CS-713 (Artificial Intelligence) : Chapter 7

Introduction to PROLOG, Introduction to LISP

Preetpal Kaur Buttar

Department of Computer Science and Engineering

Sant Longowal Institute of Engineering and Technology, Longowal

Introduction to PROLOG

- About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille

- About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille
- PROLOG = PROgrammation en LOGique (PROgramming in LOGic)

- About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille
- PROLOG = PROgrammation en LOGique (PROgramming in LOGic)
- PROLOG is a declarative programming language unlike most common programming languages

- About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille
- PROLOG = PROgrammation en LOGique (PROgramming in LOGic)
- PROLOG is a declarative programming language unlike most common programming languages
- In a declarative language:
  - the programmer specifies a goal to be achieved

- About 1970, PROLOG was designed by A. Colmerauer and P. Roussel at the University of Marseille
- PROLOG = PROgrammation en LOGique (PROgramming in LOGic)
- PROLOG is a declarative programming language unlike most common programming languages
- In a declarative language:
  - the programmer specifies a goal to be achieved
  - the PROLOG system works out how to achieve it

Traditional Programming vs PROLOG

- Traditional programming languages are said to be procedural

Traditional Programming vs PROLOG

- Traditional programming languages are said to be procedural
- Procedural programmer must specify in detail how to solve a problem:
  - mix ingredients;
  - beat until smooth;
  - bake for 20 minutes in a moderate oven;
  - remove tin from oven;
  - put on bench;
  - close oven;
  - turn off oven;

Traditional Programming vs PROLOG

- Traditional programming languages are said to be procedural
- Procedural programmer must specify in detail how to solve a problem:
  - mix ingredients;
  - beat until smooth;
  - bake for 20 minutes in a moderate oven;
  - remove tin from oven;
  - put on bench;
  - close oven;
  - turn off oven;
- In purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system

Applications of PROLOG

- Natural-language processing

Applications of PROLOG

- Natural-language processing
- Compiler construction

Applications of PROLOG

- Natural-language processing
- Compiler construction
- The development of expert systems

Applications of PROLOG

- Natural-language processing
- Compiler construction
- The development of expert systems
- Work in the area of computer algebra

Applications of PROLOG

- Natural-language processing
- Compiler construction
- The development of expert systems
- Work in the area of computer algebra
- The development of (parallel) computer architectures

Applications of PROLOG

- Natural-language processing
- Compiler construction
- The development of expert systems
- Work in the area of computer algebra
- The development of (parallel) computer architectures
- Database systems

Dialects of PROLOG

- There are several dialects of PROLOG in use, such as,
  - C-PROLOG,
  - SWI-PROLOG,
  - Sicstus-PROLOG,
  - LPA-PROLOG

Dialects of PROLOG

- There are several dialects of PROLOG in use, such as,
  - C-PROLOG,
  - SWI-PROLOG,
  - Sicstus-PROLOG,
  - LPA-PROLOG
- You are expected to use SWI-PROLOG: Open-source (GPL) PROLOG environment
  - http://www.swi-prolog.org/
  - Development began in 1987
  - Available for Linux, MacOS X and Windows
  - Fully featured, with many libraries

PROLOG Programs Answer Questions

- Specifying a logic program amounts to:

PROLOG Programs Answer Questions

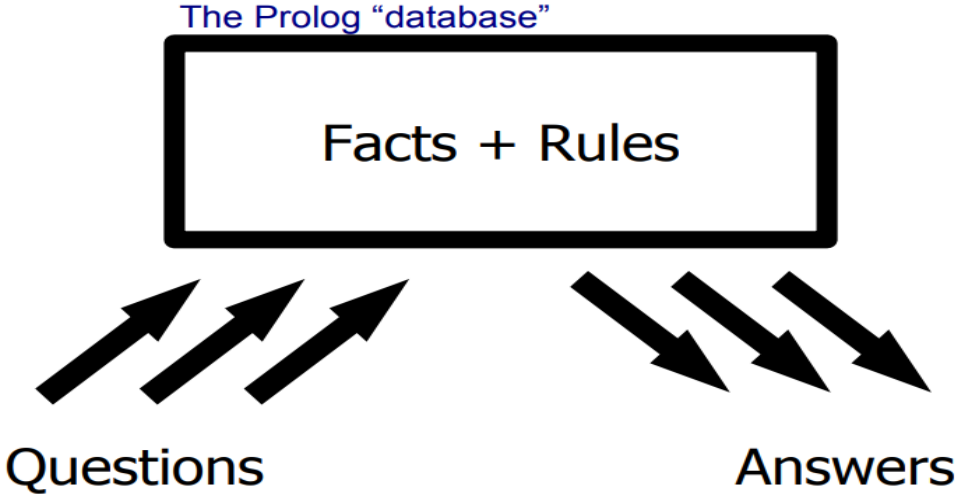
- Specifying a logic program amounts to:
  - Specifying the facts concerning the objects and relations between objects relevant to the problem at hand

PROLOG Programs Answer Questions

- Specifying a logic program amounts to:
  - Specifying the facts concerning the objects and relations between objects relevant to the problem at hand
  - Specifying the rules concerning the objects and their interrelationships

PROLOG Programs Answer Questions

- Specifying a logic program amounts to:
  - Specifying the facts concerning the objects and relations between objects relevant to the problem at hand
  - Specifying the rules concerning the objects and their interrelationships
  - Posing queries concerning the objects and relations.



SWI-PROLOG

- To run PROLOG on the Linux machines, just type pl.

SWI-PROLOG

- To run PROLOG on the Linux machines, just type pl.
- You’ll get a prompt like this: ?-

SWI-PROLOG

- To run PROLOG on the Linux machines, just type pl.
- You'll get a prompt like this: ?-
- To exit PROLOG, type halt. (Fullstop is to be typed in)

PROLOG Terms

- The central data structure in PROLOG is that of a term.

PROLOG Terms

- The central data structure in PROLOG is that of a term.
- There are terms of four kinds: atoms, numbers, variables, and compound terms.

PROLOG Terms

- The central data structure in PROLOG is that of a term.
- There are terms of four kinds: atoms, numbers, variables, and compound terms.
- Atoms and numbers are sometimes grouped together and called atomic terms.

Atoms

- Atoms are usually strings made up of lower- and upper-case letters, digits, and the underscore, starting with a lowercase letter.

Atoms

- Atoms are usually strings made up of lower- and upper-case letters, digits, and the underscore, starting with a lowercase letter.
- The following are all valid PROLOG atoms:

  elephant, b, abcXYZ, x_123, how_are_you_today

Atoms

- Atoms are usually strings made up of lower- and upper-case letters, digits, and the underscore, starting with a lowercase letter.
- The following are all valid PROLOG atoms:

  elephant, b, abcXYZ, x_123, how_are_you_today

- On top of that also any sequence of arbitrary characters enclosed in single quotes denotes an atom.

  ‘This is also a PROLOG atom.’

Atoms

- Atoms are usually strings made up of lower- and upper-case letters, digits, and the underscore, starting with a lowercase letter.
- The following are all valid PROLOG atoms:

  elephant, b, abcXYZ, x_123, how_are_you_today

- On top of that also any sequence of arbitrary characters enclosed in single quotes denotes an atom.

  'This is also a PROLOG atom.'

- Finally, strings made up solely of special characters like + - * = < > : & (check the manual of your PROLOG system for the exact set of these characters) are also atoms.
  - Examples: +, ::, <------>, ***

Numbers

- All PROLOG implementations have an integer type:
  - a sequence of digits, optionally preceded by a - (minus).
- Some also support floats.

Variables

- Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore.

Variables

- Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore.
- Examples:

    X, Elephant, _4711, X_1_2, MyVariable, _

Variables

- Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore.
- Examples:

  X, Elephant, _4711, X_1_2, MyVariable, _
- The last one of the above examples (the single underscore) constitutes a special case:
  - It is called the anonymous variable and is used when the value of a variable is of no particular interest.

Variables

- Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore.
- Examples:

  X, Elephant, _4711, X_1_2, MyVariable, _
- The last one of the above examples (the single underscore) constitutes a special case:
  - It is called the anonymous variable and is used when the value of a variable is of no particular interest.
  - Multiple occurrences of the anonymous variable in one expression are assumed to be distinct, i.e., their values don't necessarily have to be the same.

Compound Terms

- Compound terms are made up of a functor (a PROLOG atom) and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.

Compound Terms

- Compound terms are made up of a functor (a PROLOG atom) and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.
- The following are some examples for compound terms:

```
is_bigger(horse, X)
f(g(X, _), 7)
’My Functor’(dog)
```

Compound Terms

- Compound terms are made up of a functor (a PROLOG atom) and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.
- The following are some examples for compound terms:

  is_bigger(horse, X)

  f(g(X, _), 7)

  ’My Functor’(dog)

- It’s important not to put any blank characters between the functor and the opening parentheses, or PROLOG won’t understand what you’re trying to say.

Compound Terms

- Compound terms are made up of a functor (a PROLOG atom) and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.
- The following are some examples for compound terms:

  is_bigger(horse, X)

  f(g(X, _), 7)

  ’My Functor’(dog)
- It’s important not to put any blank characters between the functor and the opening parentheses, or PROLOG won’t understand what you’re trying to say.
- A term that doesn’t contain any variables is called a ground term.

Clauses, Programs and Queries

- PROLOG programs are made up of facts and rules.
- Facts and rules are also called clauses.
- They are used to define predicates.

Facts

- A fact is a predicate followed by a full stop.

Facts

- A fact is a predicate followed by a full stop.
- Examples:

```
bigger(whale, _).
life_is_beautiful.
```

Facts

- A fact is a predicate followed by a full stop.
- Examples:

  bigger(whale, _).

  life_is_beautiful.
- The intuitive meaning of a fact is that we define a certain instance of a relation as being true.

Rules

- A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).

Rules

- A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).
- Head and body are separated by the symbol :- and, like every PROLOG expression, a rule has to be terminated by a full stop.

Rules

- A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).
- Head and body are separated by the symbol :- and, like every PROLOG expression, a rule has to be terminated by a full stop.
- Examples:

```
is_smaller(X, Y) :- is_bigger(Y, X).
aunt(Aunt, Child) :- sister(Aunt, Parent), parent(Parent, Child).
```

Rules

- A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).
- Head and body are separated by the symbol :- and, like every PROLOG expression, a rule has to be terminated by a full stop.
- Examples:

  is_smaller(X, Y) :- is_bigger(Y, X).

  aunt(Aunt, Child) :- sister(Aunt, Parent), parent(Parent, Child).
- The intuitive meaning of a rule is that the goal expressed by its head is true, if we (or rather the PROLOG system) can show that all of the expressions (subgoals) in the rule's body are true.

Programs

- A PROLOG program is a sequence of clauses.

Queries

- After compilation, a PROLOG program is run by submitting queries to the interpreter.

Queries

- After compilation, a PROLOG program is run by submitting queries to the interpreter.
- A query has the same structure as the body of a rule, i.e., it is a sequence of predicates separated by commas and terminated by a full stop.

Queries

- After compilation, a PROLOG program is run by submitting queries to the interpreter.
- A query has the same structure as the body of a rule, i.e., it is a sequence of predicates separated by commas and terminated by a full stop.
- They can be entered at the PROLOG prompt, which in most implementations looks something like this: ?-.

Queries

- After compilation, a PROLOG program is run by submitting queries to the interpreter.
- A query has the same structure as the body of a rule, i.e., it is a sequence of predicates separated by commas and terminated by a full stop.
- They can be entered at the PROLOG prompt, which in most implementations looks something like this: ?-.
- Examples:

```
?- is_bigger(elephant, donkey).
?- small(X), green(X), slimy(X).
```

Queries

- After compilation, a PROLOG program is run by submitting queries to the interpreter.
- A query has the same structure as the body of a rule, i.e., it is a sequence of predicates separated by commas and terminated by a full stop.
- They can be entered at the PROLOG prompt, which in most implementations looks something like this: ?-.
- Examples:

```
?- is_bigger(elephant, donkey).
?- small(X), green(X), slimy(X).
```

- Intuitively, when submitting a query like the last example above, we ask PROLOG whether all of its three subgoals are provably true, or in other words whether there exists an X such that small(X), green(X), and slimy(X) are all true.

Some Built-in Predicates

- Built-ins can be used in a similar way as user-defined predicates.

Some Built-in Predicates

- Built-ins can be used in a similar way as user-defined predicates.
- The important difference between the two is that a built-in predicate is not allowed to appear as the principal functor in a fact or the head of a rule.

Some Built-in Predicates

- Built-ins can be used in a similar way as user-defined predicates.
- The important difference between the two is that a built-in predicate is not allowed to appear as the principal functor in a fact or the head of a rule.
- This must be so, because using them in such a position would effectively mean changing their definition.

Equality

- Instead of writing expressions such as =(X, Y), we usually write more conveniently X = Y.
- Such a goal succeeds, if the terms X and Y can be matched.

Guaranteed Success and Certain Failure

- Sometimes it can be useful to have predicates that are known to either fail or succeed in any case.

Guaranteed Success and Certain Failure

- Sometimes it can be useful to have predicates that are known to either fail or succeed in any case.
- The predicates fail and true serve exactly this purpose.

Guaranteed Success and Certain Failure

- Sometimes it can be useful to have predicates that are known to either fail or succeed in any case.
- The predicates fail and true serve exactly this purpose.
- Some PROLOG systems also provide the predicate false, with exactly the same functionality as fail.

Consulting Program Files

- Program files can be compiled using the predicate consult/1.

Consulting Program Files

- Program files can be compiled using the predicate consult/1.
- The argument has to be a PROLOG atom denoting the program file you want to compile.

Consulting Program Files

- Program files can be compiled using the predicate consult/1.
- The argument has to be a PROLOG atom denoting the program file you want to compile.
- For example, to compile the file big-animals.pl, submit the following query to PROLOG:

```
?- consult('big-animals.pl').
```

Consulting Program Files

- Program files can be compiled using the predicate consult/1.
- The argument has to be a PROLOG atom denoting the program file you want to compile.
- For example, to compile the file big-animals.pl, submit the following query to PROLOG:

  ?- consult('big-animals.pl').

- If the compilation is successful, PROLOG will reply with Yes. Otherwise a list of errors will be displayed.

Output

- If besides PROLOG's replies to queries, you wish your program to have further output you can use the write/1 predicate.

Output

- If besides PROLOG's replies to queries, you wish your program to have further output you can use the write/1 predicate.
- The argument can be any valid PROLOG term.

Output

- If besides PROLOG's replies to queries, you wish your program to have further output you can use the write/1 predicate.
- The argument can be any valid PROLOG term.
- In the case of a variable, its value will get printed to the screen.

Output

- If besides PROLOG's replies to queries, you wish your program to have further output you can use the write/1 predicate.
- The argument can be any valid PROLOG term.
- In the case of a variable, its value will get printed to the screen.
- Execution of the predicate nl/0 causes the system to skip a line.

Output

- If besides PROLOG's replies to queries, you wish your program to have further output you can use the write/1 predicate.
- The argument can be any valid PROLOG term.
- In the case of a variable, its value will get printed to the screen.
- Execution of the predicate nl/0 causes the system to skip a line.
- Here are two examples:

```
?- write('Hello World!'), nl.
Hello World!
Yes
```

```
?- X = elephant, write(X), nl.
Elephant
 X = elephant
Yes
```

```
?- X = elephant, write(X), nl.
Elephant
 X = elephant
Yes
```

- In the second example, first the variable X is bound to the atom elephant and then the value of X, i.e., elephant, is written on the screen using the write/1 predicate.

```
?- X = elephant, write(X), nl.
Elephant
 X = elephant
Yes
```

- In the second example, first the variable X is bound to the atom elephant and then the value of X, i.e., elephant, is written on the screen using the write/1 predicate.
- After skipping to a new line, PROLOG reports the variable binding(s), i.e., X = elephant.

Checking the Type of a PROLOG Term

```
?- atom(elephant).
Yes
```

Checking the Type of a PROLOG Term

```
?- atom(elephant).
Yes

?- atom(Elephant).
No
```

Checking the Type of a PROLOG Term

```
?- atom(elephant).
Yes

?- atom(Elephant).
No

?- X = f(mouse), compound(X).
X = f(mouse)
Yes
```

Checking the Type of a PROLOG Term

```
?- atom(elephant).
Yes

?- atom(Elephant).
No

?- X = f(mouse), compound(X).
X = f(mouse)
Yes
```

The last query succeeds, because the variable X is bound to the compound term f(mouse) at the time the subgoal compound(X) is being executed.

Help

- Most PROLOG systems also provide a help function in the shape of a predicate, usually called help/1.

Help

- Most PROLOG systems also provide a help function in the shape of a predicate, usually called help/1.
- Applied to a term (like the name of a built-in predicate), the system will display a short description, if available.

Help

- Most PROLOG systems also provide a help function in the shape of a predicate, usually called help/1.
- Applied to a term (like the name of a built-in predicate), the system will display a short description, if available.
- Example:

```
?- help(atom).
atom(+Term)
        Succeeds if Term is bound to an atom.
```

Example Database

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

Example Database

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

```
?- bigger(donkey, dog).
Yes
```

Example Database

bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

?- bigger(donkey, dog).
Yes

?- bigger(monkey, elephant).
No

Example Database

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

```
?- bigger(donkey, dog).
Yes

?- bigger(monkey, elephant).
No

?- bigger(elephant, monkey).
No
```

Example Database

bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).

?- bigger(donkey, dog).
Yes

?- bigger(monkey, elephant).
No

But why?

?- bigger(elephant, monkey).
No

- If you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys.

- If you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys.
- Still, we know that if elephants are bigger than horses, which in turn are bigger than donkeys, which in turn are bigger than monkeys, then elephants also have to be bigger than monkeys.

- If you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys.
- Still, we know that if elephants are bigger than horses, which in turn are bigger than donkeys, which in turn are bigger than monkeys, then elephants also have to be bigger than monkeys.
- In mathematical terms: the bigger-relation is transitive.

- If you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys.
- Still, we know that if elephants are bigger than horses, which in turn are bigger than donkeys, which in turn are bigger than monkeys, then elephants also have to be bigger than monkeys.
- In mathematical terms: the bigger-relation is transitive.
- But this also has not been defined in our program.

- If you check our little program again, you will find that it says nothing about the relationship between elephants and monkeys.
- Still, we know that if elephants are bigger than horses, which in turn are bigger than donkeys, which in turn are bigger than monkeys, then elephants also have to be bigger than monkeys.
- In mathematical terms: the bigger-relation is transitive.
- But this also has not been defined in our program.
- The correct interpretation of the negative answer PROLOG has given is the following: from the information communicated to the system it cannot be proved that an elephant is bigger than a monkey.

- If, however, we would like to receive a positive reply for a query such as bigger(elephant, monkey), then we have to provide a more accurate description of the world.

- If, however, we would like to receive a positive reply for a query such as bigger(elephant, monkey), then we have to provide a more accurate description of the world.
- One way of doing this would be to add the remaining facts, such as bigger(elephant, monkey), to our program.

- If, however, we would like to receive a positive reply for a query such as bigger(elephant, monkey), then we have to provide a more accurate description of the world.
- One way of doing this would be to add the remaining facts, such as bigger(elephant, monkey), to our program.
- For our little example this would mean adding another 5 facts.

- If, however, we would like to receive a positive reply for a query such as bigger(elephant, monkey), then we have to provide a more accurate description of the world.
- One way of doing this would be to add the remaining facts, such as bigger(elephant, monkey), to our program.
- For our little example this would mean adding another 5 facts.
- Clearly too much work and probably not too smart anyway.

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.
- Animal X is bigger than animal Y either if this has been stated as a fact or if there is an animal Z for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y.

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.
- Animal X is bigger than animal Y either if this has been stated as a fact or if there is an animal Z for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y.
- In PROLOG such statements are called rules and are implemented like this:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.
- Animal X is bigger than animal Y either if this has been stated as a fact or if there is an animal Z for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y.
- In PROLOG such statements are called rules and are implemented like this:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

- In these rules :- means something like “if” and the comma between the two terms bigger(X, Z) and is_bigger(Z, Y) stands for “and”.

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.
- Animal X is bigger than animal Y either if this has been stated as a fact or if there is an animal Z for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y.
- In PROLOG such statements are called rules and are implemented like this:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

- In these rules :- means something like “if” and the comma between the two terms bigger(X, Z) and is_bigger(Z, Y) stands for “and”.
- X, Y, and Z are variables, which in PROLOG is indicated by using capital letters.

- The far better solution would be to define a new relation, which we will call is_bigger, as the transitive closure of bigger.
- Animal X is bigger than animal Y either if this has been stated as a fact or if there is an animal Z for which it has been stated as a fact that animal X is bigger than animal Z and it can be shown that animal Z is bigger than animal Y.
- In PROLOG such statements are called rules and are implemented like this:

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

- In these rules :- means something like "if" and the comma between the two terms bigger(X, Z) and is_bigger(Z, Y) stands for "and".
- X, Y, and Z are variables, which in PROLOG is indicated by using capital letters.
- If from now on we use is_bigger instead of bigger in our queries, the program will work as intended:

```
?- is_bigger(elephant, monkey).
Yes
```

- PROLOG still cannot find the fact bigger(elephant, monkey) in its database, so it tries to use the second rule instead.

- PROLOG still cannot find the fact bigger(elephant, monkey) in its database, so it tries to use the second rule instead.
- This is done by matching the query with the head of the rule, which is is_bigger(X, Y).

- PROLOG still cannot find the fact bigger(elephant, monkey) in its database, so it tries to use the second rule instead.
- This is done by matching the query with the head of the rule, which is is_bigger(X, Y).
- When doing so the two variables get instantiated: X = elephant and Y = monkey.

- PROLOG still cannot find the fact bigger(elephant, monkey) in its database, so it tries to use the second rule instead.
- This is done by matching the query with the head of the rule, which is is_bigger(X, Y).
- When doing so the two variables get instantiated: X = elephant and Y = monkey.
- The rule says that in order to prove the goal is_bigger(X, Y) (with the variable instantiations that's equivalent to is_bigger(elephant, monkey)) PROLOG has to prove the two subgoals bigger(X, Z) and is_bigger(Z, Y), again with the same variable instantiations.

- PROLOG still cannot find the fact bigger(elephant, monkey) in its database, so it tries to use the second rule instead.
- This is done by matching the query with the head of the rule, which is is_bigger(X, Y).
- When doing so the two variables get instantiated: X = elephant and Y = monkey.
- The rule says that in order to prove the goal is_bigger(X, Y) (with the variable instantiations that's equivalent to is_bigger(elephant, monkey)) PROLOG has to prove the two subgoals bigger(X, Z) and is_bigger(Z, Y), again with the same variable instantiations.
- This process is repeated recursively until the facts that make up the chain between elephant and monkey are found and the query finally succeeds.

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.
- Suppose we want to know which animals are bigger than a donkey.

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.
- Suppose we want to know which animals are bigger than a donkey.
- The corresponding query would be:

  ?- is_bigger(X, donkey).

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.
- Suppose we want to know which animals are bigger than a donkey.
- The corresponding query would be:

  ?- is_bigger(X, donkey).
- Again, X is a variable.

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.
- Suppose we want to know which animals are bigger than a donkey.
- The corresponding query would be:

  ?- is_bigger(X, donkey).

- Again, X is a variable.
- We could also have chosen any other name for it, as long as it starts with a capital letter.

- Of course, we can do slightly more exciting stuff than just asking yes/no-questions.
- Suppose we want to know which animals are bigger than a donkey.
- The corresponding query would be:

  ?- is_bigger(X, donkey).

- Again, X is a variable.
- We could also have chosen any other name for it, as long as it starts with a capital letter.
- The PROLOG interpreter replies as follows:

  ?- is_bigger(X, donkey).

  X = horse

  - Horses are bigger than donkeys.

- The query has succeeded, but in order to allow it to succeed, PROLOG had to instantiate the variable X with the value horse.

- The query has succeeded, but in order to allow it to succeed, PROLOG had to instantiate the variable X with the value horse.
- If this makes us happy already, we can press Return now and that’s it.

- The query has succeeded, but in order to allow it to succeed, PROLOG had to instantiate the variable X with the value horse.
- If this makes us happy already, we can press Return now and that's it.
- In case we want to find out if there are more animals that are bigger than the donkey, we can press the semicolon key, which will cause PROLOG to search for alternative solutions to our query.

- The query has succeeded, but in order to allow it to succeed, PROLOG had to instantiate the variable X with the value horse.
- If this makes us happy already, we can press Return now and that’s it.
- In case we want to find out if there are more animals that are bigger than the donkey, we can press the semicolon key, which will cause PROLOG to search for alternative solutions to our query.
- If we do this once, we get the next solution X = elephant: elephants are also bigger than donkeys.

- The query has succeeded, but in order to allow it to succeed, PROLOG had to instantiate the variable X with the value horse.
- If this makes us happy already, we can press Return now and that's it.
- In case we want to find out if there are more animals that are bigger than the donkey, we can press the semicolon key, which will cause PROLOG to search for alternative solutions to our query.
- If we do this once, we get the next solution X = elephant: elephants are also bigger than donkeys.
- Pressing semicolon again will return a No, because there are no more solutions:

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

- There are many more ways of querying the PROLOG system about the contents of its database.

- There are many more ways of querying the PROLOG system about the contents of its database.
- We can ask whether there is an animal X that is both smaller than a donkey and bigger than a monkey:

- There are many more ways of querying the PROLOG system about the contents of its database.
- We can ask whether there is an animal X that is both smaller than a donkey and bigger than a monkey:

  ?- is_bigger(donkey, X), is_bigger(X, monkey).

  No

- There are many more ways of querying the PROLOG system about the contents of its database.
- We can ask whether there is an animal X that is both smaller than a donkey and bigger than a monkey:

  ?- is_bigger(donkey, X), is_bigger(X, monkey).

  No

- The (correct) answer is No.

- There are many more ways of querying the PROLOG system about the contents of its database.
- We can ask whether there is an animal X that is both smaller than a donkey and bigger than a monkey:

  ?- is_bigger(donkey, X), is_bigger(X, monkey).

  No
- The (correct) answer is No.
- Even though the two single queries is_bigger(donkey, X) and is_bigger(X, monkey) would both succeed when submitted on their own, their conjunction (represented by the comma) does not.

Matching

- Two terms are said to match if they are either identical or if they can be made identical by means of variable instantiation.

Matching

- Two terms are said to match if they are either identical or if they can be made identical by means of variable instantiation.
- Instantiating a variable means assigning it a fixed value.

Matching

- Two terms are said to match if they are either identical or if they can be made identical by means of variable instantiation.
- Instantiating a variable means assigning it a fixed value.
- Two free variables also match, because they could be instantiated with the same ground term.

- The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.

- The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.
- We could test this in the PROLOG interpreter by submitting the corresponding query to which PROLOG would react by listing the appropriate variable instantiations:

- The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.
- We could test this in the PROLOG interpreter by submitting the corresponding query to which PROLOG would react by listing the appropriate variable instantiations:

```
?- is_bigger(X, dog) = is_bigger(elephant, dog).
X = elephant
Yes
```

- The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.
- We could test this in the PROLOG interpreter by submitting the corresponding query to which PROLOG would react by listing the appropriate variable instantiations:

```
?- is_bigger(X, dog) = is_bigger(elephant, dog).
X = elephant
Yes
```

- The following is an example for a query that doesn't succeed, because X cannot match with 1 and 2 at the same time.

- The terms is_bigger(X, dog) and is_bigger(elephant, dog) match, because the variable X can be instantiated with the atom elephant.
- We could test this in the PROLOG interpreter by submitting the corresponding query to which PROLOG would react by listing the appropriate variable instantiations:

  ?- is_bigger(X, dog) = is_bigger(elephant, dog).

  X = elephant

  Yes
- The following is an example for a query that doesn't succeed, because X cannot match with 1 and 2 at the same time.

  ?- p(X, 2, 2) = p(1, Y, X).

  No

- If, however, instead of X, we use the anonymous variable _, matching is possible, because every occurrence of _ represents a distinct variable.

- If, however, instead of X, we use the anonymous variable _, matching is possible, because every occurrence of _ represents a distinct variable.
- During matching Y is instantiated with 2:

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

- If, however, instead of X, we use the anonymous variable _, matching is possible, because every occurrence of _ represents a distinct variable.
- During matching Y is instantiated with 2:

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

- Another example for matching:

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes
```

- So far so good.

- So far so good.
- But what happens, if matching is possible even though no specific variable instantiation has to be enforced (like in all previous examples)?

- So far so good.
- But what happens, if matching is possible even though no specific variable instantiation has to be enforced (like in all previous examples)?
- Consider the following query:

```
?- X = my_functor(Y).
X = my_functor(_G177)
Y = _G177
Yes
```

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.
- In PROLOG’s output, this is denoted through the use of the variable _G177.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.
- In PROLOG’s output, this is denoted through the use of the variable _G177.
- This variable has been generated by PROLOG during execution time.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.
- In PROLOG's output, this is denoted through the use of the variable _G177.
- This variable has been generated by PROLOG during execution time.
- Its particular name, _G177 in this case, may be different every time the query is submitted.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.
- In PROLOG's output, this is denoted through the use of the variable _G177.
- This variable has been generated by PROLOG during execution time.
- Its particular name, _G177 in this case, may be different every time the query is submitted.
- In fact, what the output for the above example will look like exactly will depend on the PROLOG system you use.

- In this example matching succeeds, because X could be a compound term with the functor my_functor and a non-specified single argument.
- Y could be any valid PROLOG term, but it has to be the same term as the argument inside X.
- In PROLOG's output, this is denoted through the use of the variable _G177.
- This variable has been generated by PROLOG during execution time.
- Its particular name, _G177 in this case, may be different every time the query is submitted.
- In fact, what the output for the above example will look like exactly will depend on the PROLOG system you use.
- For instance, some systems will avoid introducing a new variable (here _G177) and instead simply report the variable binding as X = my_functor(Y).

Programming in PROLOG

- You create a database offline, using a text editor.

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

```
male(charlie).
child_of(charlie, harry).
```

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

```
male(charlie).
child_of(charlie, harry).
```

- You start up PROLOG, and get a prompt.

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

```
male(charlie).
child_of(charlie, harry).
```

- You start up PROLOG, and get a prompt.
- Then you load the database:

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

  ```
  male(charlie).
  child_of(charlie, harry).
  ```

- You start up PROLOG, and get a prompt.
- Then you load the database:

  ```
  ?- consult("my\_db.pl").
  ```

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

  male(charlie).

  child_of(charlie, harry).
- You start up PROLOG, and get a prompt.
- Then you load the database:

  ?- consult("my_db.pl").
- After it's been loaded, you can type queries in at the prompt, and PROLOG will return results for these queries.

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

  male(charlie).

  child_of(charlie, harry).
- You start up PROLOG, and get a prompt.
- Then you load the database:

  ?- consult("my_db.pl").
- After it's been loaded, you can type queries in at the prompt, and PROLOG will return results for these queries.
- For instance:

  ?- male(charlie).

  Yes

  ?-

Programming in PROLOG

- You create a database offline, using a text editor.
- For instance, here's a simple database, which we could save as my_db.pl:

  male(charlie).

  child_of(charlie, harry).
- You start up PROLOG, and get a prompt.
- Then you load the database:

  ?- consult("my_db.pl").
- After it's been loaded, you can type queries in at the prompt, and PROLOG will return results for these queries.
- For instance:

  ?- male(charlie).

  Yes

  ?-
- Syntactically, a query looks just like a fact. But it's interpreted as a question.

How PROLOG Responds to a Query

- Let's say we load my_db.pl into PROLOG:

```
male(charlie).
child_of(charlie, harry).
```

How PROLOG Responds to a Query

- Let's say we load my_db.pl into PROLOG:

  male(charlie).

  child_of(charlie, harry).

- When we ask a query, PROLOG runs through the facts in the database in order, trying to match it to one of them.

How PROLOG Responds to a Query

- Let's say we load my_db.pl into PROLOG:

  male(charlie).

  child_of(charlie, harry).

- When we ask a query, PROLOG runs through the facts in the database in order, trying to match it to one of them.

- If a match is found, PROLOG replies with Yes:

```
?- male(charlie).
Yes
?-
```

How PROLOG Responds to a Query

- Let's say we load my_db.pl into PROLOG:

  male(charlie).

  child_of(charlie, harry).

- When we ask a query, PROLOG runs through the facts in the database in order, trying to match it to one of them.

- If a match is found, PROLOG replies with Yes:

  ?- male(charlie).

  Yes

  ?-

- If no match is found, PROLOG replies with No:

  ?- female(queen_victoria).

  No

  ?-

PROLOG's Search Strategy

- Let's extend the database a bit:

```
child_of(liz, charlie).
child_of(liz, anne).
child_of(liz, andrew).
child_of(charlie, harry).
child_of(charlie, will).
child_of(anne, zara).
```

PROLOG's Search Strategy

- Let's extend the database a bit:

```
child_of(liz, charlie).
child_of(liz, anne).
child_of(liz, andrew).
child_of(charlie, harry).
child_of(charlie, will).
child_of(anne, zara).
```

- PROLOG searches the database of clauses in order (first-to-last), so the first clause it matches will be the first one entered in the database.

```
?- child_of(charlie, X).
X = harry
```

Visualising the Search

- PROLOG basically executes a kind of tree search.

Visualising the Search

- PROLOG basically executes a kind of tree search.
- Each of the system’s actions is an attempt to match a query with one of the database clauses.

Visualising the Search

- PROLOG basically executes a kind of tree search.
- Each of the system’s actions is an attempt to match a query with one of the database clauses.
- The query plus the database define a set of possible states.

Visualising the Search

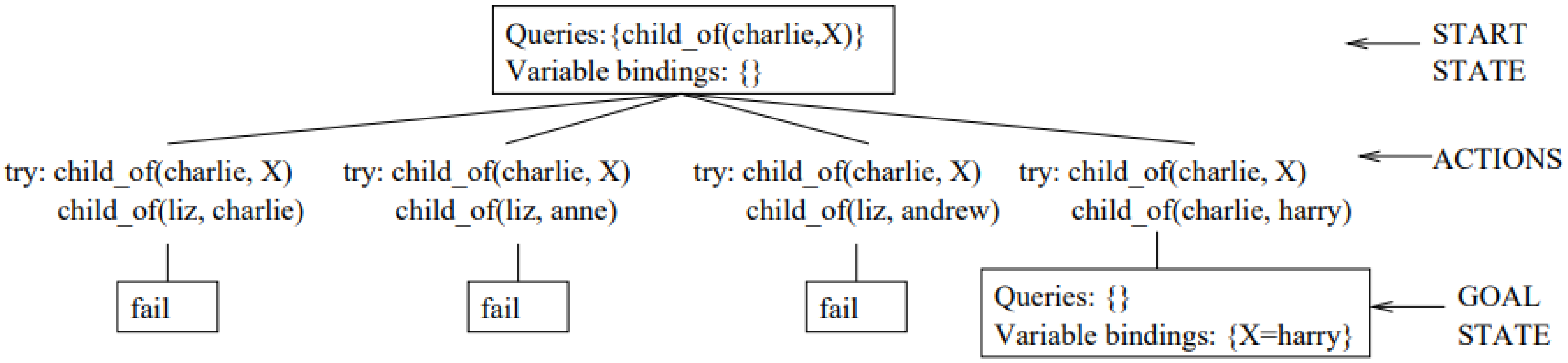
- PROLOG basically executes a kind of tree search.
- Each of the system’s actions is an attempt to match a query with one of the database clauses.
- The query plus the database define a set of possible states.
  - Each state reached by a successful match consists of a list of unresolved queries and a set of variable bindings.

Visualising the Search

- PROLOG basically executes a kind of tree search.
- Each of the system's actions is an attempt to match a query with one of the database clauses.
- The query plus the database define a set of possible states.
    - Each state reached by a successful match consists of a list of unresolved queries and a set of variable bindings.
    - An unsuccessful match results in a special state called fail.

Visualising the Search

- PROLOG basically executes a kind of tree search.
- Each of the system's actions is an attempt to match a query with one of the database clauses.
- The query plus the database define a set of possible states.
  - Each state reached by a successful match consists of a list of unresolved queries and a set of variable bindings.
  - An unsuccessful match results in a special state called fail.
- The goal state is one where the set of unresolved queries is empty.


Queries: {child_of(charlie,X)}
Variable bindings: {}
START STATE
ACTIONS
try: child_of(charlie, X)
child_of(liz, charlie)
try: child_of(charlie, X)
child_of(liz, anne)
try: child_of(charlie, X)
child_of(liz, andrew)
try: child_of(charlie, X)
child_of(charlie, harry)
fail
fail
fail
Queries: {}
Variable bindings: {X=harry}
GOAL STATE

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule's head.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule’s head.
- If the head matches, then the result state is defined as follows:

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule's head.
- If the head matches, then the result state is defined as follows:
  - The query matching the head of the rule is deleted from the list of queries.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule's head.
- If the head matches, then the result state is defined as follows:
  - The query matching the head of the rule is deleted from the list of queries.
  - All the terms in the body of the rule become queries themselves, and are added to the list.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule's head.
- If the head matches, then the result state is defined as follows:
  - The query matching the head of the rule is deleted from the list of queries.
  - All the terms in the body of the rule become queries themselves, and are added to the list.
- Rules thus introduce searches of depth greater than 1.

Query-matching with Rules

- When a query is made, PROLOG searches the clauses in order.
  - If the clause is a fact, PROLOG tries to match the query to it directly.
  - If the clause is a rule, PROLOG tries to match the query to the rule's head.
- If the head matches, then the result state is defined as follows:
  - The query matching the head of the rule is deleted from the list of queries.
  - All the terms in the body of the rule become queries themselves, and are added to the list.
- Rules thus introduce searches of depth greater than 1.
- Note: New queries are added to the front of the list of queries. So PROLOG implements a depth-first search.

An Example

- Consider this simple database:

  child_of(charlie, harry).
  child_of(charlie, will).
  loves(N1, N2) :- child_of(N2, N1).

An Example

- Consider this simple database:

  child_of(charlie, harry).
  child_of(charlie, will).
  loves(N1, N2) :- child_of(N2, N1).

- And the query loves(will, charlie).

An Example

- Consider this simple database:

  child_of(charlie, harry).

  child_of(charlie, will).

  loves(N1, N2) :- child_of(N2, N1).
- And the query loves(will, charlie).
- PROLOG runs through the clauses in order, trying to match each one.

An Example

- Consider this simple database:

  child_of(charlie, harry).

  child_of(charlie, will).

  loves(N1, N2) :- child_of(N2, N1).

- And the query loves(will, charlie).
- PROLOG runs through the clauses in order, trying to match each one.
  - The first two clauses fail directly.

An Example

- Consider this simple database:

  child_of(charlie, harry).

  child_of(charlie, will).

  loves(N1, N2) :- child_of(N2, N1).

- And the query loves(will, charlie).
- PROLOG runs through the clauses in order, trying to match each one.
  - The first two clauses fail directly.
  - The head of the third clause matches, if we bind N1 to charlie and N2 to will.

An Example

- Consider this simple database:

  child_of(charlie, harry).

  child_of(charlie, will).

  loves(N1, N2) :- child_of(N2, N1).
- And the query loves(will, charlie).
- PROLOG runs through the clauses in order, trying to match each one.
  - The first two clauses fail directly.
  - The head of the third clause matches, if we bind N1 to charlie and N2 to will.
  - We now generate a new sub-query to test: child of(charlie, will).

An Example

- Consider this simple database:

  child_of(charlie, harry).

  child_of(charlie, will).

  loves(N1, N2) :- child_of(N2, N1).
- And the query loves(will, charlie).
- PROLOG runs through the clauses in order, trying to match each one.
  - The first two clauses fail directly.
  - The head of the third clause matches, if we bind N1 to charlie and N2 to will.
  - We now generate a new sub-query to test: child of(charlie, will).
  - We test this query against each clause in the database, left-to-right. And this succeeds.

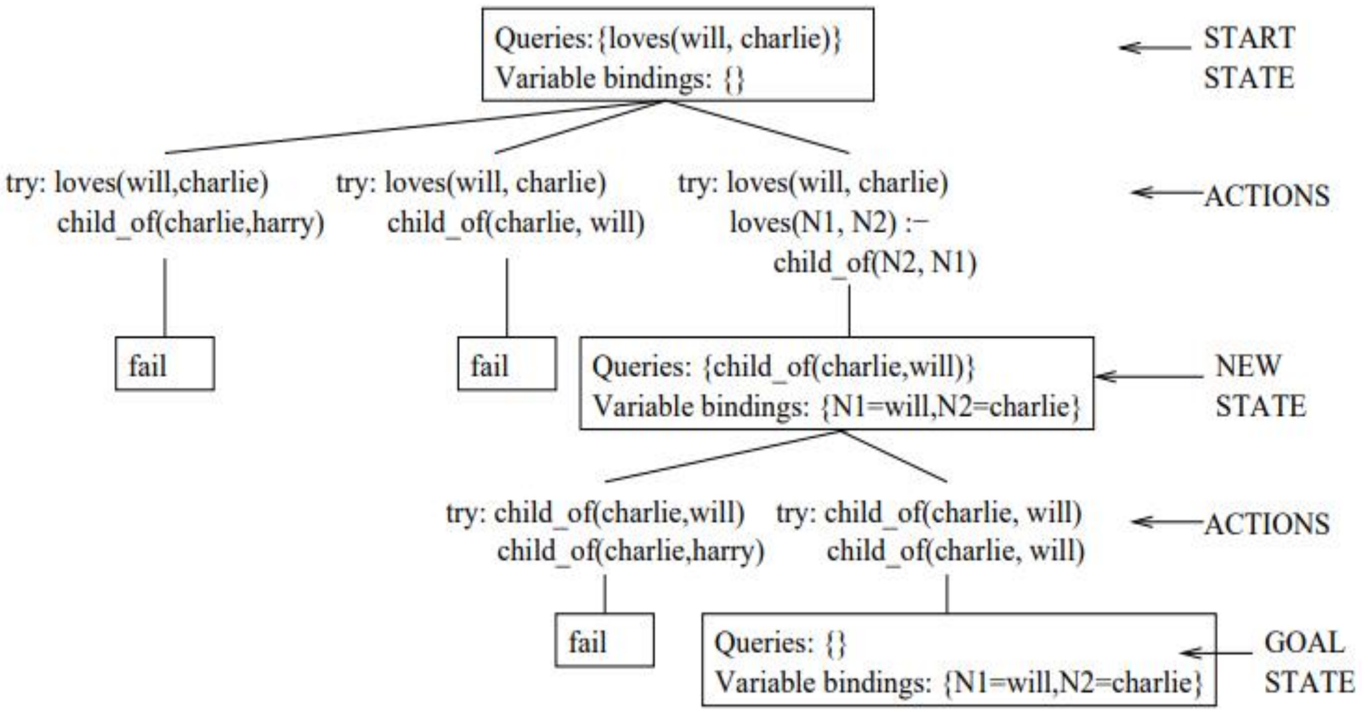
Visualizing the Search

child_of(charlie, harry).

child_of(charlie, will).

loves(N1, N2) :- child_of(N2, N1).

?- loves(will, charlie).



Recursion in PROLOG

- PROLOG rules can be recursive.

Recursion in PROLOG

- PROLOG rules can be recursive.
- Here's an example of a recursive rule for defining 'descendant of':

```
descendant_of(N1, N2) :- child_of(N1, N2).
descendant_of(N1, N2) :- child_of(N1, Nmid), descendant_of(Nmid, N2).
```

Recursion in PROLOG

- PROLOG rules can be recursive.
- Here's an example of a recursive rule for defining 'descendant of':

  descendant_of(N1, N2) :- child_of(N1, N2).

  descendant_of(N1, N2) :- child_of(N1, Nmid), descendant_of(Nmid, N2).
- The first of these rules is the base case.

Recursion in PROLOG

- PROLOG rules can be recursive.
- Here's an example of a recursive rule for defining 'descendant of':

  descendant_of(N1, N2) :- child_of(N1, N2).
  descendant_of(N1, N2) :- child_of(N1, Nmid), descendant_of(Nmid, N2).
- The first of these rules is the base case.
- The second rule is the recursive case.

Recursion in PROLOG

- PROLOG rules can be recursive.
- Here's an example of a recursive rule for defining 'descendant of':

  descendant_of(N1, N2) :- child_of(N1, N2).

  descendant_of(N1, N2) :- child_of(N1, Nmid), descendant_of(Nmid, N2).
- The first of these rules is the base case.
- The second rule is the recursive case.
- Note: The base case always has to appear first!

Introduction to LISP

- LISP (LISt Processing)is the second-oldest high-level programming language after Fortran and has changed a great deal since its early days, and a number of dialects have existed over its history.

- LISP (LISt Processing)is the second-oldest high-level programming language after Fortran and has changed a great deal since its early days, and a number of dialects have existed over its history.
- Today, the most widely known general-purpose LISP dialects are Common LISP and Scheme.

- LISP (LISt Processing)is the second-oldest high-level programming language after Fortran and has changed a great deal since its early days, and a number of dialects have existed over its history.
- Today, the most widely known general-purpose LISP dialects are Common LISP and Scheme.
- LISP was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT).

- LISP (LISt Processing)is the second-oldest high-level programming language after Fortran and has changed a great deal since its early days, and a number of dialects have existed over its history.
- Today, the most widely known general-purpose LISP dialects are Common LISP and Scheme.
- LISP was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT).
- It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively.

Local Environment Setup

- You need the following two softwares available on your computer:
  - Text Editor
  - LISP Executer

Local Environment Setup

- You need the following two softwares available on your computer:
  - Text Editor
  - LISP Executer
- Text editor: This will be used to type your program.
  - Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
  - Source files for LISP programs are typically named with the extension ".lisp".

Local Environment Setup

- You need the following two softwares available on your computer:
  - Text Editor
  - LISP Executer
- Text editor: This will be used to type your program.
  - Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
  - Source files for LISP programs are typically named with the extension ".lisp".
- LISP Executor: CLISP is the GNU Common LISP multi-architectural compiler used for setting up LISP in Windows.
  - Windows version emulates a unix environment using MingW under windows.
  - Installer takes care of this and automatically adds CLISP to Windows PATH variable.

LISP Program Structure

- LISP expressions are called symbolic expressions or s-expressions.

LISP Program Structure

- LISP expressions are called symbolic expressions or s-expressions.
- The s-expressions are composed of three valid objects: atoms, lists and strings.

LISP Program Structure

- LISP expressions are called symbolic expressions or s-expressions.
- The s-expressions are composed of three valid objects: atoms, lists and strings.
- Any s-expression is a valid program.

LISP Program Structure

- LISP expressions are called symbolic expressions or s-expressions.
- The s-expressions are composed of three valid objects: atoms, lists and strings.
- Any s-expression is a valid program.
- LISP programs run either on an interpreter or as compiled code.

LISP Program Structure

- LISP expressions are called symbolic expressions or s-expressions.
- The s-expressions are composed of three valid objects: atoms, lists and strings.
- Any s-expression is a valid program.
- LISP programs run either on an interpreter or as compiled code.
- The interpreter checks the source code in a repeated loop, which is also called the read-evaluate-print loop (REPL).
  - It reads the program code, evaluates it, and prints the values returned by the program.

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11.

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt:

    (+ 7 9 11)

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt:

    (+ 7 9 11)

- LISP returns the result:

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt:

  (+ 7 9 11)

- LISP returns the result:

  27

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt:

  ```
  (+ 7 9 11)
  ```

- LISP returns the result:

  ```
  27
  ```

- If you would like to run the same program as a compiled code, then create a LISP source code file named myprog.lisp and type the following code in it:

  ```
  (write (+ 7 9 11))
  ```

A Simple Program

- Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt:

  ```
  (+ 7 9 11)
  ```

- LISP returns the result:

  27

- If you would like to run the same program as a compiled code, then create a LISP source code file named myprog.lisp and type the following code in it:

  ```
  (write (+ 7 9 11))
  ```

- When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is:

  27

LISP Uses Prefix Notation

- You might have noted that LISP uses prefix notation.

LISP Uses Prefix Notation

- You might have noted that LISP uses prefix notation.
- In the above program the + symbol works as the function name for the process of summation of the numbers.

LISP Uses Prefix Notation

- You might have noted that LISP uses prefix notation.
- In the above program the + symbol works as the function name for the process of summation of the numbers.
- In prefix notation, operators are written before their operands. For example, the expression,

  a * (b + c) / d

LISP Uses Prefix Notation

- You might have noted that LISP uses prefix notation.
- In the above program the + symbol works as the function name for the process of summation of the numbers.
- In prefix notation, operators are written before their operands. For example, the expression,

  a * (b + c) / d

- will be written as:

  (/ (* a (+ b c)) d)

Hello World

```
(write-line "Hello World")
```

Basic Building Blocks in LISP

- LISP programs are made up of three basic building blocks:

Basic Building Blocks in LISP

- LISP programs are made up of three basic building blocks:
  - atom

Basic Building Blocks in LISP

- LISP programs are made up of three basic building blocks:
  - atom
  - list

Basic Building Blocks in LISP

- LISP programs are made up of three basic building blocks:
  - atom
  - list
  - string

Atom

- An atom is a number or string of contiguous characters. It includes numbers and special characters.

Atom

- An atom is a number or string of contiguous characters. It includes numbers and special characters.
- Following are examples of some valid atoms:

  hello-world

  name

  123008907

  hello

  Block#221

  abc123

List

- A list is a sequence of atoms and/or other lists enclosed in parentheses.

List

- A list is a sequence of atoms and/or other lists enclosed in parentheses.
- Following are examples of some valid lists:

  (i am a list)

  (a (a b c) d e fgh)

  (father tom (susan bill joe))

  (sun mon tue wed thur fri sat)

  ( )

String

- A string is a group of characters enclosed in double quotation marks.

String

- A string is a group of characters enclosed in double quotation marks.
- Following are examples of some valid strings:

  " I am a string"

  "a ba c d efg #$%^&!"

  "Please enter the following details :"

  "Hello from ‘Mr. Beans' "

Adding Comments

- The semicolon symbol (;) is used for indicating a comment line.

Adding Comments

- The semicolon symbol (;) is used for indicating a comment line.
- For Example,

```
(write-line "Hello World") ; greet the world
; tell them your whereabouts
(write-line "I am at learning LISP")
```

Adding Comments

- The semicolon symbol (;) is used for indicating a comment line.
- For Example,

  (write-line “Hello World”) ; greet the world

  ; tell them your whereabouts

  (write-line “I am at learning LISP”)
- When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is”

  Hello World

  I am at learning LISP

Some Notable Points

- The basic numeric operations in LISP are +, -, *, and /

Some Notable Points

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45

Some Notable Points

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are case-insensitive, cos 45 or COS 45 are same.

Some Notable Points

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are case-insensitive, cos 45 or COS 45 are same.
- LISP tries to evaluate everything, including the arguments of a function. Only three types of elements are constants and always return their own value:
  - Numbers
  - The letter t, that stands for logical true
  - The value nil, that stands for logical false, as well as an empty list

Use of Single Quotation Mark

- LISP evaluates everything including the function arguments and list members.

Use of Single Quotation Mark

- LISP evaluates everything including the function arguments and list members.
- At times, we need to take atoms or lists literally and don't want them evaluated or treated as function calls.

Use of Single Quotation Mark

- LISP evaluates everything including the function arguments and list members.
- At times, we need to take atoms or lists literally and don't want them evaluated or treated as function calls.
- To do this, we need to precede the atom or the list with a single quotation mark.

- The following example demonstrates this.

- The following example demonstrates this.

  Create a file named main.lisp and type the following code into it:

```
(write-line “single quote used, it inhibits evaluation”)
(write ‘(* 2 3))
(write-line “ ”)
(write-line “single quote not used, so expression evaluated”)
(write (* 2 3))
```

- The following example demonstrates this.

  Create a file named main.lisp and type the following code into it:

  (write-line “single quote used, it inhibits evaluation”)

  (write ‘(* 2 3))

  (write-line “ ”)

  (write-line “single quote not used, so expression evaluated”)

  (write (* 2 3))

- When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is:

  single quote used, it inhibits evaluation

  (* 2 3)

  single quote not used, so expression evaluated

  6

Data Types

- LISP data types can be categorized as.
  - Scalar types – for example, number types, characters, symbols etc.
  - Data structures – for example, lists, vectors, bit-vectors, and strings.

Data Types

- LISP data types can be categorized as.
  - Scalar types – for example, number types, characters, symbols etc.
  - Data structures – for example, lists, vectors, bit-vectors, and strings.
- Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions, in method declarations and some other situations.

Data Types

- LISP data types can be categorized as.
  - Scalar types − for example, number types, characters, symbols etc.
  - Data structures − for example, lists, vectors, bit-vectors, and strings.
- Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions, in method declarations and some other situations.
- The typep predicate is used for finding whether an object belongs to a specific type.

Data Types

- LISP data types can be categorized as.
  - Scalar types − for example, number types, characters, symbols etc.
  - Data structures − for example, lists, vectors, bit-vectors, and strings.
- Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions, in method declarations and some other situations.
- The typep predicate is used for finding whether an object belongs to a specific type.
- The type-of function returns the data type of a given object.

Example

Source Code

```
(defvar x 10)
(defvar y 34.567)
(defvar ch nil)
(defvar bg 11.0e+4)

(print (type-of x))
(print (type-of y))
(print (type-of ch))
(print (type-of bg))
```

Example

Source Code

```
(defvar x 10)
(defvar y 34.567)
(defvar ch nil)
(defvar bg 11.0e+4)

(print (type-of x))
(print (type-of y))
(print (type-of ch))
(print (type-of bg))
```

Output

```
(INTEGER 0 281474976710655)
SINGLE-FLOAT
NULL
SINGLE-FLOAT
```

Macros

Syntax for defining a macro is:

```
(defmacro macro-name (parameter-list))
"Optional documentation string."
body-form
```

Example

- Let us write a simple macro named setTo10, which will take a number and set its value to 10.

```
(defmacro setTo10(num)
(setq num 10)(print num))
(setq x 25)
(print x)
(setTo10 x)
```

Example

- Let us write a simple macro named setTo10, which will take a number and set its value to 10.

```
(defmacro setTo10(num)
(setq num 10)(print num))
(setq x 25)
(print x)
(setTo10 x)
```

- Output:

```
25
10
```

Variables

- Global variables:

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.
- For example:

```
(defvar x 234)
(write x)
```

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.
- For example:

```
(defvar x 234)
(write x)
```

- Output:

```
234
```

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.
- For example:

```
(defvar x 234)
(write x)
```

- Output:

```
234
```

- Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the setq construct.

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.
- For example:

```
(defvar x 234)
(write x)
```

- Output:

```
234
```

- Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the setq construct.
- For Example:

```
(setq x 10)
```

Variables

- Global variables:
- Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.
- Global variables are generally declared using the defvar construct.
- For example:

```
(defvar x 234)
(write x)
```

- Output:

  234

- Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the setq construct.
- For Example:

```
(setq x 10)
```

- The above expression assigns the value 10 to the variable x. You can refer to the variable using the symbol itself as an expression.

- The symbol-value function allows you to extract the value stored at the symbol storage place.

- The symbol-value function allows you to extract the value stored at the symbol storage place.
- For Example:

```
(setq x 10)
(setq y 20)
(format t “x = ~2d y = ~2d ~%” x y)
(setq x 100)
(setq y 200)
(format t “x = ~2d y = ~2d” x y)
```

- The symbol-value function allows you to extract the value stored at the symbol storage place.
- For Example:

  (setq x 10)

  (setq y 20)

  (format t “x = ~2d y = ~2d ~%” x y)

  (setq x 100)

  (setq y 200)

  (format t “x = ~2d y = ~2d” x y)

- Output:

  x = 10 y = 20

  x = 100 y = 200

- Local variables:

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.
- There are two other constructs - let and prog for creating local variables.

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.
- There are two other constructs - let and prog for creating local variables.
- The let construct has the following syntax:

  (let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.
- There are two other constructs - let and prog for creating local variables.
- The let construct has the following syntax:

  (let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)

- where var1, var2, ..., varn are variable names and val1, val2, ..., valn are the initial values assigned to the respective variables.

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.
- There are two other constructs - let and prog for creating local variables.
- The let construct has the following syntax:

  (let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)

- where var1, var2, ..., varn are variable names and val1, val2, ..., valn are the initial values assigned to the respective variables.
- When let is executed, each variable is assigned the respective value and lastly the s-expression is evaluated. The value of the last expression evaluated is returned.

- Local variables:
- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the setq construct.
- There are two other constructs - let and prog for creating local variables.
- The let construct has the following syntax:

  (let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)
- where var1, var2, ..., varn are variable names and val1, val2, ..., valn are the initial values assigned to the respective variables.
- When let is executed, each variable is assigned the respective value and lastly the s-expression is evaluated. The value of the last expression evaluated is returned.
- If you don't include an initial value for a variable, it is assigned to nil.

Example

```
(let ((x 'a) (y 'b)(z 'c))
(format t “x = ~a y = ~a z = ~a” x y z))
```

Example

```
(let ((x 'a) (y 'b)(z 'c))
(format t "x = ~a y = ~a z = ~a" x y z))
```

- Output:

```
x = A y = B z = C
```

Example

(let ((x 'a) (y 'b)(z 'c))

(format t “x = ~a y = ~a z = ~a” x y z))

- Output:

  x = A y = B z = C

- The prog construct also has the list of local variables as its first argument, which is followed by the body of the prog, and any number of s-expressions.

Example

```
(let ((x 'a) (y 'b)(z 'c))
(format t “x = ~a y = ~a z = ~a” x y z))
```

- Output:

  x = A y = B z = C

- The prog construct also has the list of local variables as its first argument, which is followed by the body of the prog, and any number of s-expressions.
- The prog function executes the list of s-expressions in sequence and returns nil unless it encounters a function call named return. Then the argument of the return function is evaluated and returned.

Example

```
(prog ((x '(a b c))(y '(1 2 3))(z '(p q 10)))
(format t “x = ~a y = ~a z = ~a” x y z))
```

Example

```
(prog ((x '(a b c))(y '(1 2 3))(z '(p q 10)))
(format t “x = ~a y = ~a z = ~a” x y z))
```

- Output:

  x = (A B C) y = (1 2 3) z = (P Q 10)

Constants

- Constants are declared using the defconstant construct.

```
(defconstant PI 3.141592)
(defun area-circle(rad)
  (terpri)
  (format t "Radius: ~5f" rad)
  (format t "~%Area: ~10f" (* PI rad rad)))
(area-circle 10)
```

Constants

- Constants are declared using the defconstant construct.

```
(defconstant PI 3.141592)
(defun area-circle(rad)
  (terpri)
  (format t “Radius: ~5f” rad)
  (format t “~%Area: ~10f” (* PI rad rad)))
(area-circle 10)
```

- Output:

```
Radius:  10.0
Area:   314.1592
```