LING/C SC/PSYC 438/538

Lecture 24

Prof. Sandiway Fong

Today's Topics

- Continuing from last time: the bab grammar
- Beyond regular grammars …
- How to explicitly compute a parse tree
  - by adding an extra argument to for each grammar rule
- Left recursion and infinite loops

Reminder! 538 Presentations

- *see my email and 2nd last lecture for details*

Presenter	Date	Topic	Pages
Han Han	Dec 1st	23.1-2 Information Retrieval and IR-based Factoid Question Answering	495–503
Hannia Rojas Bareda	Nov 24th	12.3 Some Grammar Rules for English	266-270
Arun Sanyal	Nov 24th	24.1 Properties of Human Conversation (Chatbots & Dialogue Systems)	425-431
Hyeri Lee	Nov 24th	20.1-20.3 Defining Emotion / Creating Affect Lexicons by Human Labeling	166-173

Last Time: 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;

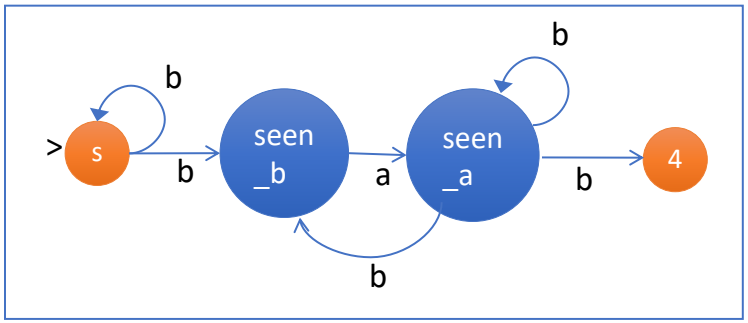
- Regular Grammar in Prolog notation (`bab.prolog`):
  - `s --> [].` % (`s` = "start state")
  - `s --> [b], seen_b.` % ("seen a `b`")
  - `s --> [b], s.`
  - `seen_b --> [a], seen_a.` % ("expect a `b`" next)
  - `seen_a --> [b].`
  - `seen_a --> [b], seen_b.`
  - `seen_a --> [b], seen_a.`

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.`

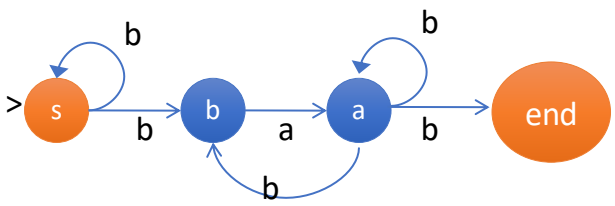


There is a straightforward correspondence between right recursive RGs and FSA

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

- Load in grammar from `bab.prolog`:

```
?- [bab]. % load
• true.
```

- Ask membership questions:

```
?- 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.
```

FSA and a Regular Grammar

What happens when you run the Prolog query `s(List,[])` ?

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

b(ab)+|ε
only a sub-language!

FSA and a Regular Grammar

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

never get to this rule on line 9!

- **Note**:
  - the nonterminal seen_a has 3 rules (*lines* 7-9):
  - *what happens if we flip lines 8 and 9*?
  - **Recall**: Prolog's computational rule is simple (*try first listed rule first*)

FSA and a Regular Grammar

Question 3: Rearrange the order of the rules so that the query `s(List, [])`., followed by ; can generate strings `[]`, `[b,a,b]`, `[b,a,b,b]`, `[b,a,b,b,b]` *etc.*

```
?- [bab2].
true.

?- s(List,[]).
List = [] ;
List = [b, a, b] ;
List = [b, a, b, b] ;
List = [b, a, b, b, b] ;
List = [b, a, b, b, b, b] ;
List = [b, a, b, b, b, b, b] ;
List = [b, a, b, b, b, b, b, b] ;
List = [b, a, b, b, b, b, b, b, b]
```

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

bab+|ε
is another sub-language!

Breadth-first Search: enumeration is possible




```
?- [bab].
true.

?- [id_meta].
true.

?- id(s(L,[])).
1.      L = [] ;
2.      L = [b] ;
3.      L = [b, a, b] ;
4.      L = [b, b] ;
5.      L = [b, a, b, b] ;
6.      L = [b, b, a, b] ;
7.      L = [b, b, b] ;
8.      L = [b, a, b, a, b] ;
9.      L = [b, a, b, b, b] ;
10.     L = [b, b, a, b, b] ;
11.     L = [b, b, b, a, b] ;
12.     L = [b, b, b, b] ;
13.     L = [b, a, b, a, b, b] ;
14.     L = [b, a, b, b, a, b] ;
15.     L = [b, a, b, b, b, b] ;
16.     L = [b, b, a, b, a, b] ;
17.     L = [b, b, a, b, b, b] ;
18.     L = [b, b, b, a, b, b] ;
19.     L = [b, b, b, b, a, b] ;
20.     L = [b, b, b, b, b] ;
21.     L = [b, a, b, a, b, a, b] ;
22.     L = [b, a, b, a, b, b, b] ;
23.     L = [b, a, b, b, a, b, b] ;
24.     L = [b, a, b, b, b, a, b] ;
25.     L = [b, a, b, b, b, b, b] ;
26.     L = [b, b, a, b, a, b, b] ;
27.     L = [b, b, a, b, b, a, b] ;
28.     L = [b, b, a, b, b, b, b] ;
29.     L = [b, b, b, a, b, a, b] ;
30.     L = [b, b, b, a, b, b, b] ;
31.     L = [b, b, b, b, a, b, b] ;
32.     L = [b, b, b, b, b, a, b] ;
33.     L = [b, b, b, b, b, b] ;
34.     L = [b, a, b, a, b, a, b, b] ;
35.     L = [b, a, b, a, b, b, a, b] ;
36.     L = [b, a, b, a, b, b, b, b] ;
37.     L = [b, a, b, b, a, b, a, b] ;
38.     L = [b, a, b, b, a, b, b, b] ;
39.     L = [b, a, b, b, b, a, b, b] ;
40.     L = [b, a, b, b, b, b, a, b] ;
41.     L = [b, a, b, b, b, b, b, b] ;
42.     L = [b, b, a, b, a, b, a, b] ;
43.     L = [b, b, a, b, a, b, b, b] ;
44.     L = [b, b, a, b, b, a, b, b] ;
45.     L = [b, b, a, b, b, b, a, b] ;
46.     L = [b, b, a, b, b, b, b, b] ;
47.     L = [b, b, b, a, b, a, b, b] ;
48.     L = [b, b, b, a, b, b, a, b] ;
49.     L = [b, b, b, a, b, b, b, b] ;
50.     L = [b, b, b, b, a, b, a, b] ;
51.     L = [b, b, b, b, a, b, b, b] ;
52.     L = [b, b, b, b, b, a, b, b] ;
53.     L = [b, b, b, b, b, b, a, b] ;
54.     L = [b, b, b, b, b, b, b, b] ;
L = [b, a, b, a, b, a, b, a, b] ;
… and so on
```

Iterative Deepening

- Suppose you have only the **depth-first search strategy**, e.g. Prolog, but you want to do **breadth-first search**, i.e. explore all alternatives simultaneously
- Let level n = the depth of the search. Then:
  - Do complete depth-first search to level 1
  - Do complete depth-first search to level 2
  - and so …
- This is the same as breadth-first search (*but less efficient*)
- It is **fair computation**, i.e. *every rule gets its turn*

Breadth-first Search: enumeration is possible

- How many a's in a string of length N?
  - Suppose we have 3 *a*'s in a string
  - Every *a* needs a *b* on either side, so the minimum length of the string has to be 7.
- **Formula**:
  - let #a be the number of a's.
  - minlen(#a) = #a * 3 – (#a – 1), for #a ≥ 1
  - minlen(#a) = #a * 2 + 1, for #a ≥ 1
  - minlen(1) = 3; minlen(2) = 5; minlen(3) = 7, and so on
  - Note that minlen must always be **odd**
  - Length is **even**: must be all b's or any odd string (of length one less) in the language + one additional b.

Breadth-first Search: enumeration is possible

bab grammar:

- length = 0 (1 solution)
  - `L = [] ;`
- length = 1 (1 solution)
  - `L = [b] ;`
- length = 2 (1 solution)
  - `L = [b, b] ;`
- length = 3 (2 solutions)
  - `L = [b, a, b] ;`
  - `L = [b, b, b] ;`
- length = 4 (3 solutions)

```
1) L = [b, a, b, b] ;
2) L = [b, b, a, b] ;
3) L = [b, b, b, b] ;
```

- length = 5 (5 solutions)

```
1) L = [b, a, b, a, b] ;
2) L = [b, a, b, b, b] ;
3) L = [b, b, a, b, b] ;
4) L = [b, b, b, a, b] ;
5) L = [b, b, b, b, b] ;
```

- length = 6 (8 solutions)

```
1) L = [b, a, b, a, b, b] ;
2) L = [b, a, b, b, a, b] ;
3) L = [b, a, b, b, b, b] ;
4) L = [b, b, a, b, a, b] ;
5) L = [b, b, a, b, b, b] ;
6) L = [b, b, b, a, b, b] ;
7) L = [b, b, b, b, a, b] ;
8) L = [b, b, b, b, b, b] ;
```

Beyond Regular Languages

- Beyond regular languages
  - $a^n b^n$ = {ab, aabb, aaabbb, aaaabbbb, ... } n≥1
  - is not a regular language
- That means no FSA, RE or RG can be built for this language

1. We only have a finite number of states to play with …
2. We're only allowed simple free iteration (looping)
3. Pumping Lemma proof

Beyond Regular Languages

- Language
  - $a^n b^n$ = {ab, aabb, aaabbb, aaaabbbb, ... } n≥1
- A regular grammar extended to allow both left and right recursive rules can accept/generate it:
- File `anbn.prolog`

```
1. a --> [a], b.
2. b --> [b].
3. b --> a, [b].
```

- Example:

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

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

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

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

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

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

Set membership

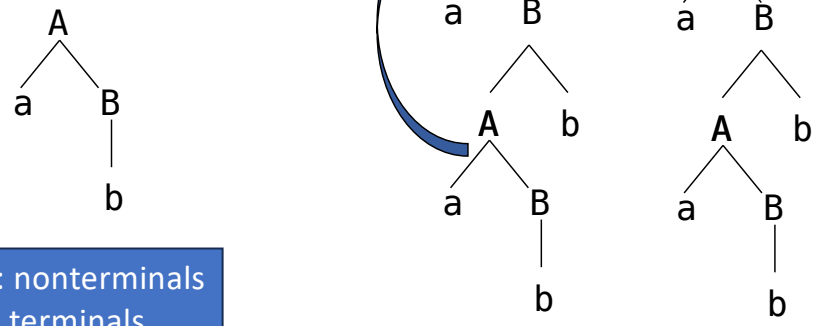
```
?- a(L,[]).
L = [a, b] ;
L = [a, a, b, b] ;
L = [a, a, a, b, b, b] ;
L = [a, a, a, a, b, b, b, b] ;
L = [a, a, a, a, a, b, b, b, b, b]
```

Set enumeration

Beyond Regular Languages

- Intuition:
  - grammar implements the stacking of partial trees balanced for a's and b's:

```
1. a --> [a], b.
2. b --> [b].
3. b --> a, [b].
```



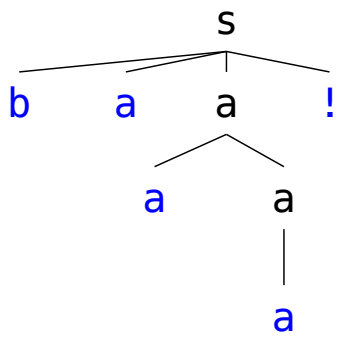
A, B: nonterminals
a, b: terminals

- A type-2 or context-free grammar (CFG) has no restrictions on what can go on the RHS of a grammar rule
- **Note**: *CFGs still have a single nonterminal limit for the LHS of a rule.*
- Example:

```
1. s --> [a], [b].
2. s --> [a], s, [b].
```

Parse Trees using Terms

- **Tree:**

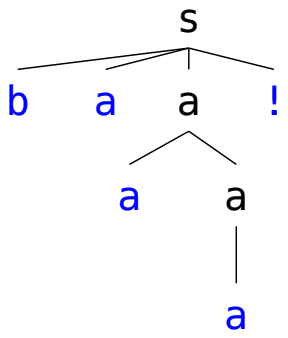


`s(b,a,a(a,a(a)),!)`

- Prolog term data structure:
  - `functor(arg`$_1$`,..,arg`$_n$`)`
  - each `arg`$_i$ could be another term or simple atom (symbol).
  - encodes hierarchy
  - allows a linear sequence of arguments

Extra Argument: Parse Tree

- **Recovering a parse tree**
  - when want Prolog to return more than just **true/false** answers
  - in case of **true**, we can *incrementally* compute a Prolog term representation of the parse
  - by adding an extra argument to each nonterminal
  - applies to all rules (*not just regular grammars*)



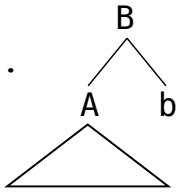
Extra Argument: Parse Tree

- **DCG**
  - `a --> [a], b.` (start symbol a)
  - `b --> [b].` (base case)
  - `b --> a, [b].` (left recursive case)

- **base case**
  - `b --> [b].`
  - `b(`*`subtree`*`) --> [b].`
  - `b(b(b)) --> [b].`

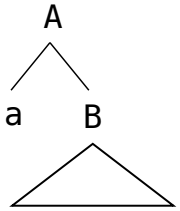


- **recursive case**
  - `b --> a, [b].`
  - `b(`*`subtree`*`) --> a(`*`subtree'`*`), [b].`
  - `b(b(A,b)) --> a(A), [b].`



- **Idea**:
  - for each nonterminal, add an argument to store its subtree

- **start symbol case**
  - `a --> [a], b.`
  - `a(`*`tree`*`) --> [a], b(`*`subtree'`*`).`
  - `a(a(a,B)) --> [a], b(B).`



Extra Argument: Parse Tree

- **Prolog grammar:**
  - anbn.prolog
  - `a --> [a], b.`
  - `b --> [b].`
  - `b --> a, [b].`
- **Equivalent Prolog grammar computing a parse**
  - anbn2.prolog
  - `a(a(a,B)) --> [a], b(B).`
  - `b(b(b)) --> [b].`
  - `b(b(A,b)) --> a(A), [b].`

Every nonterminal *x* has an extra argument *Term*: *x*(*Term*)

Extra Argument: Parse Tree

- anbn2.prolog
  1. `a(a(a,B)) --> [a], b(B).`
  2. `b(b(b)) --> [b].`
  3. `b(b(A,b)) --> a(A), [b].`
- To run the code, we also have to add the extra argument (for our parse) to the query.

```
?- [anbn2].
true.

?- a(Tree, [a,b], []).
Tree = a(a, b(b)) ;
false.

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

?- a(Tree, [a,a,b,b], []).
Tree = a(a, b(a(a, b(b)), b)) ;
false.
```

Extra Arguments

- Extra arguments are powerful
  - they allow us to impose (grammatical) constraints and change the expressive power of the system
    - if used as read-able memory
    - but for computing a parse tree only: no.
- **Example**:
  - $a^n b^n c^n$ n>0 is not a context-free language (type-2)
  - *i.e. you cannot write rules of the form* X --> RHS *to generate this language*
  - in fact, it's context-sensitive (type-1)
  - but can use context-free rules plus an extra argument to constrain #a,b,c's.

Left Recursion and Set Enumeration

- Example (`lrrg.prolog`):

```
1. s --> a, [!].
2. a --> ba, [a].
3. a --> a, [a].
4. ba --> b, [a].
5. b --> [b].
```

- Grammar is:
  - a regular grammar
  - left recursive (*nonterminal on left*)

- **Question**
  - What is the language of this grammar?
- Answer: *Sheeptalk*!
  - $ba^n!$ (# a's = n > 1)
- Sentential forms:
  - s
  - a!
  - baa!
  - baa!
  - baa!

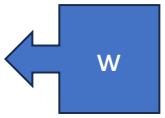
Underscores used here to indicate a nonterminal

Left Recursion and Set Enumeration

```
?- [lrrg].
true.

?- s(String,[]).
String = [b, a, a, !] ;
String = [b, a, a, a, !] ;
String = [b, a, a, a, a, !] ;
String = [b, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, a, a|...] ;
String = [b, a, a, a, a, a, a, a, a|...] [write]
String = [b, a, a, a, a, a, a, a, a, a, !] .
```

lrrg.prolog works for enumeration!



Left Recursion and Set Enumeration

- It also partially works for set membership:

```
?- s([b,a,a,!],[]).
true ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 84.2Mb, trail: 0Kb
ERROR:   Stack depth: 11,029,028, last-call: 0%, Choice points: 4
ERROR:   Probable infinite recursion (cycle):
ERROR:     [11,029,026] user:a([length:4], _22083396)
ERROR:     [11,029,025] user:a([length:4], _22083422)
   Exception: (11,029,026) a([b, a, a, !], _22083324) ? abort
% Execution Aborted
?-
```

Left Recursion and Set Enumeration

- What happens if we present a string not in Sheeptalk!?

lrrg.prolog doesn't work as a decider: response should be yes/no.

```
?- s([b,a,a,b,a,a,!],[]).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 84.1Mb, trail: 0Kb
ERROR:   Stack depth: 11,028,563, last-call: 0%, Choice points: 4
ERROR:   Probable infinite recursion (cycle):
ERROR:     [11,028,561] user:a([length:7], _22059520)
ERROR:     [11,028,560] user:a([length:7], _22059546)
   Exception: (11,028,561) a([b, a, a, b, a, a, !], _22059448) ? abort
% Execution Aborted
```

Left Recursion and Set Enumeration

- left recursive regular grammar `lrrg.prolog`:

```
1. s --> a, [!].
2. a --> ba, [a].
3. a --> a, [a].
4. ba --> b, [a].
5. b --> [b].
```

- Summary of Behavior:
  1. enumerates the language
  2. halts when presented with a string that is in the language
  3. doesn't halt when faced with a string not in the language
  - *unable to decide the language membership question*

Left Recursion and Set Enumeration

- left recursive regular grammar:

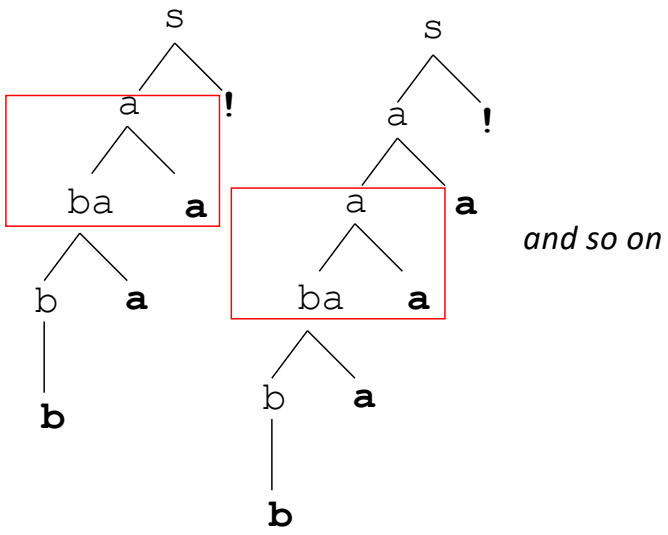
```
1. s --> a, [!].
2. a --> ba, [a].
3. a --> a, [a].
4. ba --> b, [a].
5. b --> [b].
```

Choice point

- Behavior
  - halts when presented with a string that is in the language
  - doesn't halt when faced with a string not in the language

- derivation tree for `?- s(L,[]).` [Powerpoint animation]

L = [b,a,a,!]



Left Recursion and Set Enumeration

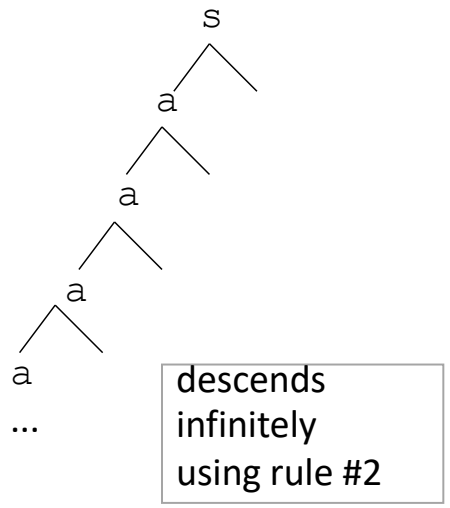
- However, this **slightly re-ordered** left recursive regular grammar:

```
1. s --> a, [!].
2. a --> a, [a].
3. a --> ba, [a].
4. ba --> b, [a].
5. b --> [b].
```

- (*rules 2 and 3 swapped*) won't halt when enumerating

- **Why**?

[Powerpoint animation]

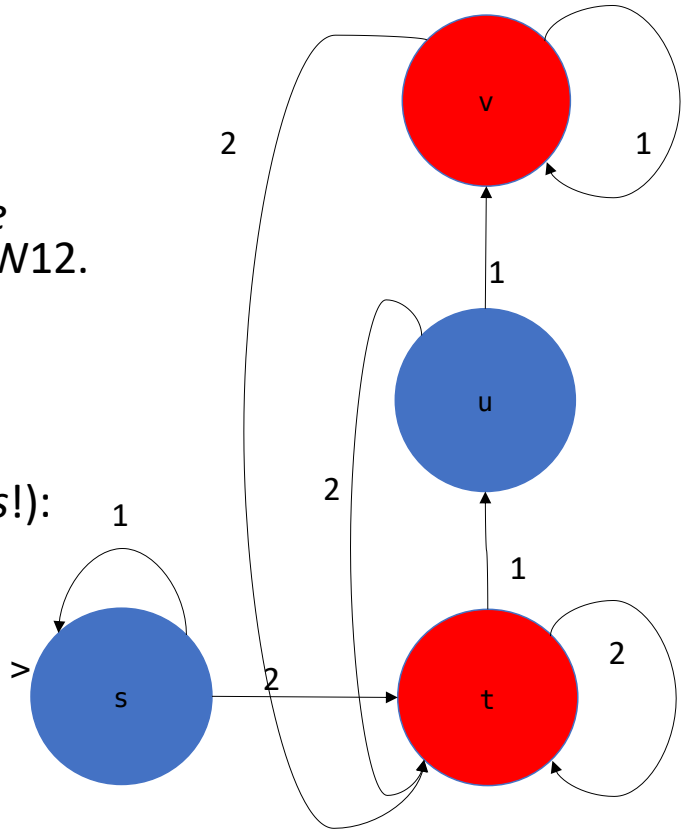


Homework 14

Question 1:

- Give a Prolog **regular grammar** (*beware of the canonical rules*!) for the DFSA at the end of HW12.
- How many rules do you have?
- **Hint:** `t --> [].`
  - encodes final state (*we permit this*!)
- Test it on (*make sure it gives the right answers*!):

```
1. s([1,1,2],[]).
2. s([1,1,2,2],[]).
3. s([2,1,2,1,1,1],[]).
4. s([1,1],[]).
5. s([2,2,1],[]).
6. s([2,1,2,1],[]).
```



Homework 14

Question 2:

- Does your grammar **enumerate** the language?
  - `s(L, []).`
- Does flipping the rules around help?
- Use the following to see answers printed in full:
  - `set_prolog_flag(answer_write_options,[max_depth(0)]).`

Homework 14

- Usual Rules
- Make sure you submit everything in one PDF file
- No links!
- Put your Prolog grammars as text inside the PDF file
- Due date: by Sunday midnight
- 438/538 Homework 14 *YOUR NAME*