LING/C SC/PSYC 438/538

Lecture 23

Prof. Sandiway Fong

538 Presentations

- *see my email and 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

- **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 (*from last time*): 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)

Prolog Recursion

- Formal language example:
  - Suppose alphabet Σ={a, b}, enumerate Σ*

```
%%% Alphabet: {a, b}
sigma(a).
sigma(b).

%%% Σ*
sigmastar([]).
sigmastar([X|L]) :- sigmastar(L), sigma(X).
```

Run (hit ; for more answers)

```
?- sigmastar(L).
```




```
L = [] ;
L = [a] ;
L = [b] ;
L = [a, a] ;
L = [b, a] ;
L = [a, b] ;
L = [b, b] ;
L = [a, a, a] ;
L = [b, a, a] ;
L = [a, b, a] ;
L = [b, b, a] ;
L = [a, a, b] ;
L = [b, a, b] ;
L = [a, b, b] ;
L = [b, b, b] ;
…
```

Prolog Recursion and Non-determinism

Predicate behavior:

- (A) *succeed*, (B) *fail* or (C) *go into an infinite loop*:

- `Goal`
  - can *succeed* or *fail*
- `findall(Variable, Goal, List)`
  - accumulate values for `Variable` in Prolog `Goal` when it succeeds into a `List`
  - *always succeeds*
- `true`
  - *succeeds* (No operation) – does nothing.
- `fail`
  - *fails* – purpose is to initiate backtracking to find more answers (if they exist).

Prolog Recursion and Non-determinism

- built-in set_prolog_flag/2 to set printing behavior (*0 = print everything*):

```
?- set_prolog_flag(answer_write_options,[max_depth(0)]).
true.
```

- built-in findall/3 builds a list of solutions:

```
?- findall(X, (sigmastar(X), length(X,N), (N>5 -> !, fail ; true)), List).
List = [[], [a], [b], [a,a], [b,a], [a,b], [b,b], [a,a,a], [b,a,a], [a,b,a], [b,b,a], [a,a,b],
[b,a,b], [a,b,b], [b,b,b], [a,a,a,a], [b,a,a,a], [a,b,a,a], [b,b,a,a], [a,a,b,a], [b,a,b,a],
[a,b,b,a], [b,b,b,a], [a,a,a,b], [b,a,a,b], [a,b,a,b], [b,b,a,b], [a,a,b,b], [b,a,b,b],
[a,b,b,b], [b,b,b,b], [a,a,a,a,a], [b,a,a,a,a], [a,b,a,a,a], [b,b,a,a,a], [a,a,b,a,a],
[b,a,b,a,a], [a,b,b,a,a], [b,b,b,a,a], [a,a,a,b,a], [b,a,a,b,a], [a,b,a,b,a], [b,b,a,b,a],
[a,a,b,b,a], [b,a,b,b,a], [a,b,b,b,a], [b,b,b,b,a], [a,a,a,a,b], [b,a,a,a,b], [a,b,a,a,b],
[b,b,a,a,b], [a,a,b,a,b], [b,a,b,a,b], [a,b,b,a,b], [b,b,b,a,b], [a,a,a,b,b], [b,a,a,b,b],
[a,b,a,b,b], [b,b,a,b,b], [a,a,b,b,b], [b,a,b,b,b], [a,b,b,b,b], [b,b,b,b,b]].
```

Prolog Recursion and Non-determinism

- built-in length/2 computes the length of a list:

```
?- findall(X, (sigmastar(X), length(X,N), (N>5 -> !, fail ; true)), List), length(List, M).

List = [[], [a], [b], [a,a], [b,a], [a,b], [b,b], [a,a,a], [b,a,a], [a,b,a], [b,b,a], [a,a,b],
[b,a,b], [a,b,b], [b,b,b], [a,a,a,a], [b,a,a,a], [a,b,a,a], [b,b,a,a], [a,a,b,a], [b,a,b,a],
[a,b,b,a], [b,b,b,a], [a,a,a,b], [b,a,a,b], [a,b,a,b], [b,b,a,b], [a,a,b,b], [b,a,b,b],
[a,b,b,b], [b,b,b,b], [a,a,a,a,a], [b,a,a,a,a], [a,b,a,a,a], [b,b,a,a,a], [a,a,b,a,a],
[b,a,b,a,a], [a,b,b,a,a], [b,b,b,a,a], [a,a,a,b,a], [b,a,a,b,a], [a,b,a,b,a], [b,b,a,b,a],
[a,a,b,b,a], [b,a,b,b,a], [a,b,b,b,a], [b,b,b,b,a], [a,a,a,a,b], [b,a,a,a,b], [a,b,a,a,b],
[b,b,a,a,b], [a,a,b,a,b], [b,a,b,a,b], [a,b,b,a,b], [b,b,b,a,b], [a,a,a,b,b], [b,a,a,b,b],
[a,b,a,b,b], [b,b,a,b,b], [a,a,b,b,b], [b,a,b,b,b], [a,b,b,b,b], [b,b,b,b,b]],

M = 63.
```

Prolog Recursion and Non-determinism

- built-in set_prolog_flag/2 to set printing behavior (*10 deep*):

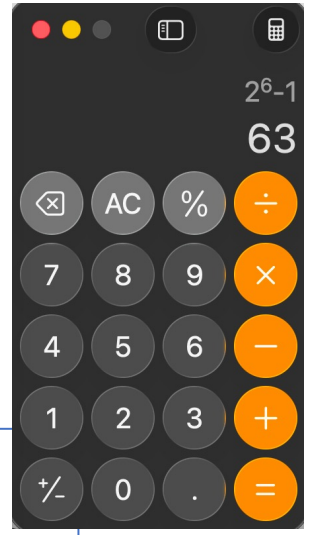
```
?- set_prolog_flag(answer_write_options,[max_depth(10)]).
true.

?- findall(X, (sigmastar(X), length(X,N), (N>5 -> !; true)), List),
length(List, M).
List = [[],[a],[b],[a,a],[b,a],[a,b],[b,b],[a|...],[...|...]|...],
M = 63.
```

Prolog Recursion and Non-determinism

Is 63 the right answer?

- L = {s | s ∈ Σ*, |s| ≤ 5}, , Σ = {a, b}
- length 0: [] (1)
- length 1: choice of either a or b (2)
- length 2: (4)
- length 3: (8)
- length 4: (16)
- length 5: (32)
- 32 + (16 + 8 + 4 + 2) + 1 = 63

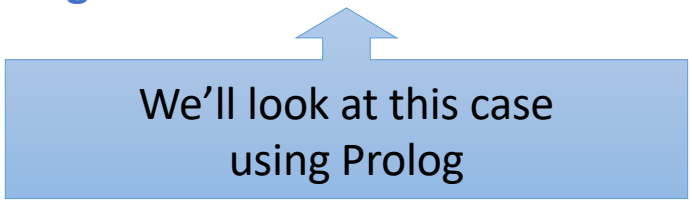


$$2^{n+1} - 1$$

Today

Regular Language family

- Three formalisms, same expressive power!
  1. Regular expressions (regex)
  2. Finite State Automata (FSA)
  3. **Regular Grammars**

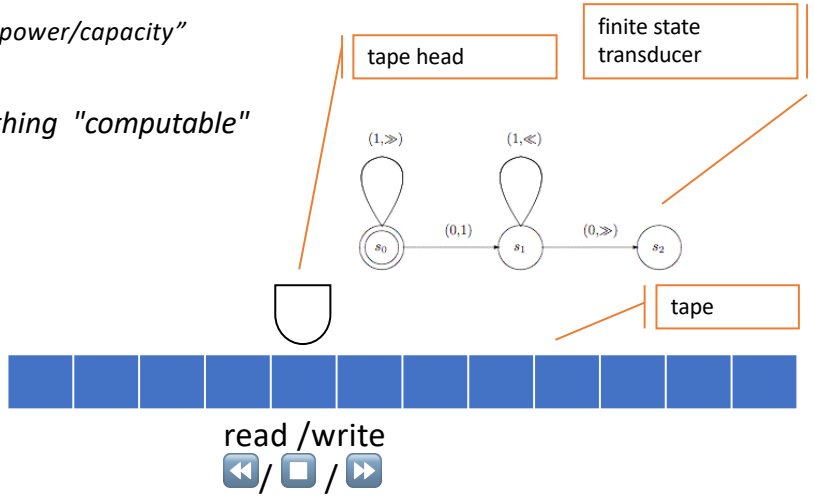


Chomsky Hierarchy

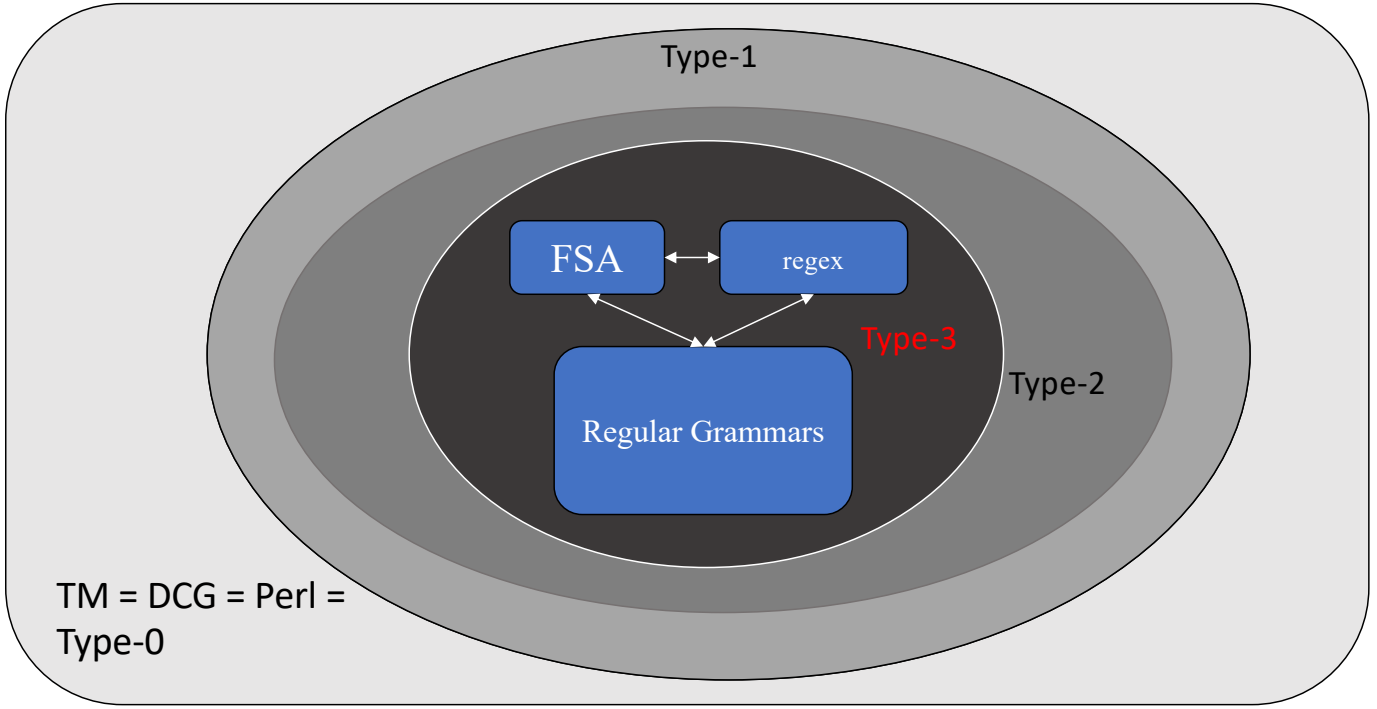
Natural languages: *do they even fit here*?

Chomsky Hierarchy

- *division of grammar into subclasses partitioned by "generative power/capacity"*
- **Type-0 General rewrite rules**
  - *Turing-complete, powerful enough to encode anything "computable"*
  - *can simulate 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
    - LHS = left hand side
    - RHS = right hand side
  - 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 → aB `s --> [a],b.`
  - B → aB `b --> [a],b.`
  - B → bC `b --> [b],c.`
  - B → b `b --> [b].`
  - C → bC `c --> [b],c.`
  - C → 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***)

  - where `x` and `y` are non-terminal symbols and
  - `t` (enclosed in square brackets) represents a terminal symbol.

Terminology: *"left/right linear"*

- **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?

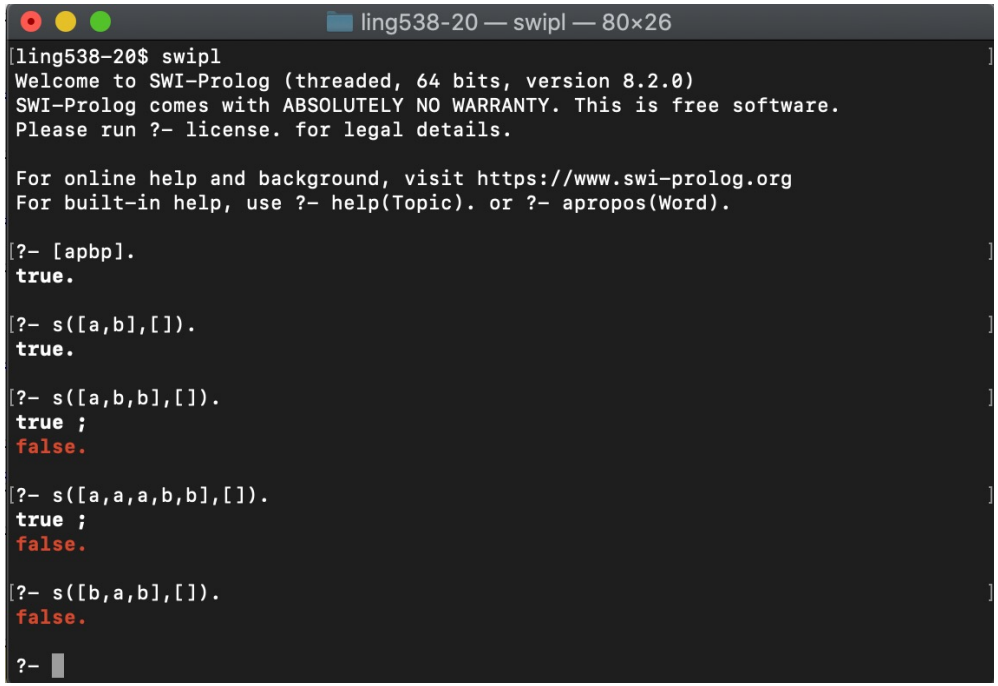
one or more a's followed by one or more b's

```
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)



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

Prolog lists revisited

- Perl lists:
  - `@list = ("a", "b", "c");`
  - `@list = qw(a b c);`
  - `@list = ();`
- Prolog lists:
  - `List = [a, b, c]`
  - `List = [a|[b|[c|[]]]]`
  - `List = []`

Python lists:

```
list = ["a", "b", "c"]

list = []
```

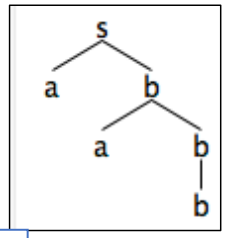
(**L**`ist` is a variable; a – c are atoms)

(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

```
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

Derivation:

s

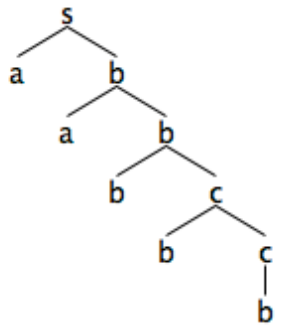
[a], **b**	(rule 1)
[a],[a],**b**	(rule 2)
[a],[a],[b]	(rule 4)

all terminals, so we stop

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 automatically translated into Prolog clauses when the program is loaded.
- 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|...]
```