Announcements

• **Project 3**: Has been posted, due Friday, May 6.

• Exam on Wednesday, May 4, noon to 3:00pm (class time slot, in class).

  [Move the final exam time?]

  Final exam period: Thursday, May 5 through Wednesday, May 11.

• Sample solutions of homeworks 5 and 6 will be posted by tomorrow.

• There will be a final exam review session. Not yet clear when.

• Will teach a seminar next semester (cs671):

  **Redundancy and Approximation for Application Optimizations**
Review: Shared Memory Programming (OpenMP)

- MIMD architecture (multiple instructions, multiple data)
- Allows expression of parallelism at different levels: task and loop level. Parallelization through pragmas.
- Basic fork/join thread execution model with barrier synchronization between parallel regions.

![Shared Memory Diagram](image)

![Parallel Threads Execution Model Diagram](image)
OpenMP program example:

```c
#pragma omp parallel for private(i, hash)
    for (j = 0; j < num_hf; j++) {
        for (i = 0; i < wl_size; i++) {
            hash = hf[j] (get_word(wl, i));
            hash %= bv_size;
            bv[hash] = 1;
        }
    }
```

This specifies:

- outermost (j-loop) is parallel
- each thread will get its own copy of variables i and hash, eliminating loop carried anti and output dependences.
Distributed-Memory Programming with MPI

MPI (message passing interface)

- MIMD architecture (multiple instructions, multiple data). SPMD programming model (single program, multiple data).
- No global shared memory. Communication through explicit send/receive operations. Receives are blocking, sends may or may not be blocking.
- MPI defines an abstract processor topology (e.g.: 3-dim grid) to allow “virtual” addressing of processors (e.g.: North, South, West, East in a 2-dim grid)
Compiling for Distributed–Memory Multiprocessors

Assumptions

- Data parallelism has been specified via data layout;
  Example
  \( \text{align } A(i) \text{ with } B(i) \)
  \( \text{distribute } A(\text{block}) \text{ on 4 procs} \)

- Compiler generates SPMD code (single program, multiple data)

- Compiler uses \textit{owner–computes} rule

Compiler challenges

- Has to manage local name spaces (there is no global name space!)

- Insert necessary communication

- \textit{Goal}: Minimize cost of communication while maximizing parallelism
Compiling for Distributed-Memory Multiprocessors

Example

```fortran
real A(100), B(100)
do i = 1, 99
    A(i) = A(i+1) + B(i+1)
endo
```

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>25</th>
<th>26</th>
<th>50</th>
<th>51</th>
<th>75</th>
<th>76</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>25</td>
<td></td>
<td>26</td>
<td>50</td>
<td>51</td>
<td>75</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Proc 0</td>
<td>Proc 1</td>
<td>Proc 2</td>
<td>Proc 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```fortran
real A(26), B(26) // 1 element overlap to the right
if my$proc == 3 then my$up=24 else my$up=25 endif
if my$proc > 0 then
    send( A(1), my$proc - 1) //send to left
    send( B(1), my$proc - 1) //send to left
endif
if my$proc < 3 then
    receive( A(26), my$proc + 1) //receive from right
    receive( B(26), my$proc + 1) //receive from right
endif
do i = 1, my$up
    A(i) = A(i+1) + B(i+1)
endo
```
Host and device (GPU) programs. Program consists of parallel kernels that are executed in sequence.

GPUs have been designed for speed for graphic applications (e.g.: real-time gaming) as a graphical co-processor. Bare metal design approach.

Explicit movement of objects between host and device (GPU) memory.

GPU optimized for streaming computation with limited temporal locality

GPU implements SIMT (single instruction multiple threads) model.
Heterogenous Computing and CUDA

CUDA: Compute Unified Device Architecture

- User maps “array elements” to blocks and threads
- Blocks are units that are assigned to single Streaming Multiprocessor (SM). Multiple blocks can be assigned to the same SM. They share the same SM hardware (e.g.: hardware caches)
- Threads within a block can access the shared memory, and can synchronize. There are hardware limits on #threads per block.
- Threads are scheduled as warps (32 threads) for execution. All 32 threads execute the same instruction (SIMT, SIMD)
Parallel Programming Project

Look at Project Description
A Simple Vectorizing Compiler

How to vectorize the following loops?

for (i=2; i<100; i++) {
    S1:  a[i] = b[i+1] + 1;
    S2:  b[i] = a[i] + 5;
}

for (i=2; i<100; i++) {
    S1:  a[i] = b[i-1] + a[i-1] + 3;
    S2:  b[i] = a[i+1] + 5;
}

Simple vectorizer assumptions:

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array variables
4. simple array index expressions of induction variable (i +/- c or c); can use ZIV or SIV test
5. no function calls
A Simple Vectorizing Source-to-Source Compiler

SKETCH OF BASIC ALGORITHM

Here is a basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more

2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do

   (a) if SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential

   (b) if SCC is a single statement and has no dependence cycle, distribute loop around it and generate vector code; otherwise, mark distributed loop sequential.
Predicate Logic – first order logic

Syntax — Well formed formulae (wff):

- **term**: constant symbols, variables, and n-place function symbols followed by n *terms* enclosed in parenthesis

- **atomic formula**: n-place predicate symbols followed by n *terms* enclosed in parenthesis

- **wff**: *atomic formulae* and expressions of the form $(\alpha \rightarrow \beta), (\alpha \lor \beta), (\alpha \land \beta), (\neg \alpha), \forall x \alpha, \text{ or } \exists x \alpha$, where $\alpha$ and $\beta$ are *wffs* and $x$ is a variable

Examples:

\[ \forall x (p(x) \rightarrow q(x)) \]
\[ \forall x \exists y (p(x) \lor q(f(y))) \]

A *wff* is **closed** if it does not contain any free variables. To simplify the discussion, we will assume that all *wffs* are closed.
**Predicate Logic** – first order logic

**Semantics** — Interpretations:

An **interpretation** $I$ consists of

- a non-empty set $D$, called the universe
- a mapping that assigns to each constant $c$ a fixed element $c^I \in D$
- a mapping that assigns to each $n$-place function symbol $f$ an $n$-ary function $f^I : D^n \to D$
- a mapping that assigns to each $n$-place predicate symbol $p$ an $n$-ary predicate $p^I : D^n \to \{\text{true, false}\}$

**Note:**

- logic connectives $\rightarrow, \lor, \land, \neg$ have their usual propositional meaning
- $\forall x$ and $\exists x$ mean “for all $x$ in $D$” and “there exists an $x$ in $D$”, respectively
Predicate Logic – first order logic

An interpretation $I$ satisfies wff $\alpha$ ($\models_I \alpha$) iff $\alpha$ evaluates to true under the interpretation $I$. Such an $I$ is called a model of $\alpha$.

A wff $\alpha$ is valid ($\models \alpha$) iff it is satisfied for all of its interpretations.

A wff $\alpha$ is unsatisfiable iff it is not satisfiable in any of its interpretations.

A set $\Gamma$ of wffs logically implies a wff $\alpha$ ($\Gamma \models \alpha$) iff for every interpretation that satisfies all members in $\Gamma$, $\alpha$ is also satisfied.

**Question:** Is there a mechanical way to derive all wffs that are logically implied by a set of wffs $\Gamma$?
A deductive calculus consists of

- A set $\Delta$ of wffs, called logical axioms
- A set of inference rules

A wff $\alpha$ is a theorem of a set $\Gamma$ of wffs ($\Gamma \vdash \alpha$) iff $\alpha$ belongs to the set of formulae that can be generated from $\Gamma \cup \Delta$ by a finite sequence of inference rule applications. Such a sequence is called a deduction.

Without going into details about its infinite set $\Delta$, there is a deductive calculus that uses only a single inference rule, called modus ponens:

$$
\alpha, \alpha \rightarrow \beta \\
\underline{\alpha} \\
\beta
$$

Example:
Assume that $\Delta$ contains $\forall x p(x) \rightarrow p(a)$. Then we can proof

$$\{\forall x p(x), (p(a) \rightarrow q(a))\} \vdash q(a)$$

with two applications of the inference rule.
Two major properties of a deductive calculus:

- A deductive calculus is **sound** iff
  \[ \Gamma \vdash \alpha \text{ implies } \Gamma \models \alpha \]

- A deductive calculus is **complete** iff
  \[ \Gamma \models \alpha \text{ implies } \Gamma \vdash \alpha \]

In other words, **soundness** means “whatever can be proven is valid”, and **completeness** means “whatever is valid can be proven”. A calculus that is not sound is pretty useless. A calculus that is sound and complete is the best we can do.

Gödel showed in 1930 that first order logic is complete, i.e., there is a sound deductive calculus for first order logic that is complete.

Questions:

- Is propositional logic decidable?

- Is \( \Gamma \vdash \alpha \) decidable?

- Is second order logic (allows quantification over predicates) complete?
Logic Programming and Prolog

Logic programming languages are not procedural or functional (Scott Chap. 11).

- Specify *relations* between objects
  - larger(3,2)
  - father(tom,jane)

- Separate logic from control:
  - Programmer declares **what** facts and relations are true
  - System determines **how** to use facts to solve problems

- Based on Predicate Logic (first order logic)

- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
Free and bound variables

A variable $x$ can occur in a wff

- within a term (use), or
- next to a quantifier $\forall$ or $\exists$ (definition).

Any use of a variable is bound to the “closest” surrounding definition of the variable, if such a definition exists.

For the wffs $\forall x \alpha$ or $\exists x \alpha$, $\alpha$ is called the scope of definition $x$.

Example:

$$(\forall x(p(x) \rightarrow \exists y \exists x(q(x) \lor q(y))) \lor x)$$

Question: What are the bindings between the uses and definitions of variables? Note: the same variable can be defined or used several times in a wff.

The use of a variable is free in a wff if there is no matching definition. Otherwise it is bound.
Prolog and predicate logic

Prolog is based on Horn clauses (clauses). Formulae have the form:

$$\forall X( (male(X) \land parent(X, jane)) \rightarrow father(X, jane))$$

written in Prolog as

$$father(X, jane) : - male(X), parent(X, jane).$$

In general,

$$A : - B_1, B_2, \ldots, B_n.$$ 

where $A, B_1, \ldots, B_n$ are atomic formulae and $n \geq 0$. $A$ is called the conclusion and the $B_i$’s are called subgoals or conditions.

Variables can occur in the atomic formulae. All variables are implicitly $\forall$-quantified and therefore there are no free variables. Note: Scope of variables is restricted to a single clause.
\[ A : - B_1, B_2, \ldots, B_n. \]

Three different instantiations:

- **rule** — conclusion (head) and conditions (tail) are non-empty

- **facts** or **assertions** — conditions are empty.  
  Example: father(tom, jane).

- **query** or **goal** — conclusion is empty.  
  Example: ?- father(X, jane).

In Prolog:

- All variables are capitalized
- All constants are in lower case
- All predicates are in lower case
Basic Facts and Relations

Use prolog command (SWI-Prolog) on ilab cluster to start prolog interpreter. Exit using \texttt{CtrlD}. A Prolog program starts with declarations of the basic facts.

\begin{verbatim}
male(albert). \texttt{---------------------------------- a fact}
female(alice). \texttt{Facts are put in file `family.pl''}
male(Edward).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
parent(albert,alice).
parent(victoria,alice).
\end{verbatim}

\begin{verbatim}
> prolog
| ?- [family]. \texttt{---------------------------------- loads file}
yes
| ?- male(albert). \texttt{---------------------------------- a query}
yes
| ?- male(alice).
no
| ?- parent(albert,edward).
yes
| ?- parent(bullwinkle,edward).
no
\end{verbatim}

Limited use: need variables and deductive rules.
Variables and Unification

| ?- female(X).
X = alice
| ?- female(X).
X = alice ; <---------- ';' means look further
X = victoria ;
no

X is **unified** to all possible values that make the query female(X) true.

| ?- parent(P,edward).
P = albert ;
P = victoria ;
no

P is unified to all possible values that make the query parent(P,edward) true.

⇒ search with pattern matching
Prolog Horn Clause Examples

A Horn clause with no tail:

```
male(albert).
```

I.e., a fact: albert is a male dependent on no other conditions

A Horn clause with a tail:

```
father(albert, edward):-
    male(albert), parent(albert, edward).
```

I.e., a rule: albert is the father of edward if albert is male and albert is a parent of edward’s.
Horn Clauses with Variables

Variables may appear in the head **and** tail of a Horn clause:

- $c(X_1, \ldots, X_n) : h(X_1, \ldots, X_n)$.
  
  “For all values of $X_1, \ldots, X_n$, the formula $c(X_1, \ldots, X_n)$ is true if the formula $h(X_1, \ldots, X_n)$ is true”

- $c(X_1, \ldots, X_n) : h(X_1, \ldots, X_n, Y_1, \ldots, Y_k)$.
  
  “For all values of $X_1, \ldots, X_n$, the formula $c(X_1, \ldots, X_n)$ is true if there exist values of $Y_1, \ldots, Y_k$ such that the formula $h(X_1, \ldots, X_n, Y_1, \ldots, Y_k)$ is true”

Example from logic:

$$\forall X (p(a) \leftarrow q(X))$$

is logically equivalent to

$$(p(a) \leftarrow \exists X q(X))$$
Examples

father(X,Y):- male(X), parent(X,Y).
(X is the father of Y if X is male and X is a parent of Y)

| ?- father(F,edward).
F = albert ;
no

child_of(C,P):- parent(P,C).

| ?- child_of(C,P).
C = edward
P = albert ;
C = edward
P = victoria ;
C = alice
P = albert ;
C = alice
P = victoria ;
no
Examples

\[
\text{sibling}(X,Y):- \text{parent}(P,X), \text{parent}(P,Y).
\]

\[
| \text{?- sibling(alice,A).} \\
A = \text{edward} ; \\
A = \text{alice} ; \\
A = \text{edward} ; \\
A = \text{alice} ; \\
\text{no}
\]
Rule Ordering and Unification

1. rule ordering used in search

2. unification requires two instances of the same variable to get the same value

3. unification does not require differently named variables to get different values: hence, sibling(edward,edward).

4. all rules searched if requested by ’;’
Prolog and predicate logic

Horn clauses allow efficient implementation of “backward” deductions through \textit{backward chaining}.

Predicate logic is \textit{declarative}. Semantics is independent of \textit{HOW} to prove theorems. There is no control information other than implied by logical inference. Control is an efficiency issue, not a correctness issue.

Prolog has a \textit{procedural} interpretation with a \textit{declarative} flavor. Goal: Separate control component of program (\textit{HOW}) from description of desired outcome (\textit{WHAT}).

\[
\text{algorithm} = \text{logic} + \text{control}
\]

Prolog: Control is part of the semantics
Things we will talk about:

- subgoal and rule selection
- cuts
- negation as failure
male(albert).
female(alice).
male(edward).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
parent(albert,alice).
parent(victoria,alice).
sibling(X,Y):- parent(P,X), parent(P,Y).

?- sibling(alice, A)

Prolog Search Tree
Control in Prolog

- **goal rule** — choose the leftmost subgoal
- **rule order** — select the first applicable rule

Prolog uses a depth-first search of the tree. This can be "dangerous" if the tree has infinite subtrees.

Replacement of a goal by the subgoals of right-hand side of rule is called *backward chaining*.
Prolog – procedural interpretation

\[ A \, : \, - \, B_1, B_2, \ldots, B_n. \]

- Interpret each clause as a procedure definition: conclusion is procedure name or head, and conditions are the body.
- Main program is procedure body with no name (original query).
- To execute the body of a procedure \( A \), call each procedure \( B_1, B_2, \ldots, B_n \).
- Procedures are invoked by unification, a generalized pattern match. Find the most general unifier (mgu) of the selected call (subgoal) and the head of the selected clause (rule).
- Execution terminates when all procedures have successfully terminated.
- A procedure with no body terminates as soon as the unifying substitution is made.
Quick Review

- Prolog has a **procedural** interpretation with a **declarative** flavor.

- Control in Prolog
  - **goal rule** — choose the leftmost subgoal
  - **rule order** — select the first applicable rule

- Replacement of a goal by the subgoals of right-hand side of rule is called **backward chaining**.

- Prolog uses a depth-first search over the **Prolog Search Tree**. This can lead to non-termination if the tree has infinite subtrees.

- A **unifier** is a substitution of variables by terms such that, when applied to both the call and the head of the selected rule, makes them **syntactically identical**. The **most general unifier** (mgu) makes as few and as general bindings as possible.

- Computation builds up a composition of unifiers. If program terminates successfully, the bindings of the variables in the initial query are the output of the program.
Prolog: unification

A unifier is a substitution of variables by terms such that, when applied to both the call and the head of the selected rule, makes them syntactically identical. The most general unifier (mgu) makes as few and as general bindings as possible.

Note: “=” is the unification operator in Prolog

Example:

times(Z, times(Y, 7) ) = times(4, W)

\[ \Theta_1 = \{ Z \rightarrow 4, Y \rightarrow plus(3, 5), W \rightarrow times(plus(3, 5), 7) \} \]

\[ \Theta_2 = \{ Z \rightarrow 4, W \rightarrow times(Y, 7) \} \]

Question: Which of the two substitutions is more general?

Note: mgu \( \theta \) is unique modulo renaming of variables
Unification — formal definition

- A substitution $\sigma$ is a mapping from variables to terms.
  The result of the substitution $T\sigma$ is recursively defined ("=" means textually identical):
  
  $- X\sigma = U$ if $X \rightarrow U$ is in $\sigma$
  
  $- X\sigma = X$ otherwise
  
  $- (f(T_1, T_2))\sigma = f(U_1, U_2)$ if $T_1\sigma = U_1$ and $T_2\sigma = U_2$

  Assumption: if $\sigma$ maps $X$ to $T$, then $X$ does not occur in $T$ (occurs check → will discuss later).

- A term $U$ is an instance of $T$ if $U = T\sigma$ for some substitution $\sigma$.

- Two terms $T_1$ and $T_2$ unify if $T_1\sigma = T_2\sigma$ for some substitution $\sigma$. $\sigma$ is called a unifier of $T_1$ and $T_2$.

- The substitution $\theta$ is the most general unifier (mgu) of $T_1$ and $T_2$ if $\forall$ other unifiers $\sigma'$, $T_1\sigma'$ is an instance of $T_1\theta$. 
Prolog: backward-chaining and unification

Example list of subgoals:

\[ : - A_1, \ldots, A_i, \ldots A_m. \quad m \geq 0 \]

Invoke procedure \( A_i \) with rule \( A : - B_1, \ldots B_n. \)

Unify \( A_i \) with \( A \), i.e., \( A_i \Theta = A \Theta \). \( \Theta \) maps variables to terms. The resulting list of subgoals is:

\[ : - (A_1, \ldots, A_{i-1}, B_1, \ldots B_n, A_{i+1} \ldots A_m) \Theta. \]

Note: Unification performs the basic data manipulation operations of

- assignment
- parameter passing
- data selection
- data construction

Note:
Computation builds up a composition of unifiers. If program terminates successfully, the bindings of the variables in the initial query are the output of the program.
Prolog: backward-chaining computation

1. start with initial query
2. Select a call to execute
3. Select a procedure to use in executing the chosen call
4. Standardize — rename variables to ensure that there are no variables that occur both in the current set of calls and in the selected procedure (renaming)
5. Find \textit{mgu} of the selected call and the name of the selected procedure
6. Replace the selected call by the body of the procedure
7. Apply the \textit{mgu} to the new set of calls resulting from previous step
8. If no calls remain, terminate successfully; If no procedure name found to match the selected call, back up and redo the previous call, using a different procedure than the one already used; if all backtrack options are exhausted, terminate with failure
Order of Arguments, Rules, and Subgoals

?- sibling(A,alice).
A = edward ;
A = edward ;
A = alice ;
A = alice ;
no

Note: arguments are interchangeable, but ordering affects order of search.

In General:
Order of subgoals and rules may make a difference in terms of efficiency.

• Order may determine whether resolution terminates or not.

• You can write inefficient logic. Example:
  sort(x,y) :- permut(x,y), ordered(y)
Prolog syntax (subset) in EBNF

\[
<\text{clause} > ::= \quad <\text{atmf} > \cdot | \\
\quad <\text{atmf} > \leftarrow <\text{atmfs} > \cdot | \\
\quad ?- <\text{atmfs} > .
\]

\[
<\text{atmfs} > ::= \quad <\text{atmf} > \{, <\text{atmf} > \}
\]

\[
<\text{atmf} > ::= \quad <\text{pred} > (<\text{terms} >)
\]

\[
<\text{terms} > ::= \quad <\text{term} > \{, <\text{term} > \}
\]

\[
<\text{term} > ::= \quad <\text{variable} > | <\text{constant} > | \\
\quad <\text{func} > (<\text{terms} >) | <\text{list} >
\]

\[
<\text{variable} > ::= <\text{uppercase\_word} >
\]

\[
<\text{constant} > ::= <\text{lowercase\_word} > | \text{number}
\]

\[
<\text{pred} > ::= <\text{lowercase\_word} >
\]

\[
<\text{func} > ::= <\text{lowercase\_word} >
\]

\[
<\text{list} > ::= [ ] | [ <\text{head} > | <\text{tail} > ]
\]

\[
<\text{head} > ::= <\text{term} >
\]

\[
<\text{tail} > ::= <\text{list} >
\]
Lists in Prolog

[a, b] is an abbreviation for [a | [b | []]]

<table>
<thead>
<tr>
<th></th>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b, c]</td>
<td>a</td>
<td>[b, c]</td>
</tr>
<tr>
<td>[X, [foo], Y]</td>
<td>X</td>
<td>[[foo], Y]</td>
</tr>
<tr>
<td>[a, [b, c], d]</td>
<td>a</td>
<td>[[b, c], d]</td>
</tr>
</tbody>
</table>

[211]

[1 | 2] [1, 2]

1 2

1

2 [ ]
More examples: Lists in Prolog

Can you unify the following lists?

\[ [X, \text{abc}, Y] \quad \text{and} \quad [X, \text{abc} \mid Y] ? \]

\[ [a, b \mid Z] \quad \text{and} \quad [X \mid Y] ? \]

\[ [a, b, c] \quad \text{and} \quad [X \mid Y] ? \]
Operations on Lists in Prolog

\[\text{member}(A, [A \mid B]).\]
\[\text{member}(A, [B \mid C]) \leftarrow \text{member}(A, C).\]

?– member(a, [a,b]).

?– member(a, [b,c]).

?– member(X, [a,b,c]).

?– member(a, [b,c,X]).

?– X = [1,2,3], member(a,X).

?– member(a,X), X = [1,2,3].
member Query: Prolog Search Tree

member(A, [A | B]).
member(A, [B | C]) :- member(A, C).

?- X = [1,2,3], member(a,X).
member – Using Placeholders

Another way or writing the *member* predicate:

member(A, [A | _]).
member(A, [ _ | C]) :- member(A, C).

?- member(a,X), X = [1,2,3].

Leads to infinite computation.

\[
X = [a|_] ; \\
X = [_, a|_] ; \\
X = [_, _, a|_] ;
\]

\(\_\) is a special Prolog variable; a placeholder (fresh variable) is generated by the Prolog system for each occurrence of "\_".

"\_" is a special Prolog variable; a placeholder (fresh variable) is generated by the Prolog system for each occurrence of \"\_\".
Arithmetic in Prolog

?- AGE is 1995 - 1956.
AGE = 39 ;
no

AGE = 1995-1956;
no

no

?- 39 is DATE - 1956.
! Instantiation error in argument 2 of is/2
! goal: 39 is _6668-1956

At the time Prolog begins processing a goal of the form: X is <Exp>, <Exp> must be a fully instantiated arithmetic expression, i.e., may not contain any variables after unification.

⇒ Arithmetic programs are not always invertible.

?- AGE is DATE - 1956, DATE is 1995.
?- DATE is 1995, AGE is DATE - 1956.
factorial(0,1).

factorial(X,Y) :- W is X-1,
               factorial(W,Z),
               Y is Z*X.

This calculates X! if X is bound to an integer. Otherwise it aborts in the first “is” clause.

?- factorial(3,6).
yes

?- factorial(5,Z).
Z = 120

?- factorial(Y,6).
! Instantiation error in argument 2 of is/2
! goal: _6568 is _6571-1
Prolog – Cuts

• A cut (denoted !) prunes parts of a Prolog search tree by restricting backtracking.

• Cuts are control features, i.e., make Prolog depart even further from “declarative logic”.

• Cut can occur on right-hand side of rules:
  \[ B : - C_1, \ldots C_{j-1}, !, C_{j+1}, \ldots C_k \] means
  Backtrack past \( C_{j-1}, \ldots C_1, B \) without considering any remaining, alternative rules for them.

• The cut is a subgoal that always succeeds.

• Optimize time and space of the computation.

• Green cuts: Prune parts of search tree that cannot contain any successful solutions
  
  Red cuts: All others
Cuts — examples

Examples:

conclusion(S) :- guess(S), !, verify(S)

Eliminates all but the first successful guess
Note: if guess(S) fails, another rule for conclusion(S) can be selected.
Negation as failure

\[
\text{not}(X) :- \ X, !, \text{fail}.
\]
\[
\text{not}(\_).
\]

If \textbf{X succeeds} in first rule, then the rule is forced to fail by the last term. We cannot backtrack over the cut in the first rule and the cut prevents us from accessing the second rule.

If \textbf{X fails} in first rule, then the second rule succeeds, since \_ unifies with anything.

Some interesting behavior:

\begin{verbatim}
?- X = 2, not(X = 1).
X=2
?- not(X = 1), X = 2
no
\end{verbatim}

Note

- The implications of negation as failure are very complex. Be careful.
Horn clause logic is a proper subset of predicate logic in terms of syntax and semantics. The following is not a Prolog program:

\[ Q(a) \lor Q(b). \]
\[ ? - Q(X). \]

In predicate logic, the answer should be YES, since \( Q(a) \lor Q(b) \models \exists x Q(x) \). However, we don’t know what the substitution for \( X \) should be.

In Horn clause logic, we can always compute definite answers through resolution and unification:

If \( P \) is a Prolog program, then

\[ P \models \exists Q(x) \] implies that there is a unifier \( \theta \) such that

\[ P \models Q(x)\theta. \]

This means that we actually can compute answers, i.e., substitutions for variables in the input query. For the example above, we don’t know whether \( \theta = \{ X \rightarrow a \} \) or \( \theta = \{ X \rightarrow b \} \) is a correct answer.
That’s it!

Hope you enjoyed the class.

Good luck with the third project and the final exam.