Computational Logic
Efficiency Issues in Prolog
Efficiency

- In general, efficiency ≡ savings:
  - Not only time
    (number of unifications, reduction steps, LIPS, etc.)
  - Also memory

- General advice:
  - Use the best algorithms
  - Use the appropriate data structures

- Each programming paradigm has its specific techniques, try not to adopt them blindly.

Note: The timings in the following examples were taken a long time ago, so computers and Prolog are much faster now, but the comparisons are still valid!
Data structures

- Do not make excessive use of lists:
  - In general, only when the number of elements is unknown
  - It is convenient to keep them ordered sometimes (e.g., set equality)
  - Otherwise, use structures (functors):
    * Less memory
    * Direct access to each argument \((\text{arg}/3)\) (like arrays!)

\[
\begin{align*}
[a, b, c] & \rightarrow \text{LST} & a \rightarrow \text{LST} & b \rightarrow \text{LST} & c \rightarrow [] \\
\text{f(a, b, c)} & \rightarrow \text{STR} & \text{f/3} & a \rightarrow b \rightarrow c
\end{align*}
\]
Data structures (Contd.)

- Use advanced data structures:
  - Sorted trees
  - Incomplete structures
  - Nested structures
  - ...
- D.H.D. Warren: “Prolog means easy pointers”
Let Unification Do the Work

- Unification is very powerful. Use it!

- Example: Swapping two elements of a structure:
  \[ f(X, Y) \sim f(Y, X) \]
  - Slow, difficult to understand, long version (exaggerated):

  ```prolog
  swap(S1, S2):-
      functor(S1, f, 2), functor(S2, f, 2),
      arg(1, S1, X1), arg(2, S1, Y1),
      arg(1, S2, X2), arg(2, S2, Y2),
      X1 = Y2, X2 = Y1.
  ```

  - Fast, intuitive, shorter version:

  ```prolog
  swap(f(X, Y), f(Y, X)).
  ```
Example: check that a list has exactly three elements.

- Weak answer:
  
  ```prolog```
  ```
  three_elements(L):-
    length(L, N), N = 3.
  ```
  
  (always traverses the list and computes its length)

- Better:
  
  ```prolog```
  ```
  three_elements([_,_,_]).
  ```
Avoid using it for simulating global variables

Example (real executions):

```prolog
bad_count(N):-
    assert(counting(N)),
    even_worse.

even_worse:-
    retract(counting(0)).
even_worse:-
    retract(counting(N)),
    N > 0, N1 is N - 1,
    assert(counting(N1)),
    even_worse.

good_count(0).
good_count(N):-
    N > 0, N1 is N - 1,
    good_count(N1).
```

bad_count(10000): 165,000 bytes, 7.2 sec.
good_count(10000): 1,500 bytes, 0.01 sec.
- Asserting results which have been found true (lemmas).

Example (real executions):

```prolog
fib(0, 0).
fib(1, 1).
fib(N, F):-
    N > 1,
    N1 is N - 1,
    N2 is N1 - 1,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

```prolog
:-- dynamic lemma_fib/2.
lemma_fib(0, 0).
lemma_fib(1, 1).

lfib(N, F):-
    lemma_fib(N, F), !.
lfib(N, F):-
    N > 1,
    N1 is N - 1,
    N2 is N1 - 1,
    lfib(N1, F1),
    lfib(N2, F2),
    F is F1 + F2,
    assert(lemma_fib(N, F)).
```

```prolog
fib(24, F): 4,800,000 bytes, 0.72 sec.
lfib(24, F): 3,900 bytes, 0.02 sec. (and zero if called again)
```

Warning: only useful when intermediate results are reused.
Determinism (I)

- Many problems are deterministic.
- Non-determinism is
  - Useful (automatic search).
  - But expensive.
- Suggestions:
  - Do not keep alternatives if they are not needed.

\[
\text{member_check}([X|\_],X) \leftarrow !. \\
\text{member_check}([\_|Xs],X) \leftarrow \text{member_check}(Xs,X).
\]

- Program deterministic problems in a deterministic way:

Simplistic:
\[
\text{decomp}(N, S1, S2) \leftarrow \\
\text{between}(0, N, S1), \\
\text{between}(0, N, S2), \\
N =:= S1 + S2.
\]

Better:
\[
\text{decomp}(N, S1, S2) \leftarrow \\
\text{between}(0, N, S1), \\
S2 \text{ is } N - S1.
\]
Determinism (II)

- Checking that two (ground) lists contain the same elements

- Naive:

```prolog
same_elements(L1, L2):-
    \+ (member(X, L1), \+ member(X, L2)),
    \+ (member(X, L2), \+ member(X, L1)).
```

- 1000 elements: 7.1 secs.

- Sort and unify:

```prolog
same_elements(L1, L2):-
    sort(L1, Sorted),
    sort(L2, Sorted).
```

(sorrying can be done in \(O(N \log N)\))

- 1000 elements: 0 secs.
Search order

- Golden rule: fail as early as possible (prunes branches)
- How: reorder goals in the body (perhaps even dynamically)
- Example: generate and test

\[
\text{generate}_z(Z) :\leftarrow \\
\quad \text{generate}_x(X), \\
\quad \text{generate}_y(X,Y), \\
\quad \text{test}_x(X), \\
\quad \text{test}_y(Y), \\
\quad \text{combine}(X,Y,Z).
\]

- Perform tests as soon as possible:

\[
\text{generate}_z(Z) :\leftarrow \\
\quad \text{generate}_x(X), \\
\quad \text{test}_x(X), \\
\quad \text{generate}_y(X,Y), \\
\quad \text{test}_y(Y), \\
\quad \text{combine}(X,Y,Z).
\]

- Even better: test as deeply as possible within the generator

\[
\text{generate}_z(Z) :\leftarrow \\
\quad \text{generate}_x\_test(X), \\
\quad \text{generate}_y\_test(X,Y), \\
\quad \text{combine}(X,Y,Z).
\]

→ c.f. Constraint Logic Programming!
Indexing

- Indexing on the first argument:
  - At compile time an indexing table is built for each predicate based on the principal functor of the first argument of the clause heads
  - At run-time only the clauses with a compatible functor in the first argument are considered

- Result: appropriate clauses are reached faster and choice-points are not created if there are no “eligible” clauses left

- Improves the ability to detect determinacy, important for preserving working storage
Indexing (Contd.)

- Example: value greater than all elements in list

```
bad_greater(_X, []).  
bad_greater(_X, [_Y|_Ys]):- X > Y, bad_greater(X, _Ys).  
```

600,000 elements: 2.3 sec.

```
good_greater([],_X).  
good_greater(_X, _Y|_Ys) :- X > Y, good_greater(_Ys, X).  
```

600,000 elements: 0.67 sec

- Can be used with structures other than lists

- Available in most Prolog systems
Iteration vs. Recursion

- When the recursive call is the last subgoal in the clause and there are no alternatives left in the execution of the predicate, we have an *iteration*

- Much more efficient

- Example:

  ```prolog
  sum([], 0).
  sum([N|Ns], Sum):-
    sum(Ns, Inter),
    Sum is Inter + N.
  ```

  ```prolog
  sum_iter(L, Res):-
    sum(L, 0, Res).
  sum([N|Ns], In, Out):-
    Inter is In + N,
    sum(Ns, Inter, Out).
  ```

  - `sum/2` 100000 elements: 0.45 sec.
  - `sum_iter/2` 100000 elements: 0.12 sec.
Iteration vs. Recursion (Contd.)

• The basic skeleton is:

```prolog
<head> :-
  <deterministic computation>
  <recursive_call>.
```

• Known as tail recursion

• Particular case of last call optimization

• It also consumes less memory
Cuts

- Cuts eliminate choice-points, so they “create” determinism
- Example:

```prolog
a :-
    test_1, !, ...
    a :-
    test_2, !, ...
    ...
    a :-
    test_n, !, ...
```

- If $test_1 \ldots test_n$ mutually exclusive, declarative meaning of program not affected.
- Otherwise, be careful: Declarativeness, Readability.
Delaying Work

- Do not perform useless operations
- In general:
  - Do not do anything until necessary
  - Put the tests as soon as possible

- Example:

  ```prolog
  x2x3([], []).
  x2x3([X|Xs], [NX|NXs]):-
    NX is -X * 2,
    X < 0,
    x2x3(Xs, NXs).
  x2x3([X|Xs], [NX|NXs]):-
    NX is X * 3,
    X >= 0,
    x2x3(Xs, NXs).
  ```

  100,000 elements: 1.05 sec.

- Delaying the arithmetic operations

  ```prolog
  x2x3_1([], []).
  x2x3_1([X|Xs], [NX|NXs]):-
    X < 0,
    NX is -X * 2,
    x2x3_1(Xs, NXs).
  x2x3_1([X|Xs], [NX|NXs]):-
    X >= 0,
    NX is X * 3,
    x2x3_1(Xs, NXs).
  ```

  100,000 elements: 0.9 sec.
Delaying Work

- Delaying head unification + determinism:

```prolog
x2x3_2([], []).
x2x3_2([X|Xs], Out):-
    X < 0, !,
    NX is -X * 2,
    Out = [NX|NXs],
    x2x3_2(Xs, NXs).
```

```prolog
x2x3_2([X|Xs], Out):-
    X >= 0, !,
    NX is X * 3,
    Out = [NX|NXs],
    x2x3_2(Xs, NXs).
```

100000 elements: 0.68 sec. (and half the memory consumption)

- Some (personal) advice: use these techniques only when performance is essential. They might make programs:
  
  ◆ Harder to understand
  ◆ Harder to debug
  ◆ Harder to maintain
Conclusions

- Avoid inheriting programming styles from other languages
- Program in a declarative way:
  - Improves readability
  - Allows compiler optimizations
- Avoid using the dynamic database when possible
- Look for deterministic computations when programming deterministic problems
- Put tests as soon as possible in the program (early pruning of the tree)
- Delay computations until needed

- Final thought: learning Prolog implementation techniques (e.g., the Warren Abstract Machine) is very instructive and useful. See the available slides and book on the topic.