LING/C SC 581:

Advanced Computational Linguistics

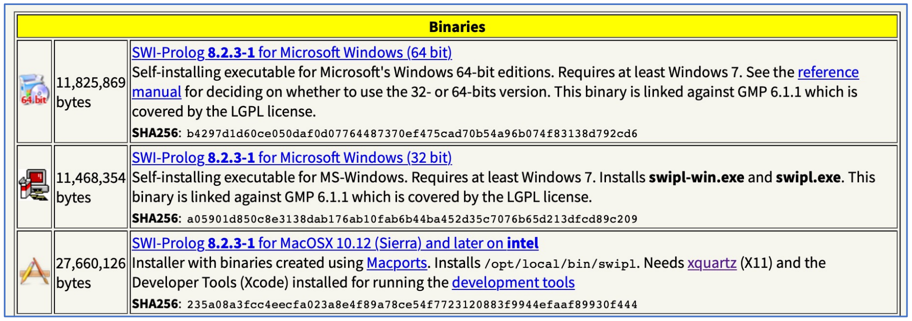
Lecture 23

Today's Topic

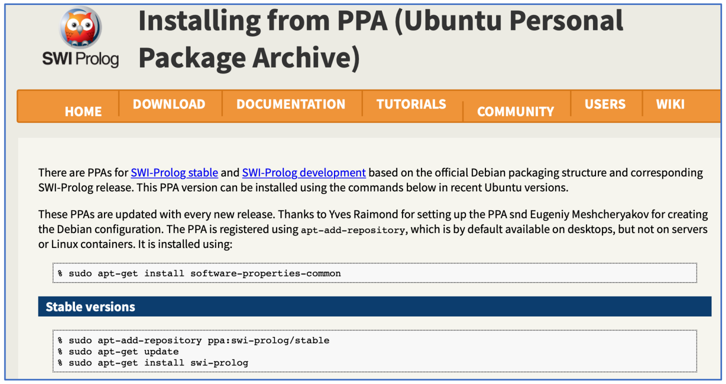
- Recent Chomsky video lecture (*I sent the link around*):
  - Possible **HLT-related** questions for Prof. Chomsky?
  - No guarantee he has time, but I can ask…
- Change of topic
  - writing our own grammars
  - but first, we must **refamiliarize** ourselves with a programming language, Prolog

SWI Prolog

- You should already have SWI Prolog installed on your machine
- If not, please install now!
- https://www.swi-prolog.org/download/stable



Binaries		
	11,825,869 bytes	SWI-Prolog **8.2.3-1** for Microsoft Windows (64 bit) Self-installing executable for Microsoft's Windows 64-bit editions. Requires at least Windows 7. See the reference manual for deciding on whether to use the 32- or 64-bits version. This binary is linked against GMP 6.1.1 which is covered by the LGPL license. **SHA256**: `b4297d1d60ce050daf0d07764487370ef475cad70b54a96b074f83138d792cd6`
	11,468,354 bytes	SWI-Prolog **8.2.3-1** for Microsoft Windows (32 bit) Self-installing executable for MS-Windows. Requires at least Windows 7. Installs **swipl-win.exe** and **swipl.exe**. This binary is linked against GMP 6.1.1 which is covered by the LGPL license. **SHA256**: `a05901d850c8e3138dab176ab10fab6b44ba452d35c7076b65d213dfcd89c209`
	27,660,126 bytes	SWI-Prolog **8.2.3-1** for MacOSX 10.12 (Sierra) and later on **intel** Installer with binaries created using Macports. Installs `/opt/local/bin/swipl`. Needs xquartz (X11) and the Developer Tools (Xcode) installed for running the development tools **SHA256**: `235a08a3fcc4eecfa023a8e4f89a78ce54f7723120883f9944efaaf89930f444`



Installing from PPA (Ubuntu Personal Package Archive)

There are PPAs for SWI-Prolog stable and SWI-Prolog development based on the official Debian packaging structure and corresponding SWI-Prolog release. This PPA version can be installed using the commands below in recent Ubuntu versions.

These PPAs are updated with every new release. Thanks to Yves Raimond for setting up the PPA snd Eugeniy Meshcheryakov for creating the Debian configuration. The PPA is registered using `apt-add-repository`, which is by default available on desktops, but not on servers or Linux containers. It is installed using:

```
% sudo apt-get install software-properties-common
```

Stable versions

```
% sudo apt-add-repository ppa:swi-prolog/stable
% sudo apt-get update
% sudo apt-get install swi-prolog
```

https://www.swi-prolog.org/build/PPA.html

SWI Prolog Cheatsheet

- **At the prompt** ?-
  1. `halt.` `^D`
  2. `listing.` `listing(`*name*`).`
  3. `[`*filename*`].` loads *filename.pl*
  4. `trace.`
  5. `notrace.`
  6. `debug.`
  7. `nodebug.`
  8. `spy(`*name*`).`
  9. `pwd.`
  10. `working_directory(_,Y).`
      switch directories to Y
- Anytime
  - `^C` (then **a**(bort) or **h**(elp) for other options)

Notation:

`\+`	negation
`,`	conjunction
`;`	disjunction
`:-`	if

Facts:

`predicate(`*Args*`).`

Rules:

`p(`*Args*`) :- q(`*Args*`) ,.., r(`*Args*`).`

Data structures:

list: `[a,..b]`

empty list: `[]`

head/tail: `[`*head*`|`*List*`]`

Atom:

name, number

Term:

`functor(arguments)`

`arguments:` comma-separated terms/atoms

Derivations

- Prolog's computation rule:
  - Try first matching (**grammar**) rule in the database
    (but remember other possibilities for backtracking)
  - Backtrack if matching rule leads to failure (or if asked by the user typing **;**)
  - undo variable bindings (i.e. *undo assignments*) and try next matching rule
- For grammars:
  - Top-down left-to-right derivations
    - **left-to-right** = expand leftmost nonterminal first
    - Leftmost expansion done recursively = **depth-first**

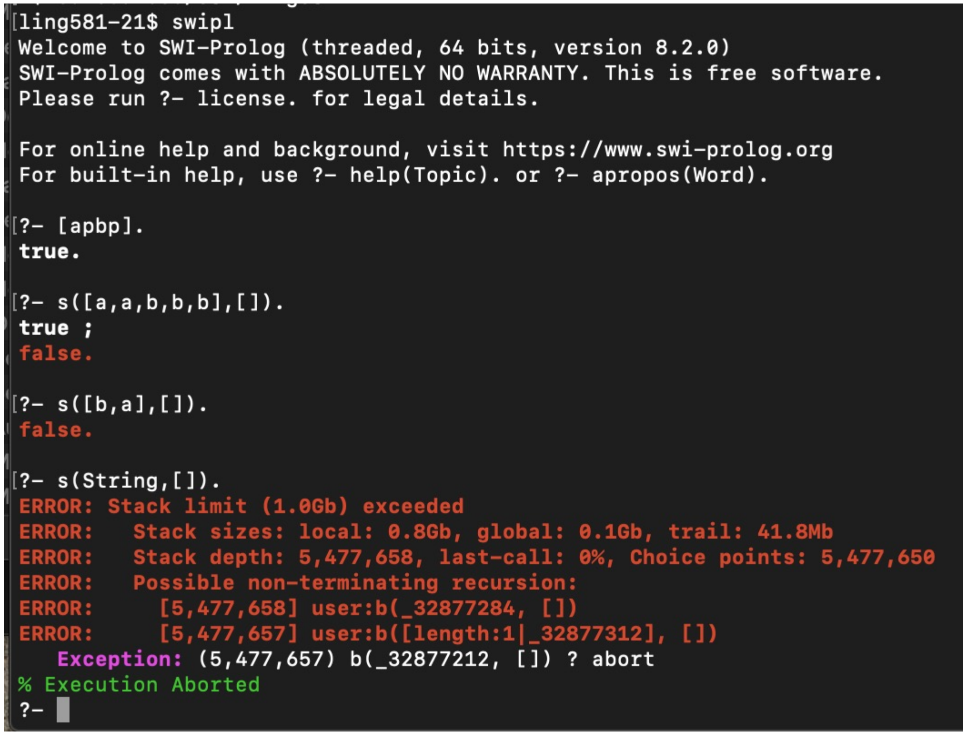
Definite Clause Grammars (DCG)

- a grammar is code, could be a recognizer program:
  - *no need to write a separate grammar rule interpreter* (in this case)
- **Example query**
  - `?- s([a,a,b,b,b],[]).` Yes
- **Note**:
  - Syntax of DCGs:
    - ---> "expands to"
    - terminal symbol enclosed in square brackets: [*terminal*]
    - non-terminal symbol, otherwise
  - Query uses the start symbol `s` with two arguments:
  - (1) sequence (as a list) to be recognized and
  - (2) the empty list `[]`

Grammar for a^+b^+

```
apbp.prolog:
s --> [a],b.
b --> [a],b.
b --> [b],c.
b --> [b].
c --> [b],c.
c --> [b].
```

Definite Clause Grammars (DCG)

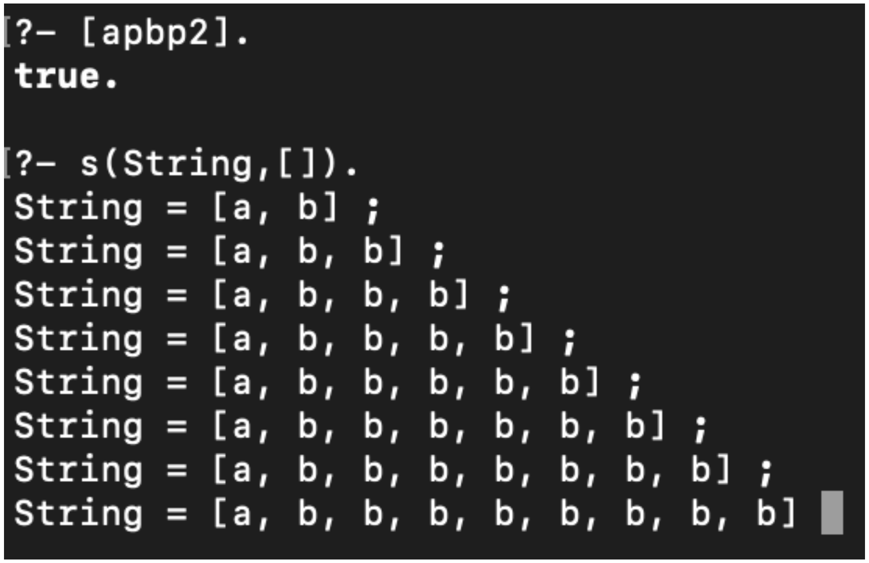


```
[ling581-21$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.0)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

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,a,b,b,b],[]).
true ;
false.

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

[?- s(String,[]).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.8Gb, global: 0.1Gb, trail: 41.8Mb
ERROR:   Stack depth: 5,477,658, last-call: 0%, Choice points: 5,477,650
ERROR:   Possible non-terminating recursion:
ERROR:     [5,477,658] user:b(_32877284, [])
ERROR:     [5,477,657] user:b([length:1|_32877312], [])
   Exception: (5,477,657) b(_32877212, []) ? abort
% Execution Aborted
?- 
```

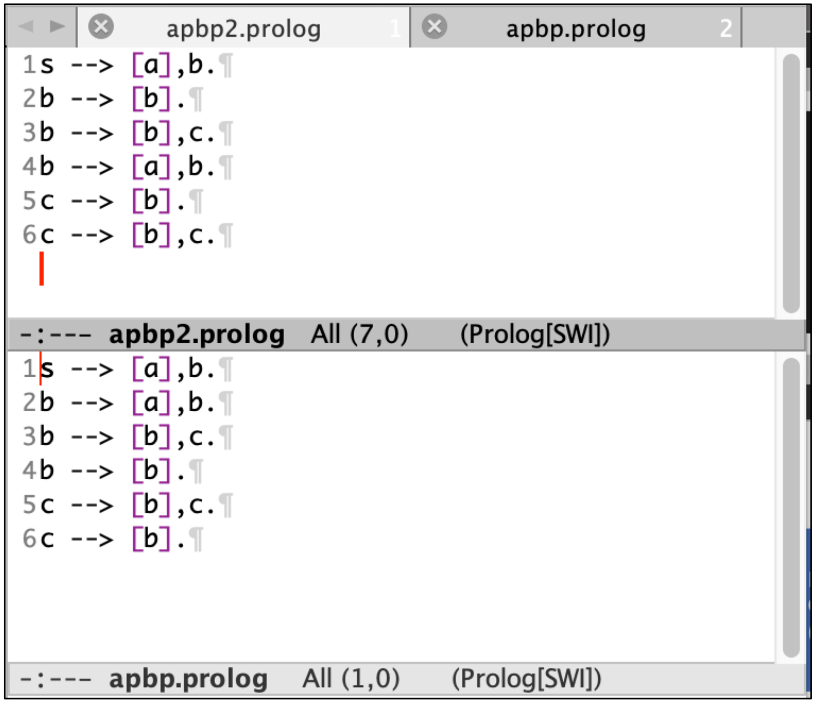
Infinite Loop: Doesn't enumerate at all!

Definite Clause Grammars (DCG)

Partial enumerator only! Why?



```
?- [apbp2].
true.

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



apbp2.prolog 1
apbp.prolog 2

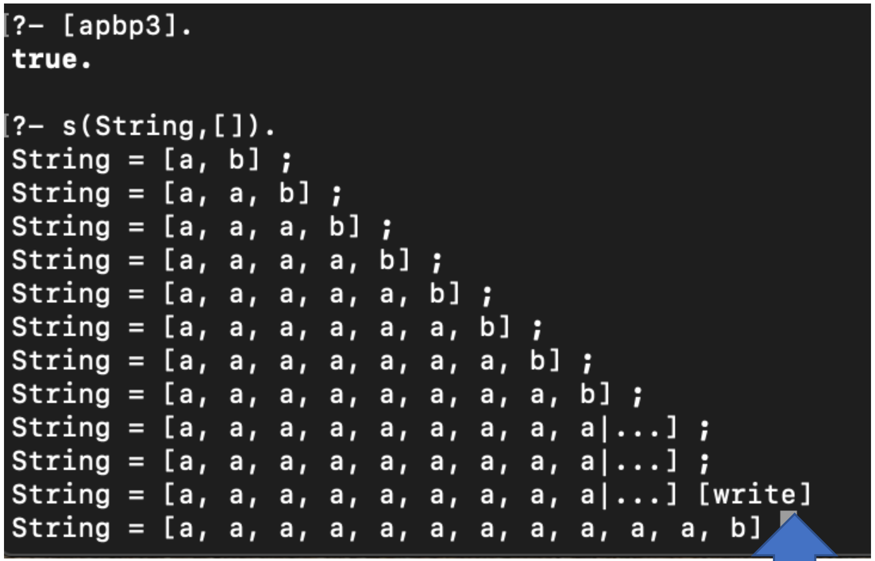
```
s --> [a],b.
b --> [b].
b --> [b],c.
b --> [a],b.
c --> [b].
c --> [b],c.
```

-:--- apbp2.prolog All (7,0) (Prolog[SWI])

```
s --> [a],b.
b --> [a],b.
b --> [b],c.
b --> [b].
c --> [b],c.
c --> [b].
```

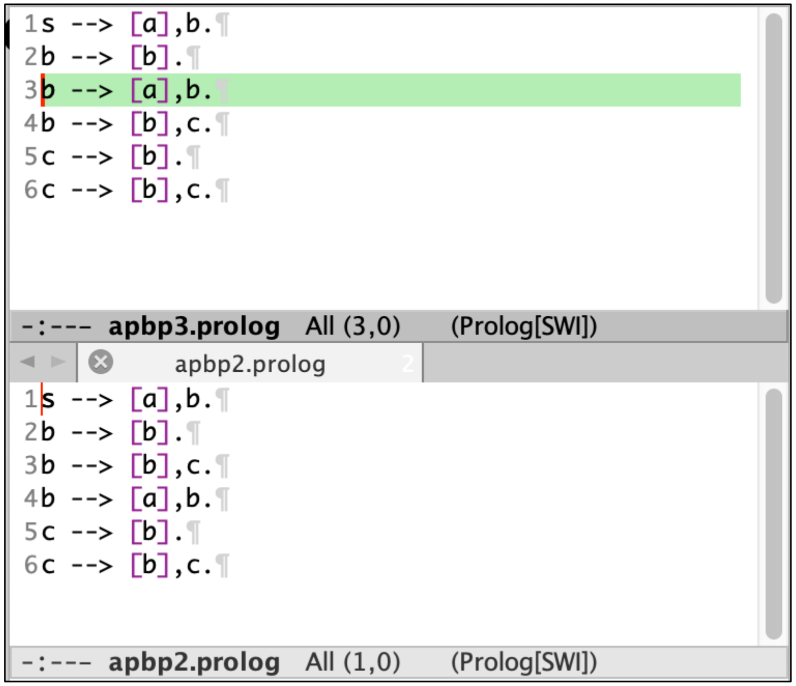
-:--- apbp.prolog All (1,0) (Prolog[SWI])

Definite Clause Grammars (DCG)



```
?- [apbp3].
true.

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

type w



```
s --> [a],b.
b --> [b].
b --> [a],b.
b --> [b],c.
c --> [b].
c --> [b],c.
```

-:--- **apbp3.prolog** All (3,0) (Prolog[SWI])

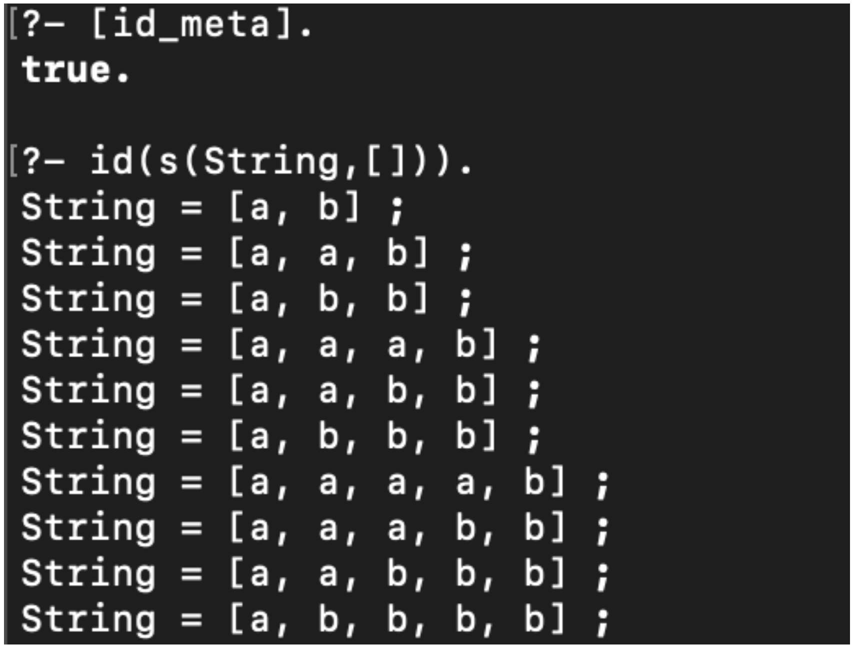
```
s --> [a],b.
b --> [b].
b --> [b],c.
b --> [a],b.
c --> [b].
c --> [b],c.
```

-:--- **apbp2.prolog** All (1,0) (Prolog[SWI])

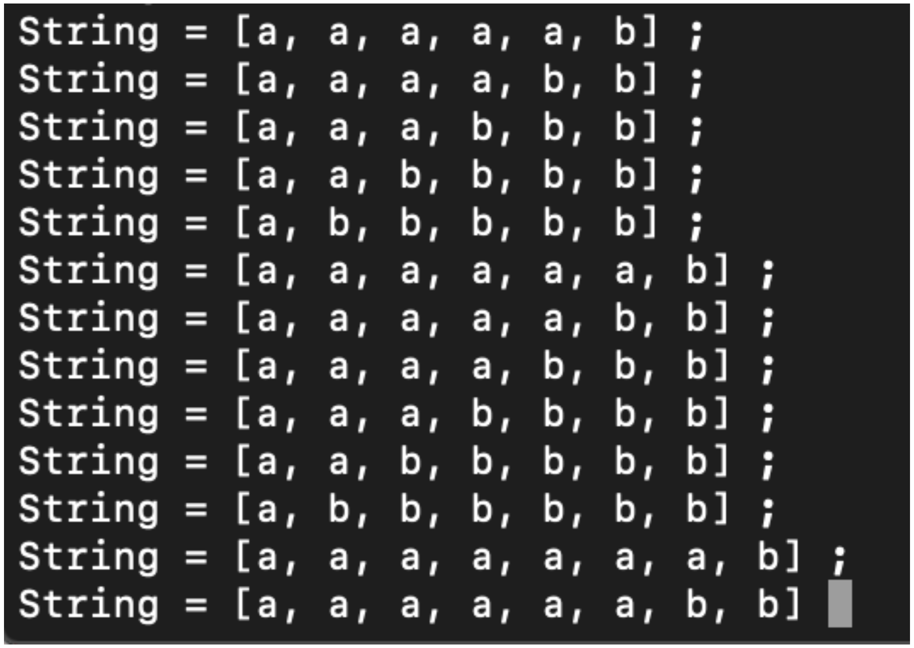
Definite Clause Grammars (DCG)

- How to guarantee enumeration?
- Iterative Deepening (ID):
  - Breadth-first search implemented in depth-first search
  - **Idea**:
    - find all solutions at depth N, remember them,
    - then search (all over) again to depth N+1,
    - and so on…

Definite Clause Grammars (DCG)



```
[?- [id_meta].
 true.

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



```
String = [a, a, a, a, a, b] ;
String = [a, a, a, a, b, b] ;
String = [a, a, a, b, b, b] ;
String = [a, a, b, b, b, b] ;
String = [a, b, b, b, b, b] ;
String = [a, a, a, a, a, a, b] ;
String = [a, a, a, a, a, b, b] ;
String = [a, a, a, a, b, b, b] ;
String = [a, a, a, b, b, b, b] ;
String = [a, a, b, b, b, b, b] ;
String = [a, b, b, b, b, b, b] ;
String = [a, a, a, a, a, a, a, b] ;
String = [a, a, a, a, a, a, b, b]
```

for strings of length N, there'll be N–1 solutions

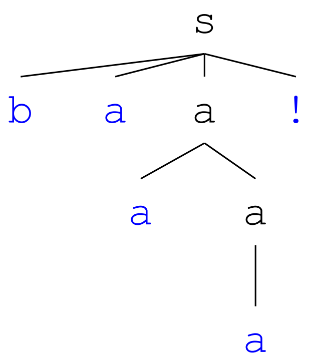
Extra Argument: Parse Tree

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

Example

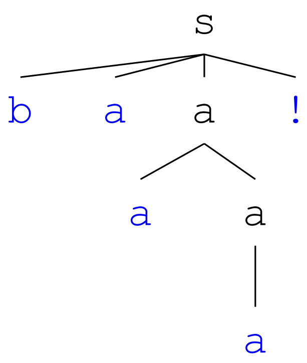
- *sheeptalk* **again**
- **DCG (non-regular, context-free):**

```
s --> [b], [a], a, [!].
a --> [a].        (base case)
a --> [a], a.  (recursive case)
```



Extra Argument: Parse Tree

- **Tree:**


s

b a a !

a a

a

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

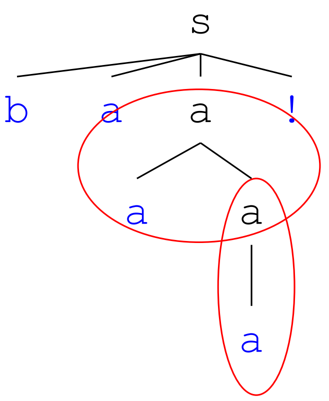
- **Prolog term data structure**:
  - hierarchical
  - allows sequencing of arguments
  - `functor(arg`$_1$`,..,arg`$_n$`)`
  - each `arg`$_i$ could be another term or simple atom

Extra Arguments: Parse Tree

- **DCG**
  - `s --> [b],[a], a, [!].`
  - `a --> [a].` (base case)
  - `a --> [a], a.` (right recursive case)

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

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



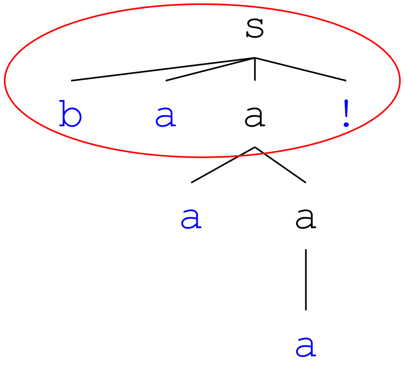
`s(b,a,a(a,a(a)),!)`

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

Extra Arguments: Parse Tree

- **Prolog grammar**
  - `s --> [b], [a], a, [!].`
  - `a --> [a].` (base case)
  - `a --> [a], a.` (right recursive case)
- **base and recursive cases**
  - `a(a(a)) --> [a].`
  - `a(a(a,A)) --> [a], a(A).`

- **start symbol case**
  - `s --> [b], [a], a, [!].`
  - `s(`*tree*`) --> [b], [a], a(`*subtree*`), [!].`
  - `s(s(b,a,A,!) ) --> [b], [a], a(A), [!].`



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

Extra Arguments: Parse Tree

- **Prolog grammar**
  - `s --> [b], [a], a, [!].`
  - `a --> [a].` (base case)
  - `a --> [a], a.` (right recursive case)

- **Equivalent Prolog grammar computing a parse**
  - `s(s(b,a,A,!)) --> [b], [a], a(A), [!].`
  - `a(a(a)) --> [a].`
  - `a(a(a,A)) --> [a], a(A).`

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

- 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 *that generate this language*
  - in fact, it's context-sensitive (type-1)

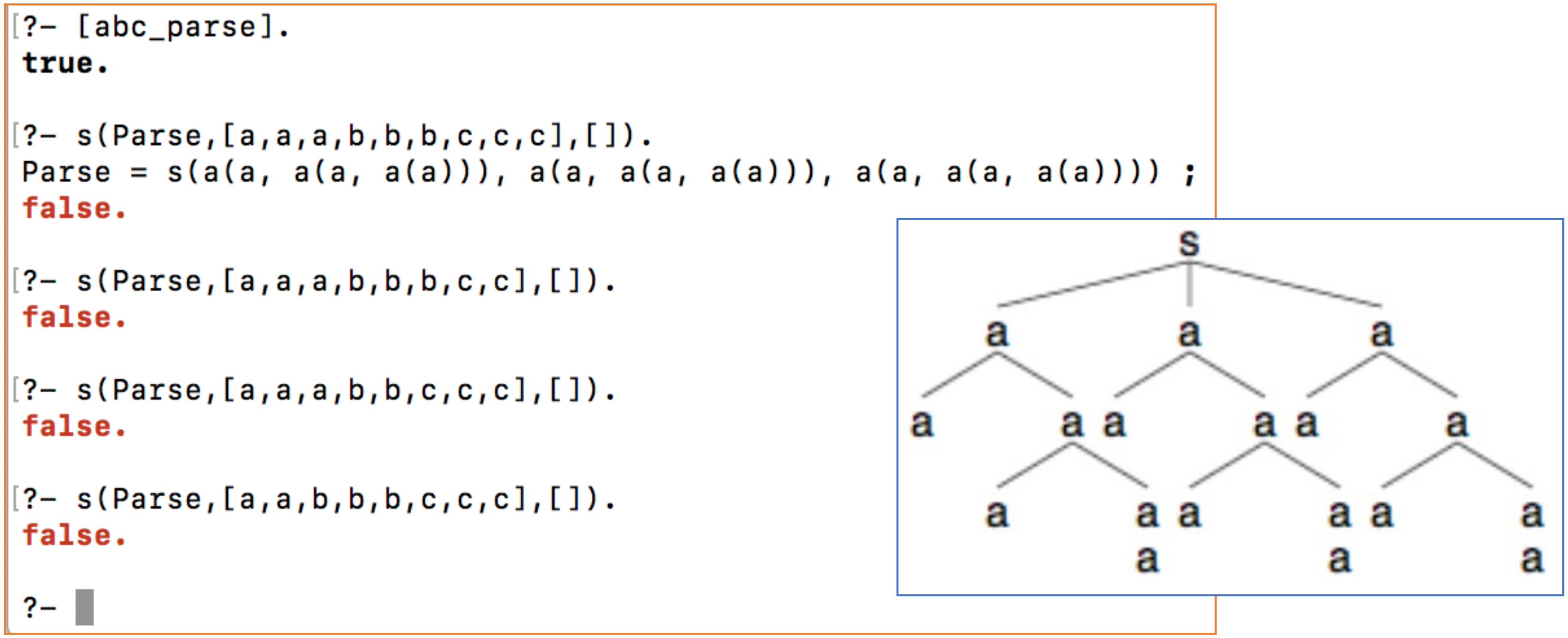
Language $\{a^n b^n c^n | n>0\}$

1. CFG (context-free grammar) + extra arguments for grammatical constraints
2. CFG + counting, cf. Perl
3. CSG (context-sensitive grammar) rules

Extra arguments

- A CFG+EA for $a^n b^n c^n$ n>0:

Set membership question

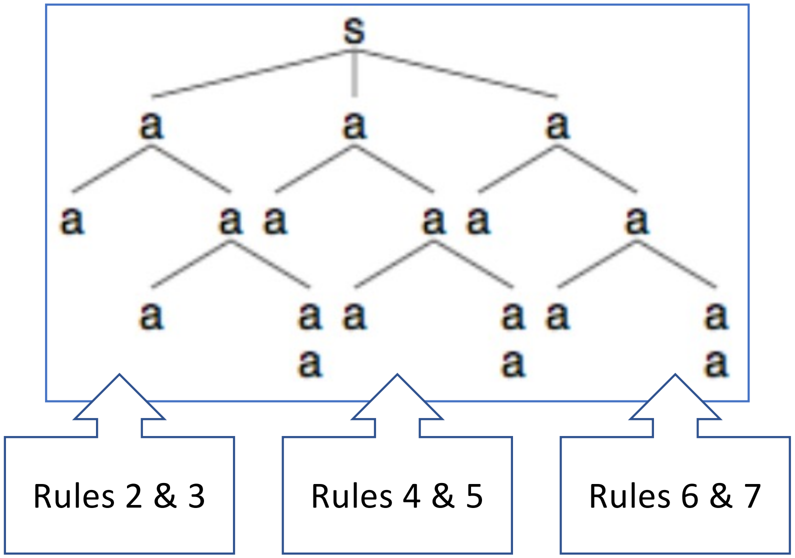


```
[?- [abc_parse].
true.

[?- s(Parse,[a,a,a,b,b,b,c,c,c],[]).
Parse = s(a(a, a(a, a(a))), a(a, a(a, a(a))), a(a, a(a, a(a)))) ;
false.

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

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

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

?- 
```

Extra arguments

- A context-free grammar (CFG) + extra argument (EA) for the context-sensitive language { $a^n b^n c^n$ | n>0}:

```
1. s(s(A,A,A)) --> a(A), b(A), c(A).
2. a(a(a)) --> [a].
3. a(a(a,X)) --> [a], a(X).
4. b(a(a)) --> [b].
5. b(a(a,X)) --> [b], b(X).
6. c(a(a)) --> [c].
7. c(a(a,X)) --> [c], c(X).
```



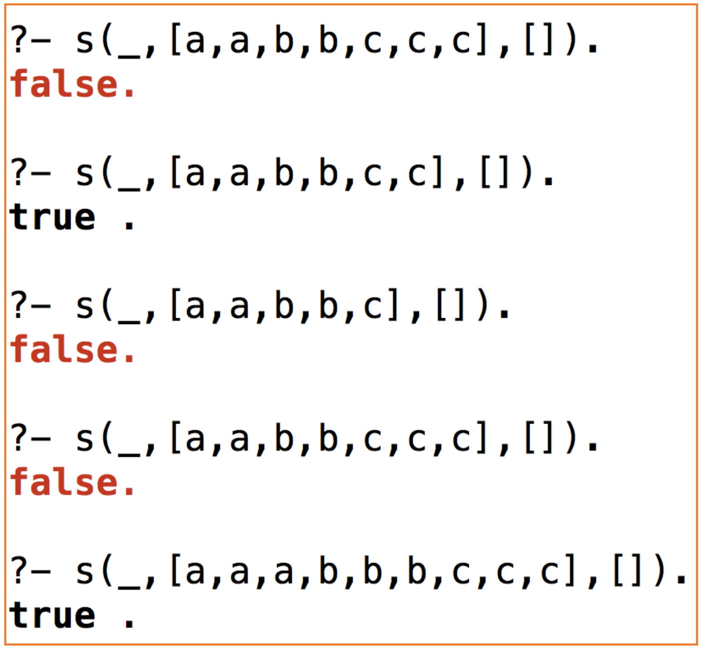
Rules 2 & 3

Rules 4 & 5

Rules 6 & 7

Extra arguments

- A CFG+EA for $a^n b^n c^n$ n>0:

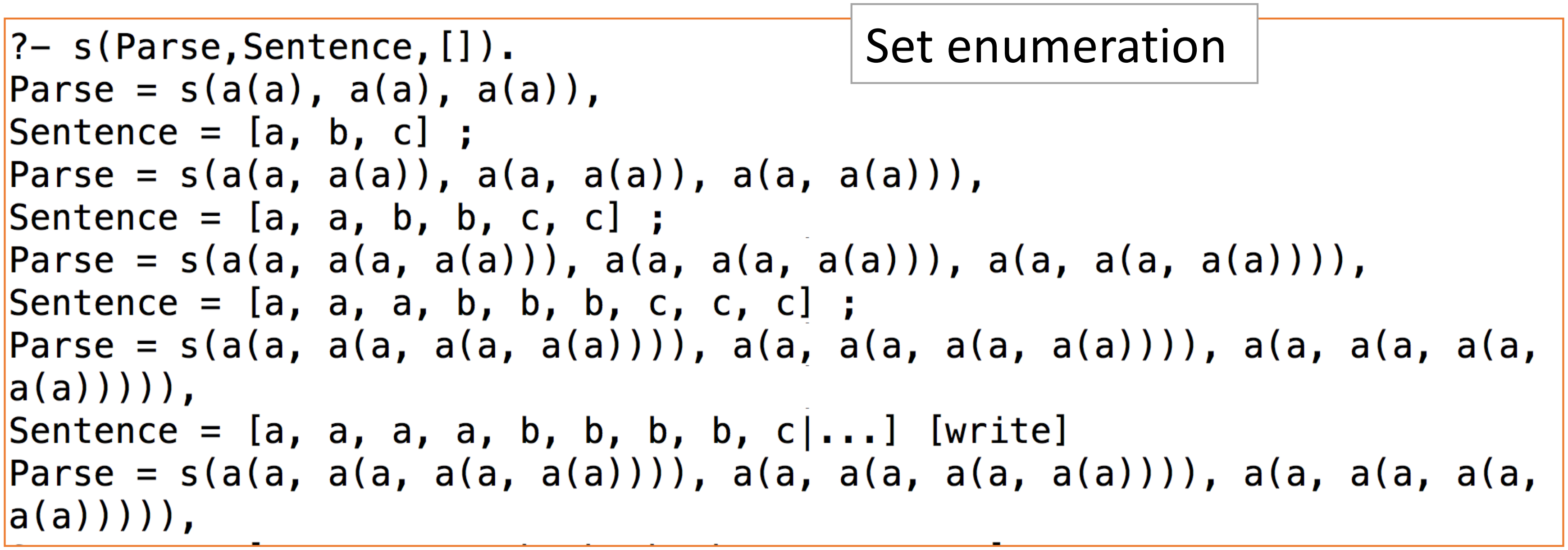


```
?- s(_,[a,a,b,b,c,c,c],[]).
false.

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

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

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

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

Set membership question

Extra arguments

- A CFG+EA grammar for $a^nb^nc^n$ n>0:



```
?- s(Parse,Sentence,[]).
Parse = s(a(a), a(a), a(a)),
Sentence = [a, b, c] ;
Parse = s(a(a, a(a)), a(a, a(a)), a(a, a(a))),
Sentence = [a, a, b, b, c, c] ;
Parse = s(a(a, a(a, a(a))), a(a, a(a, a(a))), a(a, a(a, a(a)))),
Sentence = [a, a, a, b, b, b, c, c, c] ;
Parse = s(a(a, a(a, a(a, a(a)))), a(a, a(a, a(a, a(a)))), a(a, a(a, a(a,
a(a))))),
Sentence = [a, a, a, a, b, b, b, b, c|...] [write]
Parse = s(a(a, a(a, a(a, a(a)))), a(a, a(a, a(a, a(a)))), a(a, a(a, a(a,
a(a))))),
```

Set enumeration