Programming Paradigms
Unit 10 — Advanced Concepts

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE
Outline

1. Interactive Programs
2. Sorting
3. Mapping
4. Foundation, Strengths and Weaknesses
Outline

1 Interactive Programs
2 Sorting
3 Mapping
4 Foundation, Strengths and Weaknesses
Interactive loops are implemented by while loops in conventional languages.

The following Prolog program reads and echos from the input until one of the words 'quit' or 'exit' is input:

```prolog
echo :- read(X), echo(X).

echo(X) :- last_input(X), !.

echo(X) :-
    write(X), nl,
    read(Y), !,
    echo(Y).

last_input(quit).
last_input(exit).
```

The predicate `read(X)` reads the next term from the input stream and matches it with `X` (must be followed by a '.', which is not part of the term). `read(X)` succeeds only once, i.e., no alternative choice upon backtracking.
Interactive Programs with `repeat/1`

An alternative way to implement a read/echo loop is to use the built-in predicate `repeat/0`, which is implemented as follows:

```prolog
repeat.
repeat :- repeat.
```

- If we put `repeat` in a goal, it **always succeeds** on backtracking.
- This allows to transform goals/rules that have no choice into goals/rules that always succeed again on backtracking.
- Examples are `read` and `write`, which have no choices.
Interactive Programs with `repeat/2`

- With `repeat`, the read/echo program looks as follows

```prolog
echo2 :-
    repeat,
    read(X),
    write(X),
    nl,
    ( X = 'quit' ; X = 'exit' ),
!.
```

- The operator `;` specifies a **disjunction** of goals:
  - **X ; Y** succeeds if at least one of the two **X** or **Y** succeeds
  - If **X** fails, then an attempt is made to satisfy **Y**
  - If **Y** fails, the entire disjunction fails

- Disjunction allows to express **alternatives** within the same clause:
  - Can also be replaced by several facts and rules.

- It is advisable to put a disjunction into parentheses
Outline

1 Interactive Programs

2 Sorting

3 Mapping

4 Foundation, Strengths and Weaknesses
Naïve Sorting

A naïve way to sort a list $L$ follows the generate and test pattern

- Generate first a permutation $S$ of the elements in $L$
- Test if the resulting list $S$ is sorted

```
sort(L, S) :- permutation(L,S),
              sorted(S),
              !.

sorted([]).
sorted([X]).
sorted([X,Y|T]) :- X < Y,
                   sorted([Y|T]).
```

This is not a very efficient way of sorting a list

... and we would have to write a different predicate for different sort orders

Both issues will be addressed in the following
Insertion Sort

- In the insertion sort method, each item of a list is considered one at a time and inserted into a new list in the appropriate position.

- Predicate `insort(L,S)` succeeds when list `S` is a sorted version of list `L`

\[
\begin{align*}
\text{insort([], []).} \\
\text{insort([X|L], M) :-} \\
& \text{insort(L, N),} \\
& \text{insortx(X, N, M).} \\
\text{insortx(X, [A|L], [A|M]) :-} \\
& A < X, \\
& !, \\
& \text{insortx(X, L, M).} \\
\text{insortx(X, L, [X|L]).}
\end{align*}
\]
Constructing Structures

A more general-purpose insertion sorting predicate is to pass the ordering predicate as an argument of insort, e.g.,

- insort([3,2,1], S, '<') or
- insort([3,2,1], S, aless), where aless is self-defined order predicate

In order to call the ordering predicate inside the sorting predicate, we need first to construct a predicate.

The predicate =.. (also pronounced "univ") allows to construct a structure from a list of arguments.

The goal \( P =.. L \) means that \( L \) is the list consisting of the functor of the predicate \( P \) followed by its arguments.

?- P =.. [foo, A, B, C].
P = foo(A,B,C)
yes

?- foo(a,b,c) =.. L.
L = [foo, a, b, c]
yes
Generalized Insertion Sort / 1

- Predicate \texttt{insortg(L,S,OrderPred)} succeeds when list \( S \) is a sorted version of list \( L \), using the sort predicate \texttt{OrderPred}

\[
\begin{align*}
\text{insortg}([], [], \_). \\
\text{insortg}([X|L], M, O) :&- \\
&\text{insortg}(L, N, O), \\
&\text{insortgx}(X, N, M, O).
\end{align*}
\]

\[
\begin{align*}
\text{insortgx}(X, [A|L], [A|M], O) :&- \\
&P =\_ [O, A, X], \\
&\text{call}(P), \\
&!, \\
&\text{insortgx}(X, L, M, O). \\
\text{insortgx}(X, L, [X|L], \_O).
\end{align*}
\]

- **Predicate** \texttt{call(P)} tries to prove \( P \) as a goal
  - Returns \texttt{true} if \( P \) can be satisfied, \texttt{false} otherwise
Generalized Insertion Sort /2

- We can use insortg as follows

```prolog
?- insortg([4,3,2,1], S, '<').
S = [1,2,3,4]
yes

?- insortg([4,3,2,1,5], S, '>').
S = [5,4,3,2,1]
yes
```
Alphabetical Sorting/1

- For alphabetical sorting (or sorting more complex structures), we can write our own sorting predicates.
- If we want to sort atoms, we need the predicate `name(A,L)` that relates atom A to the list L of character (ASCII codes) that make it up.
  - i.e., `name` transforms atom A into a list L of characters or vice versa.

?- name(apple, L).

?- name(A, [97,112,112,108,101]).
A = apple

?- name(apple, "apple")
true

?- name(apple, "pear")
false
Alphabetical Sorting/2

- The following predicate `aless(X, Y)` implements alphabetical sorting
  - i.e., succeeds if `X` is alphabetically smaller than `Y`

  ```prolog
  aless(X, Y) :- name(X, XL),
  name(Y, YL),
  alessx(XL, YL).
  
  alessx([], []).
  alessx([X|T1], [Y|T2]) :- X < Y.
  alessx([X|T1], [X|T2]) :- alessx(T1, T2).
  ```

- Now we can pass `aless` to the generalized insertion sort predicate

  ```prolog
  ?- insortg([c,b,a], S, aless).
  S = [a,b,c]
  true
  
  ?- insortg([tom,joe,ann], S, aless).
  S = [ann,joe,tom]
  true
  ```
Outline

1. Interactive Programs
2. Sorting
3. Mapping
4. Foundation, Strengths and Weaknesses
Mapping Lists/1

- **Mapping** one structure component-by-component to another structure is frequently needed, e.g., replace negative numbers in a list by zero.

- The following predicate `maplist(P,L,M)` applies predicate `P` to each element in `L` to form a new list `M`.

  ```prolog
  maplist(_, [], []) :- !.
  maplist(P, [X|L], [Y|M]) :-
    Q =.. [P,X,Y],
    call(Q),
    maplist(P,L,M).
  ```

- To compute the absolute value `|x|` of a list of numbers, we need the following predicate.

  ```prolog
  absolute(X,Y) :- X < 0, Y is X * -1, !.
  absolute(X,X).
  ```

- `?- maplist(absolute, [2,-1,5,-10], L).` 
  `L = [2,1,5,10]` 
  `true`
The same predicate `maplist` can be used to implement a simple translation tool that translates a list of words/sentence into another language.

For that, we just need a dictionary:

```
dict(the,le).
dict(chases,chasse).
dict(dog,chien).
dict(cat,chat).
```

```
?- maplist(dict, [the,dog,chases,the,cat], L).
L = [le,chien,chasse,le,chat]
true
```

**Example:** Write a predicate `maplist/4` that maps $X \times Y \rightarrow Z$. 
Applying a Predicate

A simplification of `maplist` is `applist(P,L)`, which applies predicate `P` that is assumed to have one argument to all elements of list `L`:

```prolog
applist(_, []) :- !.
applist(P, [X|L]) :-
    Q =.. [P,X],
    call(Q),
    applist(P,L).
```

The following will print each element of a list in a separate line:

```prolog
?- applist(writeln,[a,b,c]).
a
b
c
true.
```
Mapping is not restricted to lists, but can be defined for any kind of structure.

Consider arithmetic expression made up of '*' and '+'
- e.g., \(3 + 4 \times a + b\)

Suppose we want to remove multiplications by 1 and additions by 0.

The algebraic simplifications can be described by a predicate

\[s(Op, La, Ra, Ans)\]

It represents that an expression consisting of an operator \(Op\) with left argument \(La\) and right argument \(Ra\) can be simplified to \(Ans\)
- e.g., \(s(+, X, 0, X)\) represents that \(X + 0 = X\)
The simplification rules are

\[
\begin{align*}
&s(\text{+} , X , 0 , X) . \\
&s(\text{+} , 0 , X , X) . \\
&s(\text{+} , X , Y , X+Y) . \quad /* \text{catchall for + */} \\
&s(\times , \_ , 0 , 0) . \\
&s(\times , 0 , \_ , 0) . \\
&s(\times , \_ , 1 , X) . \\
&s(\times , X , 1 , X) . \\
&s(\times , X , Y , X*Y) . \quad /* \text{catchall for * */} \\
\end{align*}
\]

The "catchall" rules (at the end of each operator's part) are needed for the case that no simplification can be applied

- This rule will always succeed, which is important when used in a mapping.
With the above rules in place, we can write a simplification predicate that maps and simplifies arithmetic expressions.

\[
simplify(E, E) :- \text{atomic}(E), \, !.
\]
\[
simplify(E, F) :-
E =.. \, [Op, La, Ra],
\text{simplify}(La, X),
\text{simplify}(Ra, Y),
s(Op, X, Y, F), \, !.
\]

To simplify an expression \( E \), we need first to simplify the left-hand argument of \( E \), then the right-hand argument of \( E \), then see if the simplified result can further be simplified.

atomic\((E)\) succeeds if \( E \) is either an atom or an integer.

Simplifying expressions:

?- simplify(a*10+(b+0+c)*1, S).
\( S = a \ast 10 + (b + c) \)
true
Outline

1. Interactive Programs
2. Sorting
3. Mapping
4. Foundation, Strengths and Weaknesses
Mathematical Foundation

- Just a brief explanation how Prolog fits into the framework of mathematical logic
  - First-order logic is a powerful mathematical tool for formalizing descriptions
    - It is also sometimes called predicate logic
  - Unfortunately, first-order logic is not decidable
  - Prolog is based on a decidable subset of first-order logic called Horn clauses
  - It is still Turing-complete, though
Strengths of Prolog

- Prolog is very well suited for application centered around Artificial Intelligence (AI)
  - Natural-language processing
  - AI behavior in games
  - Constraint satisfaction problems, such as time tabling and scheduling
- Prolog (or its descendants) is used in the context of the Semantic Web
  - A variant called Datalog is used in databases
- Also used for simulation and prediction software
Weaknesses of Prolog

- Prolog has a steeper learning curve compared to other languages
- Fairly focused niche applications, not really a general-purpose language
- There are scalability issues, the basic matching strategy used by Prolog is computationally expensive
  - Has problems to process large data sets
- It is not as declarative as it seems at first glance
  - If you want to write efficient Prolog programs, you have to know what is going on behind the scenes
Summary

- Prolog is a declarative programming language based on First-order logic
  - Specifies what to compute and not how to do it
- A Prolog program/knowledge base consists of facts and rules
- Evaluating a Prolog program means to prove a goal
  - Thereby, key concepts are instantiation, matching, and backtracking
- Prolog uses recursion instead of loops
- Lists and structures are two very important data structures
- The cut operator allows to stop backtracking
  - Should be used with care
  - A better programming style is to replace it by negation
- “Generate and test” is a very common programming pattern
Summary

- **The box model** shows the execution of a Prolog program
  - Has four ports: CALL, EXIT, REDO, FAIL
- **Debugger** shows the program execution according to the box model
  - `trace` provides an exhausitve tracing mode
  - `debug` allows to jump to spy points set by the `spy` predicate
- **Accumulators** are frequently needed to collect intermediate results when traversing structures or lists
  - Helpful to make programs tail-recursive
- **Sorting** is an important operation
  - Generalized insertion sort, which allows to pass a sorting predicate
  - Constructing structures with the `=..` (univ) operator needed
- Another frequent and powerful operation is **mapping** structures and lists
  - General map-functions can be used
- `read` and `write` predicates for simple interactive programs