Contents

Comparing Terms

- Prolog contains an important predicate for comparing terms
- This is hte identity predicate ==/2
- The identity predicate ==/2 does not instantiate variables, that is, it behaves differently from =/2

```
?- a==a.
true
?- a==b
false
?- a=='a'
true
?- a==X
X=_443
false
```

- The predicate ==/2 s defined so that is succeeds in precisely those cases where ==/2 fails
- In other words, it succeeds whenever two terms are **not identical**, and fails otherwise

```
?- a \== a.
false
?- a \== b.
true
?- a \== 'a'.
false
?- a \== X.
X = _443
true
```

Comparing Variables

- Two different **uninstantiated** variables are not identical terms
- Variables **instantiated** with a term T are identical to T

```
?- X==X.
X=_443
true
?- Y==X
Y=_442
```

```
X=_443
false
?- a=U, a==U.
U=_443
true
```

Terms with a Special Notation

- Sometimes terms look different, but Prolog regards them as identical
- For example: **a** and **'a'**, but there are many other cases
- Why does Prolog do this?
  - Because it makes programming more pleasant
  - More natural way of coding Prolog programs

Arithmetic Terms

- `+`, `-`, `<`, `>`, etc are functors and expressions such as `2+3` are actually ordinary complex terms
- The term `2+3` is identical to the term `+(2, 3)`

```
?- 2+3 == +(2, 3).
true
?- -(2, 3) == 2-3.
true
?- (4<2) == <(4, 2).
true
```

Summary of Comparison Predicates

=	Unification predicate
=	Negation of unification predicate
==	Identity predicate
==	Negation of identity predicate
=:=	Arithmetic equality predicate
==	Negation of arithmetic equality predicate

Lists as Terms

- Another example of Prolog working with one internal representation, while showing another to the user
- sing the | constructor, there are many ways of writing the same list

```
?- [a, b, c, d] == [a|[b, c, d]].
true
?- [a, b, c, d] == [a, b, c|[d]].
true
?- [a, b, c, d] == [a, b, c, d|[]].
true
?- [a, b, c, d] == [a, b|[c, d]].
true
```

Lists Internally

- Internally, lists are built out of two special terms:
  - `[]` which represents the empty list
  - `.` a functor of arity 2 used to build non-empty lists
- There two terms are also called *list constructors*
- A recursive definition shows how they construct lists

Definition of Prolog List

- The empty list is the term `[]`
- It has length 0
- A non-empty list is any term of the form `.(term, list)` where *term* is any Prolog term, and *list* is any Prolog list
- If *list* has length *n*, then `.(term, list)` has length *n+1*

```
?- .(a, []) == [a].
true
?- .(f(d, e), []) == [f(d, e)].
true
?- .(a, .(b, [])) == [a, b]
true
?- .(a, .(b, .(f(d, e), []))) == [a, b, f(d, e)].
true
```

Internal List Representation

- Works similar to the | notation
- It represents a list in two parts
  - Its first element, the *head*
  - The rest of the list, the *tail*
- The trick is to read these terms as trees
  - Internal nodes are labeled with .
  - All nodes have two daughter nodes
    * Subtree under left daughter is the head
    * Subtree under right daughter is the tail

Examining Terms

- We will now look at built-in predicates that let us examine Prolog terms more closely
  - Predicates that determine the type of terms
  - Predicates that tell us something about the internal structure of terms

The Structure of Terms

- Given a complex term of unknown structure, what kind of information might we want to extract from it?
- Obviously:
  - The functor
  - The arity
  - The argument
- Prolog provides built-in predicates to produce this information

The functor/3 predicate

- The functor/3 predicate gives the functor and arity of a complex predicate

```
?- functor(friends(lou, andry), F, A).
F = friends
A = 2
true
```

```
?- functor([loud, andry, vicky], F, A).
F = .
A = 2
true
?- functor(mia, F, A).
F = mia
A = 0
true
```

Constructing Terms

```
?- functor(Term, friends, 2).
Term = friends(_, _)
true
```

Checking for Complex Terms

```
complexTerm(X) :- nonvar(X), functor(X, _, A), A>0.
```

- Prolog also provides us with the predicate arg/3
- This predicate tells us about the arguments of complex terms
- It takes three arguments
  - A number N
  - A complex term T
  - The Nth argument of T

```
?- arg(2, likes(lou, andy), A).
A = andy
true
```

Strings

- Strings are represented in Prolog by a list of character codes
- Prolog offers double quotes for an easy notation for strings

```
?- S = "Vicky".
S = [86, 105, 99, 107, 121]
true
```

Working with Strings

- There are several standard predicates for working with strings
- A particular useful one is atom_codes/2

```
?- atom_codes(Vicky, S).
S = [118, 105, 99, 107, 121]
true
```

Operators

- As we have seen, in certain cases Prolog allows us to use operator notations that are more user friendly
- Recall, for instance, the arithmetic expressions such as 2+2 which internally means +(2, 2)
- Prolog also have a mechanism to add your own operators

Properties of Operators

- Infix operators
  - Functors written *between* their arguments
  - Examples: `+ - = == , ; . -->`
- Prefix operators
  - Functors written *before* their argument
  - Example: `-`
- Postfix operators
  - Functors written *after* their argument
  - Example: `++`

Precedence

- Every operator has a certain precedence to work out ambiguous expressions
- For instance, does `2+3*3` mean `2+(3*3)` or `(2+3)*2`?
- Because the precedence of + is greater than that of *, Prolog chooses + to be the main functor of 2+3*3*

Associativity

- Prolog uses associativity to disambiguate operators with the same precedence value
- Example: 2+3+4
  - Does this mean (2+3)+4 or 2+(3+4)?
    * Left associative
    * Right associative
- Operators can also be defined as non-associative, in which case you are forced to use bracketing in ambiguous cases
  - Examples in Prolog: `:-` `-->`

Defining Operators

- Prolog lets you define your own operators
- Operator definitions look like this:
  - `:- op(Precedence, Type, Name).`
  - Precedence: number between 0 and 1200
  - Type: the type of operator

Types of Operators

- yfx: left-associative, infix
- xfy: right-associative, infix
- xfx: non-associative, infix
- fx: non-associative, prefix
- fy: right-associative, prefex
- xf: non-associative, postfix
- yf: left-associative, postfix