Did you install SWI Prolog?

?- listing(canfly).
  canfly(A) :-
      bird(A).

  true.

?- canfly(X).
  X = tweety ;
  X = penguin.

?- [test].
  true.

?- canfly(X), \+ canfly(X).
  X = tweety ;
  false.

?- listing(cantfly).
  cantfly(penguin).

  true.
SWI Prolog Cheatsheet

• At the prompt ?-  
  1. halt.  
  2. listing.  
  3. [filename].  
  4. trace.  
  5. notrace.  
  6. debug.  
  7. nodedbug.  
  8. spy(name).  
  9. pwd.  
  10. working_directory(_, Y).  

• Anytime  
  – ^C  

Everything typed at the prompt must end in a period.

listing(name). more useful  
loads filename.pl  
(step through derivation, hit return)  

turn off debugger  
spy on predicate name  
print working directory  
switch directories to Y  

(then a(bort) or h(elp) for other options)
Prolog online resources

• Useful Online Tutorial
  – Learn Prolog Now!
    • Patrick Blackburn, Johan Bos & Kristina Striegnitz
    • [http://www.learnprolognow.org](http://www.learnprolognow.org)
Prolog Recursion

• Example (factorial):
  – 0! = 1
  – n! = n * (n-1)! for n>0

• In Prolog:
  – factorial(0,1).
  – factorial(N,NF) :- M is N-1, factorial(M,MF), NF is N * MF.

• Problem: *infinite loop*

• Fix: 2\textsuperscript{nd} case only applies to numbers > 0
  factorial(N,NF) :- \texttt{N>0}, M is N-1, factorial(M,MF), NF is N * MF.
Regular Languages

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

We’ll look at this case using a logic programming language: Prolog
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)
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)
Definite Clause Grammars (DCG)

• Example
  – Formal grammar
  – $S \rightarrow aB$
  – $B \rightarrow aB$
  – $B \rightarrow bC$
  – $B \rightarrow b$
  – $C \rightarrow bC$
  – $C \rightarrow b$

  – DCG format
  – $s --> [a],b.$
  – $b --> [a],b.$
  – $b --> [b],c.$
  – $b --> [b].$
  – $c --> [b],c.$
  – $c --> [b].$

• 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 (, : and) 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 $\rightarrow$ 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:
  $x \rightarrow y, [t].$  $x \rightarrow [t].$  *(left recursive)*
  or
  $x \rightarrow [t], y.$  $x \rightarrow [t].$  *(right recursive)*
  – where $x$ and $y$ are non-terminal symbols and
  – $t$ (enclosed in square brackets) represents a terminal symbol.

• Note:
  – can’t 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?

- 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 queries
  - `?- s([a,a,b,b,b],[ ]).` Yes
  - `?- s([a,b,a],[ ]).` No

- 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. $[a]$ $[a,b,c]$
Prolog lists

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

- Prolog lists:
  - List = [a, b, c]  \(\text{(List is a variable)}\)
  - List = [a | [b | [c | []]]]  \(\text{(a = head, tail = [b | [c | []]])}\)
  - List = []

Mixed notation:

- [a | [b,c]]
- [a,b | [c]]
Regular Grammars

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

  ![Tree diagram]

  **Derivation:**
  1. `s`  
     `[a], b` (rule 1)
  2. `b`  
     `[a], [a], b` (rule 2)
  3. `b`  
     `[a], [a], [b]` (rule 4)

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

  `true`

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

  Through backtracking it can try other choices

  `our a regular grammar`

  all terminals, so we stop
Regular Grammars

• Tree representation
  – Example
    • ?- s([a,a,b,b,b],[ ]).
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. \[ s \rightarrow [a], b. \]
2. \[ b \rightarrow [a], b. \]
3. \[ b \rightarrow [b], c. \]
4. \[ b \rightarrow [b]. \]
5. \[ c \rightarrow [b], c. \]
6. \[ c \rightarrow [b]. \]

- **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... *you’d have to program it up*
SWI Prolog

• Grammar rules are translated when the program is loaded into Prolog rules.
• Solves the mystery why we have to type two arguments with the nonterminal at the command prompt
• Recall list notation:
  – \([1\mid[2,3,4]] = [1,2,3,4]\)