Lecture 9: A closer look at terms

- Theory
  - Introduce the == predicate
  - Take a closer look at term structure
  - Introduce strings in Prolog
  - Introduce operators
- Exercises
  - Exercises of LPN: 9.1, 9.2, 9.3, 9.4, 9.5
  - Practical session

Comparing terms: ==/2

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

Comparing terms: ==/2

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

```
?- a==a.
yes

?- a==b.
no

?- a=='a'.
yes

?- a==X.
X = _443
no
```

Comparing variables

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

Comparing variables

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

```
?- X==X.
X = _443
yes

?- Y==X.
Y = _442
X = _443
no

?- a=U, a==U.
U = _443
yes
```

Comparing terms: \==/2

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

Comparing terms: \==/2

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

```
?- a \== a.
no

?- a \== b.
yes

?- a \== 'a'.
no

?- a \== X.
X = _443
yes
```

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

- Recall lecture 5 where we introduced arithmetic
- +, -, <, >, 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)

Arithmetic terms

- Recall lecture 5 where we introduced arithmetic
- +, -, <, >, 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).
yes

?-  -(2,3) == 2-3.
yes

?-  (4<2) == <(4,2).
yes
```

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
- Using the | constructor, there are many ways of writing the same list

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

Prolog 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)
- These 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 few examples…

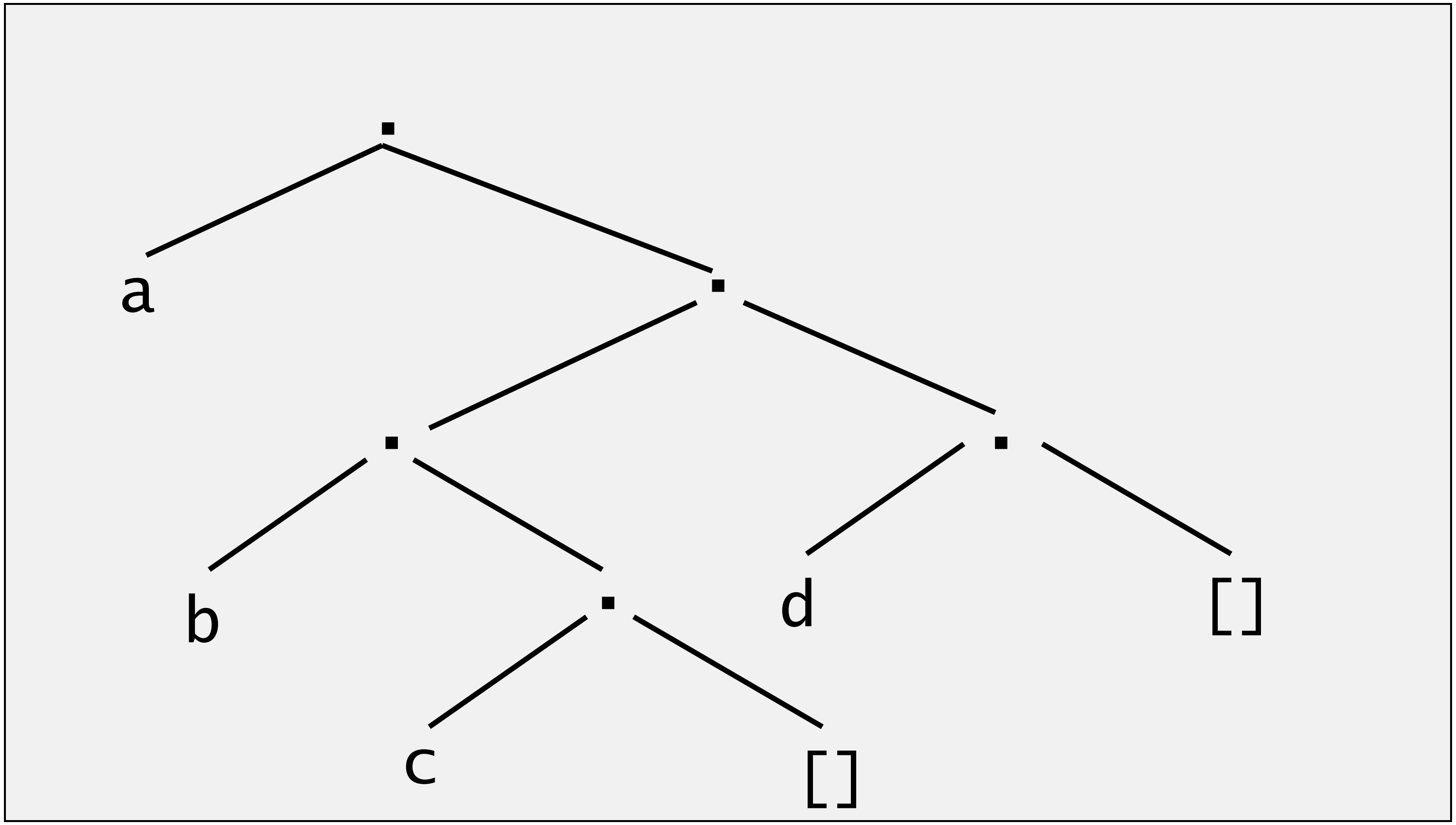
```
?- .(a,[]) == [a].
yes

?- .(f(d,e),[]) == [f(d,e)].
yes

?- .(a,.(b,[])) == [a,b].
yes

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

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

Example of a list as tree

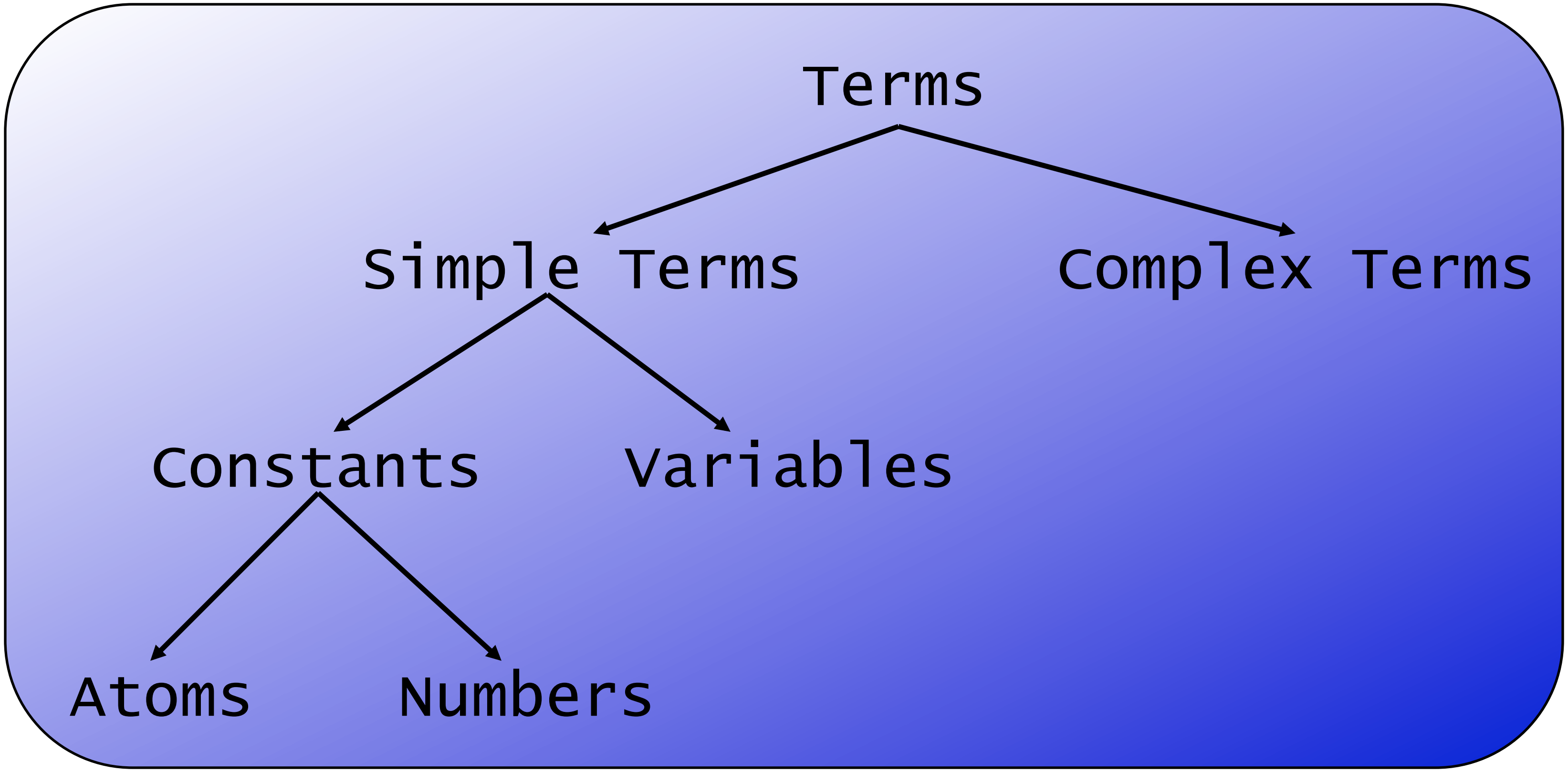
- Example: [a,[b,c],d]



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

Type of terms



Checking the type of a term

atom/1	*Is the argument an atom?*
integer/1	*… an integer?*
float/1	*… a floating point number?*
number/1	*… an integer or float?*
atomic/1	*… a constant?*
var/1	*… an uninstantiated variable?*
nonvar/1	*… an instantiated variable or another term that is not an uninstantiated variable*

Type checking: atom/1

```
?- atom(a).
yes

?- atom(7).
no

?- atom(X).
no
```

Type checking: atom/1

```
?- X=a, atom(X).
X = a
yes

?- atom(X), X=a.
no
```

Type checking: atomic/1

```
?- atomic(mia).
yes

?- atomic(5).
yes

?- atomic(loves(vincent,mia)).
no
```

Type checking: var/1

```
?- var(mia).
no

?- var(X).
yes

?- X=5, var(X).
no
```

Type checking: nonvar/1

```
?- nonvar(X).
no

?- nonvar(mia).
yes

?- nonvar(23).
yes
```

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

The functor/3 predicate

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

```
?- functor(friends(lou,andy),F,A).
   F = friends
   A = 2
   yes
```

The functor/3 predicate

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

```
?- functor(friends(lou,andy),F,A).
   F = friends
   A = 2
   yes

?- functor([lou,andy,vicky],F,A).
   F = .
   A = 2
   yes
```

functor/3 and constants

- What happens when we use functor/3 with constants?

functor/3 and constants

- What happens when we use functor/3 with constants?

```
?- functor(mia,F,A).
   F = mia
   A = 0
   yes
```

functor/3 and constants

- What happens when we use functor/3 with constants?

```
?- functor(mia,F,A).
   F = mia
   A = 0
   yes
?- functor(14,F,A).
   F = 14
   A = 0
   yes
```

functor/3 for constructing terms

- You can also use functor/3 to construct terms:

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

Checking for complex terms

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

Arguments: arg/3

- 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 *N*th argument of *T*

Arguments: arg/3

- 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 *N*th argument of *T*

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

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]
yes
```

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]
yes
```

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 has 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: - (to represent negative numbers)
- Postfix operators
  - Functors written after their argument
  - Example: ++ in the C programming language

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)*3?
- 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 in Prolog

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

Operators in SWI Prolog

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile
1100	xfy	;, |
1050	xfy	->, op*->
1000	xfy	,
954	xfy	\
900	fy	\+
900	fx	~
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	+, -, ?, \
400	yfx	*, /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^