Prolog Notes

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Overview

Prolog.

- Designed by Alain Colmerauer (Marseille, France).
- First appeared in 1972.
- Popularized in the 80‘ies.
  - Artificial intelligence.
  - Computational linguistics.

Key Features.

- A **declarative language**.
- A small language: few primitives.
- Uses (a subset of) **propositional logic** as primary model.

Overview

Prolog.

- ➡Designed by Alain Colmerauer (Marseille, France).
- ➡First appeared in 1972.
- ➡Popularized in the 80'ies.

"Nevertheless, my aim at that time was not to create a new programming language but to describe to the computer in natural language (French) a small world of concepts and then ask the computer questions about that world and obtain answers. We wrote an embryo of such a system and in that process the tool Prolog was developed. It was used for the analysis and the generation of French text, as well as for the deductive part needed to compute the answers to the questions."

Application Scenarios

Standalone.

- ➡Prolog is a **general-purpose language**.
- ➡Can do I/O, networking, GUI.
- ➡Web-application backend.

Embedded.

- ➡**Prolog as a library**.
- ➡"Intelligent core" of program.
  - ‣Business logic.
  - ‣Rules processor.
  - ‣Authentication / authorization rules.
- ➡E.g., *tuProlog* is a Java class library.

Logic Programming Associates Ltd

SWI Prolog

The ECLiPSe Constraint Programming System

tuProlog

and many more…

Prolog in 3 Steps

(1) Provide inference rules.

➡If ***condition***, then also ***conclusion***.

➡E.g., If "*it rains*", then "*anything outside becomes wet*."

➡E.g., If "*it barks*", then "*it is a dog*."

➡E.g., If "*it is a dog*" **and** "*it is wet*", then "*it smells*."

(2) Provide facts.

➡The "**knowledge base**."

➡ E.g., "It rains.", "Fido barks.", "Fido is outside."

(3) Query the Prolog system.

➡Provide a **goal statement**.

➡E.g., "Does Fido smell?"

Prolog in 3 Steps

(1) Provide inference rules.

➡If ***condition***, then also ***conclusion***.

➡E.g., If "*it rains*", then "*anything outside becomes wet.*"

➡E.g., If "*it barks*", then "*it is a dog.*"

➡E.g., If "*it is a dog*" **and** "*it is wet*", then "*it smells.*"

(2) Provide facts.

➡The "**knowledge base**."

➡ E.g., "It rains.", "Fic

(3) Query the Prolog

➡Provide a **goal statement**.

➡E.g., "Does Fido smell?"

True for any "it."
"It" is ***a variable***.

Tuesday, February 16, 2010

Prolog in 3 Steps

“Fido” is a specific entity.
“Fido” is *an atom*.

(1) Provide in

➡If ***condition***,

➡E.g., If “*it rai* *wet*.”

➡E.g., If “*it barks*”, then “*it is a dog*.”

➡E.g., If “*it is a dog*” **and** “*it is wet*”, then “*it smells*.”

(2) Provide facts.

➡The “**knowledge base**”

➡ E.g., “It rains.”, “Fido barks.”, “Fido is outside.”

(3) Query the Prolog system.

➡Provide a **goal statement**.

➡E.g., “Does Fido smell?”

Prolog Term

one of the following

Variables

```
X, Y, Z
Thing, Dog
```

*must begin with **capital** letter*

Atoms

```
x, y, fido
'Atom', 'an atom'
```

*must begin with **lower-case** letter or be **quoted***

Numeric Literal

```
1, 2, 3, 4, 5
0.123
200
```

integers or floating points

Structures

```
date(march ,2, 2010)
state('NC', 'Raleigh')
state(Abbrev, Capital)
```

*an **atom** followed by a **comma-separated list** of **terms** enclosed in parenthesis*

(1) Inference Rules

Describe known implications / relations.

- ➡ Axioms.
- ➡ Rules to infer new facts from known facts.
- ➡ Prolog will “search and combine” these rules to find an answer to the provided query.

If “***it*** *barks*”, then “***it*** *is a dog*.”

Such rules are expressed as **Horn Clauses**.

Horn Clause

$$\textit{conclusion} \leftarrow \textit{condition}_1 \wedge \textit{condition}_2 \ldots \wedge \textit{condition}_n$$

"**conclusion** is true if conditions 1–n are all true"

"to prove **conclusion**,
first prove conditions 1–n are all true"

Horn Clause Example

If "*it barks*", then "*it is a dog*."

Use a proper variable for "**it**".

If "***X*** *barks*", then "***X*** *is a dog*."

Formalized as Horm Clause.

dog(X) ← ***barks(X)***

Prolog Syntax:

```
dog(X) :- barks(X).
```

Prolog Clause / Predicate

Clause

conclusion(*arg_1, arg_2,…,arg_n*) :-
condition_1(*some arguments*),
…
condition_m(*some arguments*).

*each argument must be a **term***

The number of arguments ***n*** is called the **arity** of the predicate.

(2) Facts

The knowledge base.

- ➡Inference rules allow to create new facts from known facts.
- ➡Need some facts to start with.
- ➡Sometimes referred to as the "**world**" or the "universe."

"**Fido barks**.", "**Fido is outside**."

```
barks(fido).
outside(fido).
```

Facts are clauses without conditions.

(3) Queries

Reasoning about the "world."

➡Provide a **goal clause**.

➡Prolog attempts to **satisfy the goal**.

"Find something that smells."

```
?- smell(X).
X = fido.
```

"Is fido a dog?"

```
?- dog(fido).
true.
```

Alternative Definitions

Multiple definitions for a clause.

- ➡Some predicates can be inferred from multiple preconditions.
- ➡E.g., not every dogs barks; there are other ways to classify an animal as a dog.

If "***X*** *barks* ***or*** *wags the tail*", then "***X*** *is a dog*."

```
dog(X) :- barks(X).
dog(X) :- wags_tail(X).
```

Note: all clauses for a given predicate should occur in consecutive lines.

Example

- A snow day is a good day for anyone.
- Payday is a good day.
- Friday is a good day unless one works on Saturday.
- A snow day occurs when the roads are icy.
- A snow day occurs when there is heavy snowfall.
- Payday occurs if one has a job and it's the last business day of the month.

Example Facts

- Roads were icy on Monday.
- Thursday was the last business day of the month.
- Bill has a job.
- Bill works on Saturday.
- Steve does not have a job.
- It snowed heavily on Wednesday.

Another Example

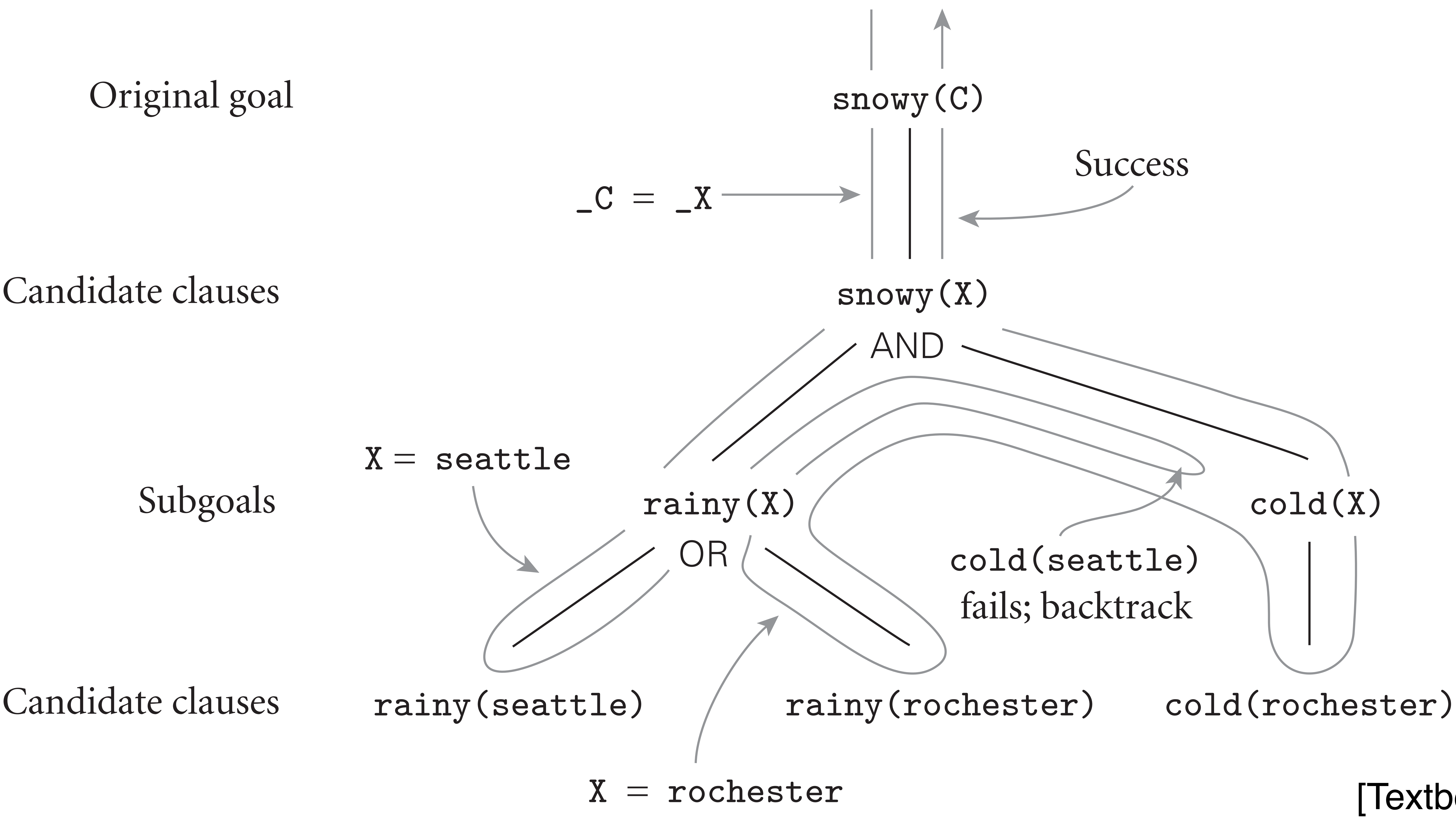
- A parent is either a father or mother.
- A grandparent is the parent of a parent.
- Two persons are sibling if they share the same father and mother (simplified model…).
- Two persons are cousins if one each of their respective parents are siblings.
- An ancestor is…?

How Prolog Works

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

Prolog tries to find an answer.

➡Depth-first tree **search** + **backtracking**.



[Textbook Figure 11.1]

Tuesday, February 16, 2010

Resolution Principle

Axiom to create proofs.

➡Robinson, 1965.

➡Formalized notion of how implications can be combined to **obtain new implications**.

➡Let's Prolog combine clauses.

$$\frac{\begin{array}{l} C \leftarrow A \wedge B \\ D \leftarrow C \end{array}}{D \leftarrow A \wedge B}$$

*"If **A** and **B** imply **C**, and **C** implies **D**, then **A** and **B** also imply **D**."*

Resolution Principle

Axiom to create proofs.

- ➡Robinson, 1965.
- ➡Formalized notion of how i
combined to **obtain new i**
- ➡Let's Prolog combine claus

$$\frac{\begin{array}{l}\textbf{barks}(\textit{fido})\\ \textbf{dog}(X) \leftarrow \textbf{barks}(X)\end{array}}{\textbf{dog}(\textit{fido}).}$$

$$\frac{\begin{array}{l}\mathbf{C} \leftarrow \mathbf{A} \wedge \mathbf{B}\\ \mathbf{D} \leftarrow \mathbf{C}\end{array}}{\mathbf{D} \leftarrow \mathbf{A} \wedge \mathbf{B}}$$

*"If **A** and **B** imply **C**, and **C** implies **D**, then **A** and **B** also imply **D**."*

Tuesday, February 16, 2010

Unification

Resolution requires "matching" clauses to be found.

- ➡ Basic question: does one **term** "match" another **term**?
- ➡ Defined by **unification**: terms "match" if they can be unified.

Unification rules.

- ➡ Two **atoms** only unify if they are identical.
  - E.g., `fido` unifies `fido` but not `'Fido'`.
- ➡ A **numeric** literal only unifies with itself.
  - E.g., `2` **does not** unify with `1 + 1`. (We'll return to this…)
- ➡ A **structure** unifies with another structure if both have the **same name**, the **same number of elements**, and each element unifies with its counterpart.
  - E.g., `date(march, 2, 2010)` does not unify `date(march, 2, 2009)`, and also not with `day(march, 2, 2010)`.

Unifying Variables

There are two kinds of variables.

- Variables cannot be updated in Prolog!
- **Unbound**: value unknown.
- **Bound**: value known.

Unification of a variable X and some term T.

- If **X** is unbound, then **X** unifies with **T** by becoming bound to **T**.
- If **X** is already bound to some term **S**, then **X** unifies with **T** only if **S** unifies with **T**.

Examples.

- **X** unbound, **T** is `fido`: unifies, **X** becomes bound to `fido`.
- **X** bound to `'NC'`, **T** is `'NC'`: unifies.
- **X** bound to `'UNC'`, **T** is `'Duke'`: never unifies.
- **X** unbound, **T** is variable **Y**: unifies, **X** becomes bound to **Y**.
- **X** bound to `'UNC'`, **T** is variable **Y**: unifies only if `'UNC'` unifies with **Y**.

Backtracking and Goal Search

Prolog “depth-first tree search” (simplified):

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw “no”
      …
      satisfy goal condM
      yield “yes” for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw “no”
```

Search fails if no answers remain.

Prolog “depth-first tree search”:

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw “no”
      …
      satisfy goal condM
      yield “yes” for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw “no”
```

Clauses are tested in source file order.

Prolog "depth-first tree search":

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw "no"
      …
      satisfy goal condM
      yield "yes" for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw "no"
```

First unify all arguments ("do they match the query terms?").

Prolog "depth-first tree search":

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1   // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1   // can throw "no"
      …
      satisfy goal condM
      yield "yes" for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw "no"
```


Tuesday, February 16, 2010

If the arguments match, then try to satisfy all conditions.

Prolog “depth-first tree search”:

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw “no”
      …
      satisfy goal condM
      yield “yes” for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw “no”
```


Tuesday, February 16, 2010

If all conditions can be satisfied, then report answer.
If there are more clauses, then search can continue.
Prolog inherently supports finding all answers!

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw “no”
      …
      satisfy goal condM
      yield “yes” for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw “no”
```

If unification fails, or if a sub goal fails, or if next answer should be found, then variable bindings have to be restored!

```
To satisfy the goal pred(T1,…,TN):
  for each clause pred(Arg1,…,ArgN) :- cond1,…,condM. :
    make snapshot of T1,…,TN
    try:
      unify T1 with Arg1  // can throw UnificationFailed
      …
      unify TN with ArgN
      satisfy goal cond1  // can throw “no”
      …
      satisfy goal condM
      yield “yes” for current T1,…,Tn // found answer!
    finally:
      restore T1,…,TN from snapshot
  throw “no”
```

Cut Operator

controlling backtracking

"Cut" branches from the search tree.

- ➡ Avoid finding "**too many**" answers.
  - E.g., answers could be symmetrical / redundant.

```
one_of(X, A, _, _) :- X = A.
one_of(X, _, B, _) :- X = B.
one_of(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, state).
true .

?- one_of(unc, duke, unc, unc).
true ;
true.
```

Cut Operator

controlling backtracking

"Cut" branches from the search tree.

➡ Avoid finding "**too many**" answers.

- E.g., answers could be symmetrical / redundant.

```
one_of(X, A, _, _) :- X = A.
one_of(X, _, B, _) :- X = B.
one_of(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, state).
true .
```

Syntax: _ is an **anonymous variable**.
(i.e., an unused argument)

Cut Operator

...king

Superfluous answer because **X** unified with both **B** and **C**.

"Cut"

➡ Avoid finding "**too many**" answers.

‣ E.g., answers could be symmetrical / redundant.

```
one_of(X, A, _, _) :- X = A.
one_of(X, _, B, _) :- X = B.
one_of(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, state).
true .

?- one_of(unc, duke, unc, unc).
true ;
true.
```

Cut Operator

controlling backtracking

```
one_of_cut(X, A, _, _) :- X = A, !.
one_of_cut(X, _, B, _) :- X = B, !.
one_of_cut(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, unc).
true.
```

The cut (!) predicate.

- Written as exclamation point.
- **Always succeeds**.
- **Side effect**: discard all previously-found backtracking points.
  - i.e., **commit to the current binding of variables**; don't restore.

Cut Operator

controlling backtracking

```
one_of_cut(X, A, _, _) :- X = A, !.
one_of_cut(X, _, B, _) :- X = B, !.
one_of_cut(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, unc).
true.
```

The cut (!) predicate.

➡ Written as exclamation point.

Meaning:
if X matches A, then **stop looking** for other answers.

Also useful for optimization.

- Prune branches that **cannot possibly contain answers**.
  - *“If we got this far, then don’t even bother looking at other clauses.”*

```
one_of_cut(X, A, _, _) :- X = A, !.
one_of_cut(X, _, B, _) :- X = B, !.
one_of_cut(X, _, _, C) :- X = C.

?- one_of(unc, duke, unc, unc).
true.
```

The cut (!) predicate.

- Written as exclamation point.
- **Always succeeds**.
- **Side effect**: discard all previously-found backtracking points.
  - i.e., **commit to the current binding of variables**; don’t restore.

Negation

Prolog negation differs from logical negation.

- Otherwise not implementable.
- **Math**: (**not X**) is true if and only if **X** is false.
- **Prolog**: (**not X**) is true if goal **X** cannot be satisfied.
  - i.e., (**not X**) is true if Prolog cannot find an answer for **X**.

SWI Syntax: `\+ X` means **not X**.

Can be defined in terms of **cut**.

```
not(X) :- call(X), !, fail.
not(X).
```

Meaning:
If you can satisfy the goal **X**,
then **don't try the other clause**, and **fail**.

- **Math**: (**not X**) is true if and only if **X** is false.
- **Prolog**: (**not X**) is true if goal **X** cannot be satisfied.
  - i.e., (**not X**) is true if Prolog cannot find an answer for **X**.

SWI Syntax: `\+ X` means **not X**.

Can be defined in terms of **cut**.

```
not(X) :- call(X), !, fail.
not(X).
```

Negation

Always succeeds, but only reached if `call(X)` fails.

➡ **Math**: (**not X**) is true if and only if **X** is false.

➡ **Prolog**: (**not X**) is true if goal **X** cannot be satisfied.

- i.e., (**not X**) is true if Prolog cannot find an answer for **X**.

SWI Syntax: `\+ X` means **not X**.

Can be defined in terms of **cut**.

```
not(X) :- call(X), !, fail.
not(X).
```


Tuesday, February 16, 2010

Closed World Assumption

Prolog assumes that the world is fully specified.

➡All facts, all rules known.

➡Thus, the **definition of negation**: anything that cannot be proven correct must be false.

➡This is the "closed world assumption."

```
ugly(worm).
pretty(X) :- \+ ugly(X).

?- pretty(ugly_dog).
true.
```

Arithmetic in Prolog

```
add(X, Y, Z) :- Z = X + Y.

?- add(1, 2, Answer).
Answer = 1+2.
```

```
add_is(X, Y, Z) :- Z is X + Y.

?- add_is(1, 2, Answer).
Answer = 3.
```

Arithmetic requires the `is` operator.

- ➡Does **not support backtracking** (E.g., **X** and **Y** must be bound).
- ➡There are too many numbers to try backtracking…
- ➡Prolog is not a computer algebra system (e.g., try Mathematica).