CS 603: Programming Languages

Lecture 24

Spring 2004

Department of Computer Science
University of Alabama
Joel Jones

Professor Marjan Mernik

The University of Maribor

Automatic generation of language-based tools and Grammar-based Systems

Monday, 7-March

11:00 AM in HO 108

Abstract

A grammar-based system is any system that uses a grammar and/or sentences produced by this grammar to solve various problems outside the domain of programming language definition and its implementation. The vital component of such a system is well structured and expressed with a grammar or with sentences produced by this grammar in an explicit or implicit manner. In this talk several language-based tools will be presented (e.g. editors, inspectors, debuggers and visualizers/animators) which are automatically generated using an attribute grammar-based compiler generator called LISA. Furthermore, several examples of our grammar-based systems will be given to show usefulness of grammars in software engineering.

Recursive Inference

Example, given

(1) $\forall$ X[mortal(son_of(X)) $\leftarrow$ mortal(X)]

(2) mortal(plato)

derive:

mortal(son_of(plato))

(using X=plato)

mortal(son_of(son_of(plato)))

(using X=son_of(plato))

mortal(son_of(son_of(son_of(plato))))

(using X=son_of(son_of(plato)))

Prolog Notation

A rule:

$\forall X [p(X) \leftarrow (q(X) \wedge r(X))]$

is written as

$p(X) \leftarrow q(X), r(X).$

Prolog conventions:

- variables begin with upper case (A, B, X, Y, Big, Small, ACE)
- constants begin with lower case (a, b, x, y, plato, aristotle)

Query = list of facts with variables, e.g.

- mortal(X)
- sorted([5,3,4,9,2], X)
- sonOf(martha,S), sonOf(george,S)

Prolog program = facts+rules+query

Prolog Syntax

< fact > → < term > .

< rule > → < term > :- < terms > .

< query > → < terms > .

< term > → < number > | < atom > | <variable>
| < atom > (< terms >)

< terms > → < term > | < term > , < terms >

Syntax

- Integers
- Atoms: user defined, supplied
  - name starts with lower case: john, student2
- Variables
  - begin with upper case: Who, X
  - '_' can be used in place of variable name
- Structures
  - student(ali, freshman, 194).
- Lists
  - [x, y, Z]
  - [Head | Tail]
    - syntactic sugar for . (Head, Tail)
  - [ ]

Constructors like student and "." are called functors in Prolog

Prolog Introduction

```
/* list of facts in prolog, stored in an
   ascii file, 'family.pl'*/
mother(mary, ann).
mother(mary, joe).
mother(sue, mary ).

father(mike, ann).
father(mike, joe).

grandparent(sue, ann).
```

Prolog Introduction (cont.)

- ```
  /* reading the facts from a file */
  ?-  consult ( family).
  %family compiled, 0.00 sec, 828 bytes
  ```
- Comments are either bound by "/*","*/" or any characters following the "%".
- Structures are just relationships. There are no inputs or outputs for the variables of the structures.
- The swipl documentation of the built-in predicates does indicate how the variables should be used.
  - pred(+var1, -var2, +var3).
    - + indicates input variable
    - - indicates output variable
  ```

```
/* Prolog the order of the facts and rules is the order it is
searched in */
/* Variation from pure logic model  */

2 ?- father( X, Y ).

X = mike   /* italics represents computer output */
Y = ann  ;   /* I type ‘;’ to continue searching the data base */

X = mike
Y = joe  ;

no

3 ?- father( X, joe).

X = mike ;

no
```

Rules

- `parent( X , Y ) :– mother( X , Y ). /* If mother( X ,Y ) then parent( X ,Y ) */`
- `parent( X , Y ) :– father( X , Y ).`
-
- `/* Note:  grandparent(sue, ann).  redundant */`
- `/* if parent( X ,Y ) and parent(Y,Z ) then grandparent( X ,Z ). */`

Pair Up:

- Define grandparent

grandparent(X , Z) :– parent(X ,Y),parent(Y, Z).

Ancestry Example

?- parent(X , ann), parent(X , joe).
X = mary;
X = mike
yes

?- grandparent(sue, Y).
Y = ann;
Y = joe
yes

program

```
mother(mary, ann).
mother(mary, joe).
mother(sue, mary).
father(mike, ann).
father(mike, joe).

parent( X ,Y ) :– mother( X ,Y ).
parent( X ,Y ) :– father( X ,Y ).
grandparent( X , Z ) :– parent( X ,Y ),parent(Y, Z ).
```

Tracing Queries

Example Query #1:

query ?- mother(X,joe). X=X,Y=joe binding

mother(mary,ann). /* fails */

mother(mary,joe). /* succeeds */

program

```
mother(mary, ann).
mother(mary, joe).
mother(sue, mary).
father(mike, ann).
father(mike, joe).

parent( X ,Y ) :– mother( X ,Y ).
parent( X ,Y ) :– father( X ,Y ).
```

Example Query #2:

?- parent(X,joe). X=X,Y=joe

parent(X,Y) :- mother(X,joe).

?- mother(X,joe). X=X,Y=joe

mother(mary,ann). /* fails */

mother(mary,joe). /* succeeds */

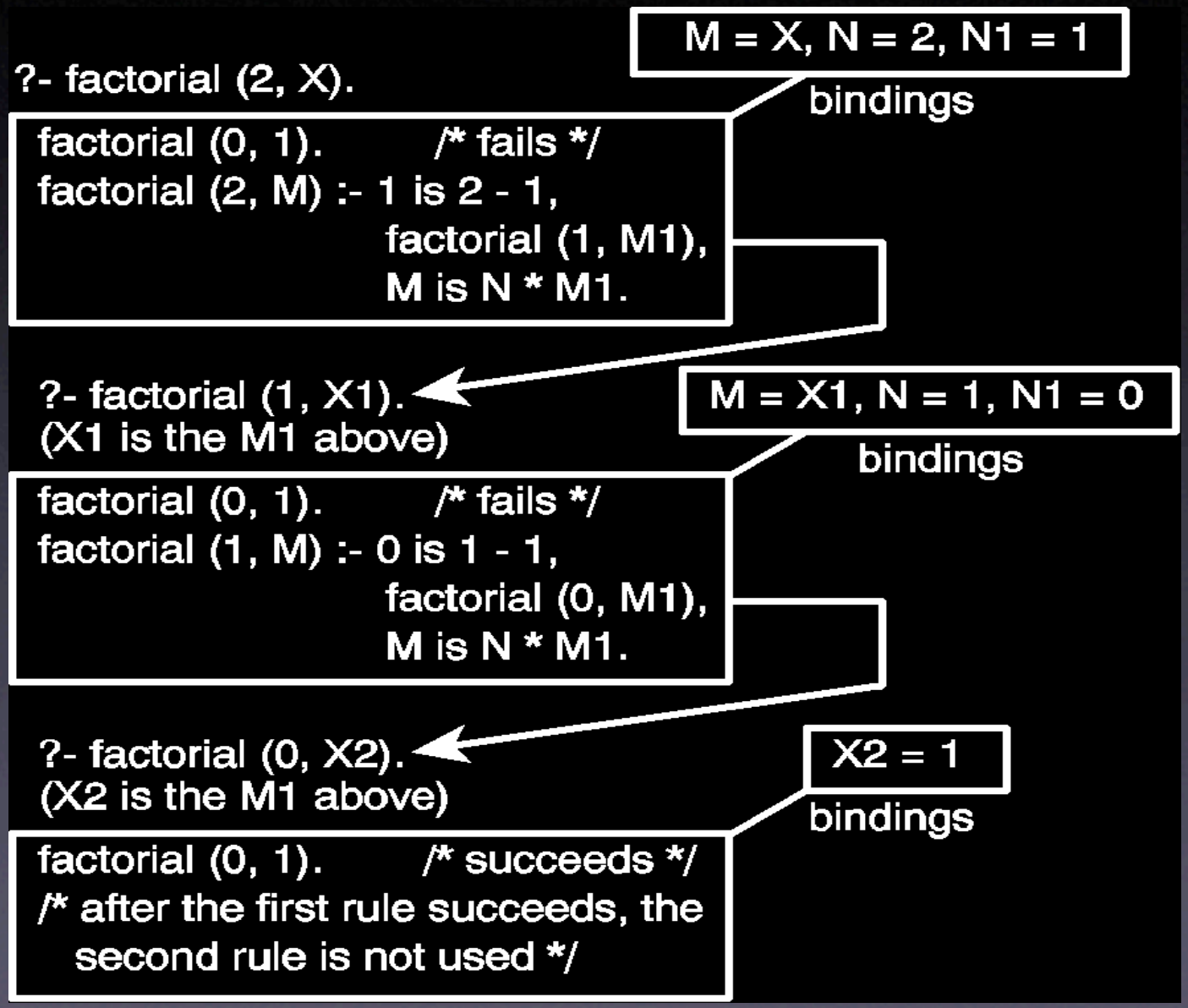
Tracing exercise

```
/* specification of factorial n! */
factorial(0,1).
factorial(N, M):– N1 is N – 1, factorial (N1, M1), M is N*M1.
```

Pair Up:

- Trace factorial(2,X)

Solution


?- factorial (2, X).
factorial (0, 1). /* fails */
factorial (2, M) :- 1 is 2 - 1,
factorial (1, M1),
M is N * M1.
M = X, N = 2, N1 = 1
bindings
?- factorial (1, X1).
(X1 is the M1 above)
factorial (0, 1). /* fails */
factorial (1, M) :- 0 is 1 - 1,
factorial (0, M1),
M is N * M1.
M = X1, N = 1, N1 = 0
bindings
?- factorial (0, X2).
(X2 is the M1 above)
factorial (0, 1). /* succeeds */
/* after the first rule succeeds, the
second rule is not used */
X2 = 1
bindings

Recursion in Prolog

- trivial, or boundary cases
- 'general' cases where the solution is constructed from solutions of (simpler) version of the original problem itself.
- What is the length of a list ?
- THINK:
  The length of a list, [e | Tail], is 1 + the length of Tail
- What is the boundary condition?
  - The list [ ] is the boundary. The length of [ ] is 0.

Recursion

- Where do we store the value of the length?—an accumulator

```
mylength( [ ], 0).
mylength( [X | Y], N):–mylength(Y, Nx), N is Nx+1.

? – mylength( [1, 7, 9], X ).
X = 3
? - mylength(jim, X ).
no
? - mylength(Jim, X ).
Jim = [ ]
X = 0
```

Recursion

```
mymember( X , [X | _ ] ).
mymember( X , [ _ | Z ] ) :–  mymember( X , Z ).
% equivalently:  However swipl will give a warning
% Singleton variables :Y W
mymember( X , [X | Y] ).
mymember( X , [W | Z ] ) :– mymember( X , Z ).

1?–mymember(a, [b, c, 6] ).
no
2? – mymember(a, [b, a, 6] ).
yes
3? – mymember( X , [b, c, 6] ).
X = b;
X = c;
X = 6;
no
```

Appending Lists I

- The Problem: Define a relation append(X,Y,Z) to mean that X appended to Y yields Z
-
- The Program:

```
append([],Y,Y).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

Watch it work:

```
 ?- [append].
 ?- append([1,2,3,4,5],[a,b,c,d],Z).
Z = [1,2,3,4,5,a,b,c,d]?;
no
 ?- append(X,Y,[1,2,3]).
X = [] Y = [1,2,3]?;
X = [1] Y = [2,3]?;
X = [1,2] Y = [3]?;
X = [1,2,3] Y = []?;
no
 ?-
 ?- append([1,2,3],Y,Z).
Z = [1,2,3|Y]
```

Order Dependency

- Previous Program

  mylength([ ], 0).
  mylength([X | Y], N):–mylength(Y, Nx), N is Nx+1.
- The Program:

```
llength([],0).

llength([X|Z],N) :- N is M + 1, llength(Z,M).
?- [length2].
?- llength([a,b,c,d],M).
uncaught exception: error(instantiation_error,(is)/2)
```