:**
  - Start symbol: S
  - Non-terminals: {S,B,C}
  - (*uppercase letters*)
  - Terminals: {a,b}
  - (*lowercase letters*)

DCG format:

- **both** terminals and non-terminal symbols begin with lowercase letters
  - *variables begin with an uppercase letter (or underscore)*
- `-->` is the rewrite symbol
- terminals are enclosed in square brackets (*list notation*)
- nonterminals don't have square brackets surrounding them
- the comma (`,`) represents the concatenation symbol
- a period (`.`) is required at the end of every DCG rule

Regular Grammars

- Regular or Chomsky hierarchy type-3 grammars
  - are a class of formal grammars with a restricted RHS
    - LHS → **RHS** *"LHS rewrites/expands to RHS"*
    - *all rules contain only a single non-terminal, and (possibly) a single terminal) on the right hand side*

- **Canonical Forms:**

```
x --> y, [t].        x --> [t].
```
(***left recursive***)

or

```
x --> [t], y.        x --> [t].
```
(***right recursive***)

Terminology:
or *"left/right linear"*

  - where `x` and `y` are non-terminal symbols and
  - `t` (enclosed in square brackets) represents a terminal symbol.
- **Note**:
  - *<u>can't</u> mix these two forms (and still have a regular grammar)!*
  - *can't have both left and right recursive rules in the same grammar*

Definite Clause Grammars (DCG)

- What language does our regular grammar generate?

> *one or more a's followed by one or more b's*

```
1. s --> [a],b.
2. b --> [a],b.
3. b --> [b],c.
4. b --> [b].
5. c --> [b],c.
6. c --> [b].
```

- by writing the grammar in Prolog,
- we have a ready-made recognizer program
  - *no need to write a separate grammar rule interpreter* (in this case)
- **Example query (set membership):**
  - `?- s([a,a,b,b,b],[]).`
  - Yes
  - `?- s([a,b,a],[]).`
- **Note**:
  - Query uses the start symbol `s` with two arguments:
  - (1) sequence (as a list) to be recognized and
  - (2) the empty list `[]`

Prolog lists:
In square brackets, separated by commas
e.g. `[a]` `[a,b,c]`

Definite Clause Grammars (DCG)

ling538-20 — swipl — 80×26

```
ling538-20$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.0)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [apbp].
true.

?- s([a,b],[]).
true.

?- s([a,b,b],[]).
true ;
false.

?- s([a,a,a,b,b],[]).
true ;
false.

?- s([b,a,b],[]).
false.

?-
```

- file on course webpage:
  - `apbp.prolog`

Prolog lists revisited

- Perl lists:
  - `@list = ("a", "b", "c");`
  - `@list = qw(a b c);`
  - `@list = ();`

Python lists:
`list = ["a", "b", "c"]`

`list = []`

- Prolog lists:
  - `List = [a, b, c]`
  - `List = [a|[b|[c|[]]]]`
  - `List = []`

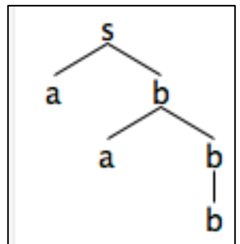
(**L**`ist` is a variable; a – c are atoms)

(a = head, tail = `[b|[c|[]]]`)

Mixed notation:
[a|[b,c]]
[a,b|[c]]

Regular Grammars

- Tree representation
  - Example
    - `?- s([a,a,b],[]).`
      `true`



There's a choice of rules for nonterminal b: Prolog tries the first rule

Through backtracking
It can try other choices

Derivation:
s

[a], **b**	(rule 1)
[a],[a],**b**	(rule 2)
[a],[a],[b]	(rule 4)

all terminals, so we stop

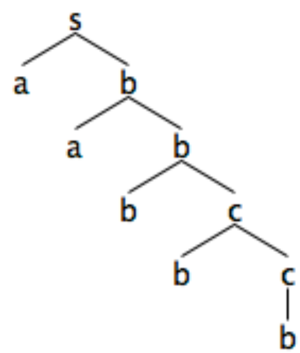
```
1. s --> [a],b.
2. b --> [a],b.
3. b --> [b],c.
4. b --> [b].
5. c --> [b],c.
6. c --> [b].
```

our a regular grammar

Using trace, we can observe the progress of the derivation…

Regular Grammars

- Tree representation
  - Example
    - `?- s([a,a,b,b,b],[]).`




```
1. s --> [a],b.
2. b --> [a],b.
3. b --> [b],c.
4. b --> [b].
5. c --> [b],c.
6. c --> [b].
```

Derivation:	
s	
[a], **b**	(rule 1)
[a],[a],**b**	(rule 2)
[a],[a],[b],**c**	(rule 3)
[a],[a],[b],[b],**c**	(rule 5)
[a],[a],[b],[b],[b]	(rule 6)

Prolog Derivations

- Prolog’s computation rule:
  - Try first matching rule in the database
    (remember others for backtracking)
  - Backtrack if matching rule leads to failure
  - undo and try next matching rule
    (or if asked for more solutions)
- For grammars:
  - Top-down left-to-right derivations
    - **left-to-right** = expand leftmost nonterminal first
    - Leftmost expansion done recursively = **depth-first**

Prolog Derivations

For a top-down derivation, logically, we have:

- **Choice**
  - about which rule to use for nonterminals b and c
- **No choice**
  - About which nonterminal to expand next

```
1. s --> [a],b.
2. b --> [a],b.
3. b --> [b],c.
4. b --> [b].
5. c --> [b],c.
6. c --> [b].
```

- Bottom up derivation for [a,a,b,b]

1.	[a],[a],[b],[b]	
2.	[a],[a],[b],c	(rule 6)
3.	[a],[a],b	(rule 3)
4.	[a],b	(rule 2)
5.	s	(rule 1)

Prolog doesn't give you bottom-up derivations for free
… *you'd have to program it up separately*

SWI Prolog

- Grammar rules are translated when the program is loaded into Prolog rules.
- Sheds light on the mystery why we have to type two arguments with the nonterminal at the command prompt
- Recall list notation:
  - [1|[2,3,4]] = [1,2,3,4]

```
1. s --> [a],b.
2. b --> [a],b.
3. b --> [b],c.
4. b --> [b].
5. c --> [b],c.
6. c --> [b].
```

```
1. s([a|A], B) :- b(A, B).
2. b([a|A], B) :- b(A, B).
3. b([b|A], B) :- c(A, B).
4. b([b|A], A).
5. c([b|A], B) :- c(A, B).
6. c([b|A], A).
```

FSA and a Regular Grammar

- Regular Grammar in Prolog.

4. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

- Let’s begin with something like (`bbp.prolog`):
  - `s --> [b], b.`
  - `s --> [b], s.`
  - `b --> [b].`
  - (start symbol `s`; grammar generates bb^+)

Let’s modify this grammar!

```
?- [bbp].
true.

?- s([b,b],[]).
true ;
false.

?- s([b,b,b],[]).
true ;
false.

?- s([b,b,b,a],[]).
false.

?- s([b],[]).
false.

?- s([],[]).
false.
```

FSA and a Regular Grammar

- Regular Grammar in Prolog.

4. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

- Let's begin with something like:
  - `s --> [b], b.`
  - `s --> [b], s.`
  - `b --> [b].`
  - (start symbol `s`; grammar generates bb^+)

It enumerates too!

```
?- s(L,[]).
L = [b, b] ;
L = [b, b, b] ;
L = [b, b, b, b] ;
L = [b, b, b, b, b] ;
L = [b, b, b, b, b, b] ;
L = [b, b, b, b, b, b, b] ;
L = [b, b, b, b, b, b, b, b] ;
L = [b, b, b, b, b, b, b, b, b] ;
L = [b, b, b, b, b, b, b, b, b|...] ;
L = [b, b, b, b, b, b, b, b, b|...]
```

FSA and a Regular Grammar

4. the set of all strings from the alphabet a, b such that each a is immediately preceded by and immediately followed by a b;

- Regular Grammar in Prolog notation (bab.prolog):
  - `s --> [].` % (s = ”start state”)
  - `s --> [b], seen_b.` % (”seen a b”)
  - `s --> [b], s.`

  - `seen_b --> [a], seen_a.` % (”expect a b” next)

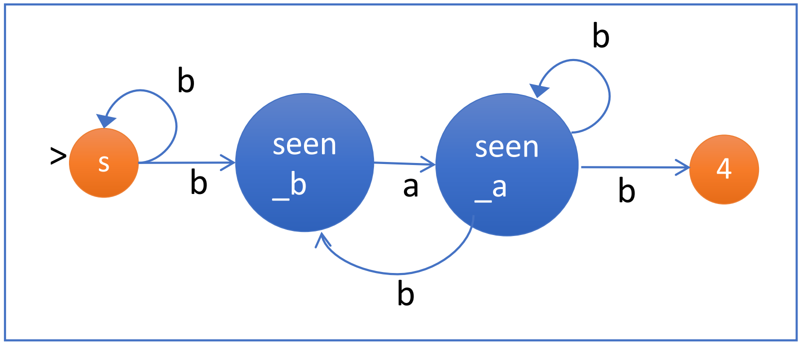
  - `seen_a --> [b].`
  - `seen_a --> [b], seen_b.`
  - `seen_a --> [b], seen_a.`

FSA and a Regular Grammar

- Compare the FSA with our Regular Grammar (RG) `bab.prolog`
  - `s --> [].`   `% (s = ` *start state*`)`
  - `s --> [b], seen_b.`
  - `s --> [b], s.`

  - `seen_b --> [a], seen_a.`

  - `seen_a --> [b].`
  - `seen_a --> [b], seen_b.`
  - `seen_a --> [b], seen_a.`

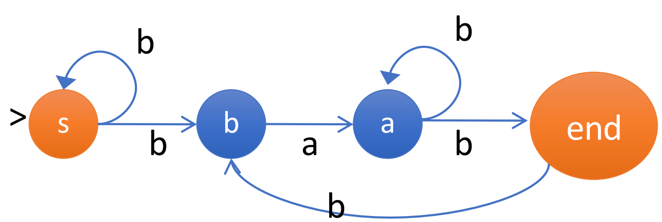


There is a straightforward correspondence between right recursive RGs and FSA

FSA and a Regular Grammar

- Informally, we can convert RG to a FSA
  - by treating
  - non-terminals as states
  - and introducing (new) states for rules of the form x --> [a].

[Powerpoint animation]
in order of the RG rules




```
1.  s --> [].
2.  s --> [b], b.
3.  s --> [b], s.
4.  b --> [a], a.
5.  a --> [b].
6.  a --> [b], b.
7.  a --> [b], a.
```

FSA and a Regular Grammar

- File `bab.prolog`:

```
?- [bab]. % load
true.

?- s([b,a,b],[]).
true ;
false.

?- s([b,a,a,b],[]).
false.
```

```
?- s([b,b],[]).
true.

?- s([b],[]).
true.

?- s([],[]).
true.

?- s([c],[]).
false.
```

```
?- s([b,a,b,b,a],[]).
false.

?-
s([b,a,b,b,a,b],[]).
true ;
false.
```