KI –Programmierung

A First Look at Prolog

Bernhard Beckert

Winter Term 2007/2008

Universität Koblenz–Landau

This set of slides is based on the slides by Adam Brooks Webber
provided on his web site at
http://www.webber-labs.com/mpl.html
to accompany Chapter 19 of his book
Modern Programming Languages: A Practical Introduction

Terms

Terms

- Everything in Prolog is built from *terms*:
  - Prolog programs
  - The data manipulated by Prolog programs
- Three kinds of terms:
  - Constants: integers, real numbers, atoms
  - Variables
  - Compound terms

Constants

- Integer constants: `123`
- Real constants: `1.23`
- Atoms:
  - A lowercase letter followed by any number of additional letters, digits or underscores: `fred`
  - A sequence of non-alphanumeric characters: `*`, `.`, `=`, `@#$`
  - Plus a few special atoms: `[]`

Atoms vs. Variables

- An atom can look like a Java variable:
  - `i`, `size`, `length`
- But an atom is not a variable
  - it is not bound to anything
  - never equal to any other atom
  - cannot be instantiated
  - does not have a value (except itself)

Variables

- Any name beginning with an uppercase letter or an underscore, followed by any number of additional letters, digits or underscores:
- **`X`**, **`Child`**, **`Fred`**, **`_`**, **`_123`**

- Most variables start with an uppercase letter
- Those starting with an underscore, including _, get special treatment

Compound Terms

- An atom followed by a parenthesized, comma-separated list of one or more terms: **`x(y,z)`**, **`+(1,2)`**, **`.(1,[])`**, **`parent(adam,seth)`**, **`x(Y,x(Y,Z))`**

- A compound term can look like a function call: **`f(x,y)`**
- Again, this is misleading
- Think of them as structured data

Terms

- All Prolog programs and data are built from terms

- **`+(1,2)`** is usually written as **`1+2`**
- But these are not new kinds of terms, just abbreviations

Unification

- Pattern-matching using Prolog terms
- Two terms *unify* if there is some way of binding their variables that makes them **identical**

- **`parent(adam,Child)`**
`parent(adam,seth)`
- unify by binding the variable **`Child`** to the atom **`seth`**
- More details later

The Prolog Database

- A Prolog language system maintains a collection of facts and rules of inference
- It is like an internal database
- A Prolog program is just a set of data for this database
- The simplest kind of thing in the database is a *fact*: a term followed by a period

Example

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```

- A Prolog program of six facts
- Defining a *predicate* **`parent`** of *arity* 2
- We would naturally interpret these as facts about families: Kim is the parent of Holly and so on

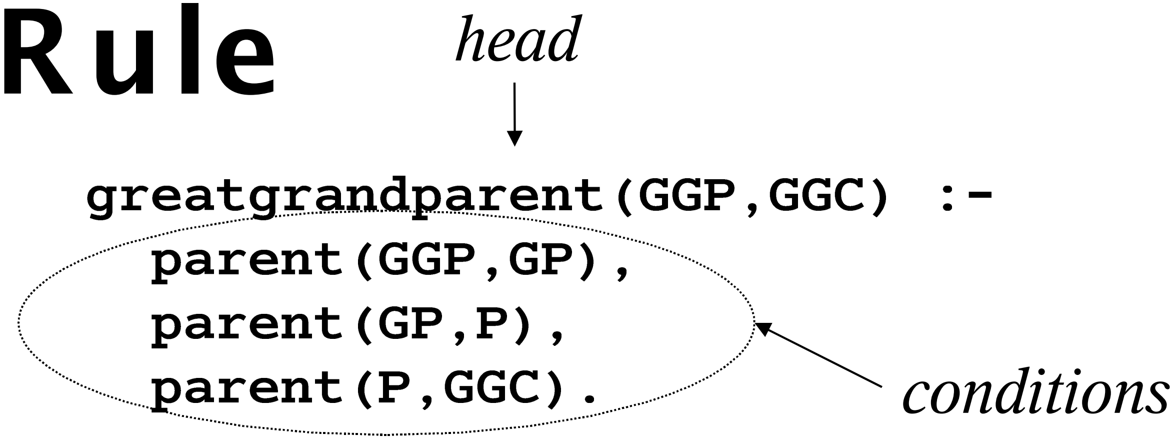
Rules

The Need For Rules

- Previous example had a lengthy query for great-grandchildren of Esther
- It would be nicer to query directly:

  ```
  greatgrandparent(esther,GGC)
  ```
- But we do not want to add separate facts of that form to the database
- The relation should follow from the **`parent`** relation already defined

A Rule



```
greatgrandparent(GGP,GGC) :-
  parent(GGP,GP),
  parent(GP,P),
  parent(P,GGC).
```

- A rule says how to prove something: to prove the head, prove the conditions

- To prove `greatgrandparent(GGP,GGC)`, find some `GP` and `P` for which you can prove `parent(GGP,GP)`, then `parent(GP,P)` and then finally `parent(P,GGC)`

Program with a Rule

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
greatgrandparent(GGP,GGC) :-
  parent(GGP,GP), parent(GP,P), parent(P,GGC).
```

- A program consists of a list of *clauses*
- A clause is either a fact or a rule, and ends with a period

Example

```
?- greatgrandparent(esther,GreatGrandchild).

GreatGrandchild = holly

Yes
```

- Shows initial query and final result
- Also, there are intermediate *goals:*
  - The first goal is the initial query
  - The next is what remains to be proved after transforming the first goal using one of the rules
  - And so on, until nothing remains to be proved

1. `parent(kim,holly).`
2. `parent(margaret,kim).`
3. `parent(margaret,kent).`
4. `parent(esther,margaret).`
5. `parent(herbert,margaret).`
6. `parent(herbert,jean).`
7. `greatgrandparent(GGP,GGC) :-`
   `parent(GGP,GP), parent(GP,P), parent(P,GGC).`

`greatgrandparent(esther,GreatGrandchild)`

⇓ Clause 7, binding **`GGP`** to **`esther`** and **`GGC`** to **`GreatGrandChild`**

`parent(esther,GP), parent(GP,P), parent(P,GreatGrandchild).`

⇓ Clause 4, binding **`GP`** to **`margaret`**

`parent(margaret,P), parent(P,GreatGrandchild)`

⇓ Clause 2, binding **`P`** to **`kim`**

`parent(kim,GreatGrandchild)`

⇓ Clause 1, binding **`GreatGrandchild`** to **`holly`**

Rules Using Rules

```
grandparent(GP,GC) :-
  parent(GP,P), parent(P,GC).

greatgrandparent(GGP,GGC) :-
  grandparent(GGP,P), parent(P,GGC).
```

- Same relation, defined indirectly
- Note that both clauses use a variable `P`
- The scope of the definition of a variable is the clause that contains it

Recursive Rules

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :-
  parent(Z,Y),
  ancestor(X,Z).
```

- **X** is an ancestor of **Y** if:
  - Base case:
    X is a parent of **Y**
  - Recursive case:
    there is some **Z** such that **Z** is a parent of **Y**, and **X** is an ancestor of **Z**
- Prolog tries rules in their syntactic order, so put base-case rules and facts first

```
?- ancestor(jean,jean).

No
?- ancestor(kim,holly).

Yes
?- ancestor(A,holly).

A = kim ;

A = margaret ;

A = esther ;

A = herbert ;

No
```

Core Syntax Of Prolog

- You have seen the complete core syntax:

```
<clause>  ::=   <fact>  |  <rule>
<fact>  ::=   <term>  .
<rule>  ::=   <term>  :-  <termlist>  .
<termlist>  ::=   <term>  |  <term>  ,  <termlist>
```

- There is not much more syntax for Prolog than this:
  it is a very simple language
- Syntactically, that is!

Operators

Operators

- Prolog has some predefined operators (and the ability to define new ones)

- An operator is just a predicate for which a special abbreviated syntax is supported

The = Predicate

- The goal **`=(X,Y)`** succeeds if and only if **`X`** and **`Y`** can be unified:

```
?- =(parent(adam,seth),parent(adam,X)).

X = seth

Yes
```

- Since = is an operator, it can be and usually is written like this:

```
?- parent(adam,seth)=parent(adam,X).

X = seth

Yes
```

Arithmetic Operators

- Predicates `+`, `-`, `*` and `/` are operators too, with the usual precedence and associativity

```
?- X = +(1,*(2,3)).

X = 1+2*3

Yes
?- X = 1+2*3.

X = 1+2*3

Yes
```

Prolog lets you use operator notation, and prints it out that way, but the underlying term is still **`+(1,*(2,3))`**

Not Evaluated

```
?- +(X,Y) = 1+2*3.

X = 1
Y = 2*3

Yes
?- 7 = 1+2*3.

No
```

- The term is still **`+(1,*(2,3))`**
- It is not evaluated
- There is a way to make Prolog evaluate such terms, but we won’t need it yet

Lists

Lists in Prolog

- The atom `[]` represents the empty list

- The predicate . is the list constructor

List Notation

List notation	Term denoted
`[]`	`[]`
`[1]`	`.(1,[])`
`[1,2,3]`	`.(1,.(2,.(3,[])))`
`[1,parent(X,Y)]`	`.(1,.(parent(X,Y),[]))`

- [a,b,c] and [a|[b,c]] notations for lists
- These are just abbreviations for the underlying term using the . Predicate
- Prolog usually displays lists in this notation

Example

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

X = [1, 2, 3]

Yes
?- .(X,Y) = [1,2,3].

X = 1
Y = [2, 3]

Yes
```

List Notation With Tail

List notation	Term denoted
[1|X]	**.(1,X)**
[1,2|X]	**.(1,.(2,X)))**
[1,2|[3,4]]	same as **[1,2,3,4]**

- Last in a list can be symbol | followed by a term for the tail of the list
- Useful in patterns:
  [1,2|X] unifies with any list that starts with **1,2** and binds **X** to the tail

```
?- [1,2|X] = [1,2,3,4,5].

X = [3, 4, 5]

Yes
```