Prolog: Bits and Pieces

Backtracking (again)
- Backtracking = systematic search for alternatives
- Backtracking + recursion replaces:
  - iteration (for/while/repeat structure)
  - recursion
  in imperative languages

Example: Simple Bubble Sort

```prolog
proc bubblesort(var L : list;)
    interchanged := true;
    while interchanged do
        interchanged := false;
        i := 1;
        while (i < n) and not interchanged do
            temp := L[i];
            L[i] := L[i+1];
            L[i+1] := temp;
            interchanged := true
            i := i + 1
        end
    end
end;
```

Bubble Sort in Prolog

```prolog
bubblesort(L, S) :-
    conc([], A, B),
    B < A,
    bubblesort(M, S),
    bubblesort(L, 1).
```

Bubble Sort: Next Pair

```prolog
bubblesort(L, S) :-
    conc(X, [A, B|Y], L),
    B < A,
    bubblesort([A, B|Y], M),
    bubblesort(M, S),
    bubblesort(L, 1).
```

Bubble Sort: Recursion

```prolog
bubblesort(L, S) :-
    conc(X, [A, B|Y], L),
    B < A,
    bubblesort([A, B|Y], M),
    bubblesort(M, S),
    bubblesort(L, 1).
```

Example: Simple Bubble Sort

```prolog
proc bubblesort(var L : list;)
    interchanged := true;
    while interchanged do
        interchanged := false;
        i := 1;
        while (i < n) and not interchanged do
            temp := L[i];
            L[i] := L[i+1];
            L[i+1] := temp;
            interchanged := true
            i := i + 1
        end
    end
end;
```

Example: Simple Bubble Sort

```prolog
proc bubblesort(var L : list;)
    interchanged := true;
    while interchanged do
        interchanged := false;
        i := 1;
        while (i < n) and not interchanged do
            temp := L[i];
            L[i] := L[i+1];
            L[i+1] := temp;
            interchanged := true
            i := i + 1
        end
    end
end;
```
### Bubble Sort: Recursion

```
bubsort(L, S) :-
    conc(X, A, B[Y], L),
    B < A,
    conc(X, B, A[Y], M), !,
    bubsort(M, S),
    bubsort(L, L).
```

### Special Predicates

- Evaluation of expressions, e.g.
  
  ```
  ?- X is (10 + 2) * 4.
  ```

- Matching ‘=’ versus evaluation ‘is’

### Terms (again)

- **Term**: functor(arg1, arg2, ..., argn)
- **args**: again term

**Example:**

```
square(upper_left(a),
       upper_right(b),
       lower_left(c),
       lower_right(d)).
```

### Arithmetic Expressions

- For example:
  
  ```
  ?- X = (10 + 2) * 4.
     X = (10 + 2) * 4
     yes.
  ```

- Prolog sees term: `*+(+(10, 2), 4)`

- Example:
  
  ```
  ?- (10 + 2) * 4 = *+(+(10, 2), 4).
  yes.
  ```

### Evaluation of Arithmetic Expressions: ‘is’

- **Immediate evaluation:**
  
  ```
  ?- X is (10 + 2) * 4.
     X = 48
     yes.
  ```

- **Delayed evaluation:**
  
  ```
  ?- X = (10 + 2) * 4, Z is X.
     X = (10 + 2) * 4
     Z = 48
     yes.
  ```

### Example: length

- **Length of a list L**: length(L, N);
  
  ```
  length([], 0).
  length([_ | Tail], N) :-
    N = M + 1,
    length(Tail, M).
  ```

- **Queries/calls:**
  
  ```
  ?- length([a, b], X), Z is X.
     X = 0 + 1 + 1
     Z = 2
     yes.
  ```
Length with ‘is’

- Length of a list L:
  - length([], 0).
  - length([_ | Tail], N) :-
    length(Tail, M),
    N is M + 1.
- Queries/calls:
  - ?- length([a,b], X).
    X = 2
    yes

Length with ‘is’

- Length of a list L:
  - length([], 0).
  - length([_ | Tail], N) :-
    N is M + 1,
    length(Tail, M).
- Queries/calls:
  - ?- length([a,b], X).
    error, M is uninstantiated

Conclusions ‘is’

- Evaluation by ‘is’ requires that all variables in the expression at the right-hand side are instantiated at evaluation time

- This may impose a specific order on the clause’s conditions

- This contradicts declarative (logic) programming, where order is irrelevant