

of **not** and $\backslash=$, which can be read as “for all X, not(X) is true” and “for all X and Y, X is not equal to Y” respectively, do not state true facts. The definitions of not and $\backslash=$ only work in the presence of the cut in the rule above.

- The minimum program in Prolog can be written as

```
min(X,Y,X) :- X <= Y, !.
```

```
min(X,Y,Y) :- X > Y, !.
```

Similar to the use of cut in **not** and $\backslash=$, one might be tempted to write a more efficient minimum:

```
min(X,Y,X) :- X <= Y, !.
```

```
min(X,Y,Y).
```

However, this program is incorrect, since **min(4,7,7)?** will succeed. While tempting, the use of cuts to write logically incorrect but efficient Prolog programs should be avoided.

- If **X** is an uninstantiated variable, then an error condition is reported.
- Predicates are also provided to access parts of compound terms.

```
functor(Term,F,Arity), arg(N,Term,Arg)
```

17.5. Meta-Logical Predicates

- Prolog provides predicates to inspect the state of the program.
- These predicates are outside the scope of first-order logic because they treat logical variables as objects themselves, rather than as representatives of other terms.

```
var(Term) :- Term is an uninstantiated logical variable
nonvar(Term) succeeds if var(Term) fails.
```

- Example: A more efficient grandparent program

```
grandparent(X,Z) :- nonvar(X), parent(X,Y), parent(Y,Z).
grandparent(X,Z) :- nonvar(Z), parent(Y,Z), parent(X,Y).
```

- Prolog provides a meta-variable facility, in which a variable can appear as a goal in a conjunctive goal or in the body of a clause. By the time it is called, it must have been instantiated to a term.

```
X ; Y :- X. (the OR operation)
```

```
X ; Y :- Y.
```

17.6. Cuts and Negation

- Prolog provides a predicate called cut (and written !) for pruning the search tree generated by backtracking. Its purpose is 1) to increase efficiency of search 2) to prevent non-termination, and 3) to provide a form of negation

- Green Cuts: Express determinism by stating that once a choice is made there is no alternative.

```
merge([X|Xs],[Y|Ys],[X|Zs]) :- X<Y, !, merge([Xs],[Y|Ys],Zs).
...
```

- Operational Definition of cut: *The goal succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in.*

- prune all clauses below
- prune all alternative solutions to the goals to the left of the cut.
- no affect on goals to the right.

- Negation: Forcing failure using the cut and the fail predicates.

```
not X :- X,!, fail.
```

```
not X.
```

- also

```
X \= X :- !, fail.
```

```
X \= Y.
```

- Red Cuts: These are cuts that are used for efficiency, based on knowledge about Prolog's evaluation order, but that result in a *logically incorrect* program. Each rule in a logic program states an implication of the head by the (perhaps empty) body. However, the second clauses in the definitions

clauses with the same predicate in the head) fails and control returns to another goal in the computation tree.

17.2.2 Clause Order

- The order in which clauses (rules) occur in a pure Prolog program does not change the structure of the search tree derived from a computation, but only the order in which the branches are traversed.
- Since a computation returns all solution to a query, changing the order of clauses in a program simply changes the order in which solutions are produced.
- However, some branches of the search tree may be *infinite*, so the computation may not terminate.
- If the prolog programmer wants only one solution to a query (or some subset of the set of all solutions) then the order of the clauses has an effect on whether any solution is produced in finite time.

17.2.3 Goal Order

- Goal order is more significant than clause order: It determines the branches of the search tree.
- Goal order can have a tremendous effect on the efficiency of a search. For example,

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

is better for `grandparent(john,A)?`

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

is better for `grandparent(A,sue)?`

- Goal order can determine whether a computation terminates or not!

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

versus

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

in which all computations are non-terminating!

17.3. Arithmetic in Prolog

- Prolog provides arithmetic predicates that utilize the underlying machine's arithmetic capabilities for efficiency, and provides infix syntax for convenience.

```
8+X, 3-2, Y*X, 4/Z
```

- The arguments to the arithmetic predicates must be either numbers or logical variables that have been instantiated by the time the arithmetic is performed. Otherwise, a run-time error will occur.
- A system predicate, variously written as `is` or `:=`, is provided. The right hand side must be an evaluable arithmetic expression. The predicate is satisfied if the right hand side and left hand side unify (after the evaluation of the right hand side).

```
fac(0,1).
```

```
fac(N,F) :- N > 0, N1 is N-1, fac(N1,F1), F is N*F1.
```

17.4. Structure Inspection

- Prolog provides predicates for examining the state and structure of terms.

```
integer(X), atom(x), constant(X) (atoms+integers), compound(X)
```

G22.3110 Programming Languages (Honors)

Lecture 17

Prolog

17.1. Prolog as an approximation of Logic Programming

Prolog is an *approximate* realization of the logic programming model. It departs from the logic programming as follows:

- *Execution Order*: Since logic programming's angelic (i.e. always correct) non-deterministic selection of clauses is unimplementable, Prolog specifies the order in which clauses are chosen during execution and provides backtracking to simulate the angelic non-determinism. Also, the arbitrary choice of the order of resolution of the goals in the body of a clause is replaced by a specific order in Prolog.
- *Meta-logical and Extra-logical Facilities*: Prolog provides facilities for enhancing the expressiveness of the logic programming model. Such facilities include the capability to prune the search for solutions and to examine the state of logical variable.
- *Arithmetic and I/O*: In order to take advantage of the capabilities of the underlying machine, Prolog provides fast arithmetic and I/O. Restrictions must be placed on the kinds of expressions using these capabilities (for example, an operand of an arithmetic operator cannot be an uninstantiated logical variable)

For now, we consider pure prolog, in which the only departure from logic programming is the choice of a particular (deterministic) execution order.

17.2. The Execution Order of Prolog

- Prolog's execution mechanism departs from the interpreter presented previously in the following manner:
 - The leftmost goal of the resolvent is chosen, instead of an arbitrary one.
 - The non-deterministic choice of a clause that unifies with the goal is replaced by a sequential search, determined by the order in which the clauses were written, for a unifiable clause and backtracking.
- Thus, Prolog adopts a stack-scheduling algorithm, in which the resolvent is a stack of goals. Execution proceeds by popping a goal off of the resolvent stack. The first clause whose head unifies with the goal is chosen and the goals of the body of the clause are pushed onto the stack (with the leftmost goal being pushed last).
- If no clause can be unified with the popped goal, then the computation is unwound to the last choice of clause made and the next unifiable clause is found. This is backtracking.

17.2.1 The Search Tree

A computation of a goal G with respect to a prolog program P is the generation of *all* solutions of G. Thus, a Prolog computation on a goal G is a complete depth first traversal of the search tree generated by always choosing the leftmost goal of the resolvent.

EXAMPLE: GRANDFATHER? ANCESTOR?

- Shallow backtracking occurs when the unification of a clause and a goal fails, and the next clause is chosen. Deep backtracking occurs when the unification of the last clause of a procedure (i.e. set of

```

append([a,b],Y,[a,b,c,d])?
yes, Y = [c,d]
append(X,Y,[a,b,c,d])?
yes, X = [], Y = [a,b,c,d]
      X = [a], Y = [b,c,d]      /* if more solutions are desired */
      X = [a,b], Y = [c,d]

```

- This example highlights the *two-way* nature of logical variables, that unification provides. Depending on the query, a logical variable in the head of a clause can serve as an input parameter or output parameter.

16.2.12 Termination and Order of Evaluation

Consider the following program:

```

parent(tom,mary).
parent(mary,sue).
parent(mary,john).

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

```

One answer to the query:

```
ancestor(A,B)?
```

is

```
yes, A = tom, B = sue.
```

Notice that if the clauses were chosen in the order in which they were written, and the goals in the bodies were executed from right to left, then the ancestor program would never terminate. But in logic programming, at least theoretically, if a query is a logical consequent of the program then a solution will be found. As we said before, clauses and goals are chosen in a non-deterministic fashion that guarantees finding a solution if one exists.

16.2.13 Negation in Logic Programming

- Logic Programs describe what is true. Untrue facts are simply omitted. Thus, logic programming must be extended in order to provide negation.
- Negation can be added to logic programming by introducing a **not** relation (that is only a partial form of negation in logic). A goal **not** G is a consequence of a program P if G is not a consequence of G.
- Negation is characterized by *failure*, that is **not** G succeeds if G fails.
- As we shall see, negation in Prolog must be handled in a special way.

A Logic Program Interpreter

Input: A logic program P and a goal G

Output: $G\theta$, the instance of G deduced P, or *failure*.

Algorithm:

Initialize the resolvent to be G, the input goal.

While the resolvent is not empty do

Choose a goal A from the resolvent and a (renamed) clause

$A' \leftarrow B_1, B_2, \dots, B_n$

from P such that A and A' unify with mgu θ (exit if no such goal and clause exist).

Remove A from the resolvent and add B_1, B_2, \dots, B_n to the resolvent.

Apply θ to the resolvent and to G.

end while

If the resolvent is empty then output G, else output *failure*.

- Since variables are local to a clause, variable name conflicts may occur if the resolvent has some variables names in common with a clause that is being unified (if possible) a goal **A** chosen from the resolvent. To avoid this, the variables in the clause can be substituted by new variables (that do not occur in the resolvent) before the unification step. For example, given the logic program:

teacher(goldberg, X).

which states that goldberg teaches all students, and the query

teacher(X, jones)?

which asks if there is someone who teaches Jones, the unification will fail unless one of the **X**'s is renamed to another logical variable name.

- Notice that how a goal **A** is chosen, and which clause $A' \leftarrow B_1, B_2, \dots, B_n$ is chosen is not specified. In fact, logic programming says that the goals and clauses are chosen *non-deterministically* in such a manner to insure that an instance of G deduced from P can be found (if there is one). That is to say, the logic program interpreter always chooses the *correct* goal from the resolvent and the *correct* clause to unify with it! As we shall see, Prolog must depart from this rule, since it is unimplementable.

16.2.11 List Structure

- As a syntactic convenience, **cons(X, Y)** is typically written as **[X|Y]**, **cons(a, cons(b, nil))** is written **[a,b,c]**, and **[]** denotes the empty list.
- Here is the code for append:

```
append([X|Xs], Ys, [X|Z]) :- append(Xs, Ys, Z).
append([], Ys, Ys).
```

Notice that append can be used in several ways:

```
append([a,b], [c,d], Z)?
yes, Z = [a,b,c,d]
```

A Unification Algorithm

Input: Two terms T_1 and T_2 to be unified

Output: θ , the most general unifier of T_1 and T_2 , or *failure*.

Algorithm:

Initialize the substitution θ to be empty, the stack to contain the equation $T_1 = T_2$, and failure to *false*.

While stack not empty and no failure do

pop $X = Y$ from the stack

case

X is a variable that does not occur in Y :

substitute Y for X in the stack and in θ .

add $X = Y$ to θ .

Y is a variable that does not occur in X :

substitute X for Y in the stack and in θ .

add $Y = X$ to θ .

X and Y are identical constants or variables:

continue

X is $f(X_1, \dots, X_n)$ and Y is $f(Y_1, \dots, Y_n)$ for some functor f and $n > 0$:

push $X_i = Y_i$, $1 \leq i \leq n$, on the stack.

otherwise:

failure := true;

end while

if failure, then output *failure*

else output θ .

- A logic program interpreter is presented below. It uses the unification algorithm to unify goals with the heads of clauses.

- A is called the head and the B_i 's comprise the body of the rule.
- The above rule can be interpreted as follows: *A is true if B_1, B_2, \dots, B_n are all true.*
- Example:

teaches(X,Y) ← teacher(X,Z), student(Y,Z).

Definition (Law of Universal Modus Ponens): From the rule

$$R = A \leftarrow B_1, B_2, \dots, B_n$$

and the facts B'_1 .

$$B'_2.$$

...

$$B'_n.$$

A' can be deduced if

$$A' \leftarrow B'_1, B'_2, \dots, B'_n$$

is an instance of R.

- Rules, facts, and queries are also called Horn Clauses (or clauses). A fact is a unit clause and a rule with a single goal in the body is called an iterative clause.
- A clause may be recursive, i.e. the functor in the head may occur in the body.

Ancestor(X,Y) ← Parent(X,Z), Ancestor(Z,Y).

16.2.9 Logic Programs

- A logic program is a finite set of rules.
- An existentially quantified goal G is a logical consequence of a program P if there is a clause in P with a ground instance $A \leftarrow B_1, B_2, \dots, B_n$ such that B_1, B_2, \dots, B_n are logical consequences of P and A is an instance of G.
- The meaning of a logic program P, M(P), is the set of ground unit goals deducible from P.

16.2.10 Unification and Logic Program Interpretation

- A term s is more general than a term t if t is an instance of s but s is not an instance of t . A term s is an alphabetic variant of a term t if both s is an instance of t and t is an instance of s . Alphabetic variants can be converted to one another simply by renaming variables.
- A unifier of two terms t_1 and t_2 is a substitution θ making the terms identical, $t_1\theta = t_2\theta$. If a unifier of two terms exists, then the terms are said to unify.
- A most general unifier (or mgu) of two terms t_1 and t_2 is a substitution θ that unifies t_1 and t_2 such that the common instance $t_1\theta$ is more general than any other common instance of t_1 and t_2 . That is, for any other unifier θ' of t_1 and t_2 , $t_1\theta$ is more general than $t_1\theta'$. If two terms unify, there is a unique most general unifier (up to the renaming of variables).
- A unification algorithm computes the most general unifier for two terms. An algorithm for unification is as follows:

16.2.5 Instances and Substitutions

- A term containing no variable is said to be ground.
- A substitution is a finite set (possibly empty) of pairs of the form $X_i = t_i$ where X_i is a variable and t_i is a term, and $X_i \neq X_j$ for every $i \neq j$ and X_i does not occur in t_j for any i and j .
 - The result of applying a substitution θ to a term A , denoted by $A\theta$, is the term obtained by replacing every occurrence of X in A by t , for each pair $X = t$ in θ .
- A term A is an instance of a term B if there is a substitution θ such that $A = B\theta$.
 - `In_WWH(pl_class)` is an instance of `In_WWH(X)`, in which $\theta = \{X = \text{pl_class}\}$
- A term A is a common instance of terms B and C if there are substitutions θ_1 and θ_2 such that $A = B\theta_1$ and $A = C\theta_2$.

16.2.6 Existential vs. Universal Quantification of Logical Variables

- In queries, logical variables are existentially quantified.

`student(smith, Y)?`

is really

$\exists Y. \text{student}(\text{smith}, Y)?$

Answer will supply one or more instances of the query.

- In facts, logical variables are universally quantified.

`fail(X, pl_class).`

is really

$\forall X. \text{fail}(X, \text{pl_class}).$

- Existentially quantified queries are answered by universally quantified facts.
- A term C is a common instance of terms A and B if its in an instance of A and an instance of B . In other words, if there are substitutions θ_1 and θ_2 such that $C = A\theta_1$ and $C = B\theta_2$.
- To answer a query using a fact, find the common instance.

16.2.7 Conjunctive Queries

- A query can consist of a conjunction of goals

`student(Z, pl_class), student(Z, algorithms_class)?`

means “Is there a Z that is a student in both the `pl_class` and the `algorithms_class`?”

- A single substitution is applied to the whole conjunctive query (so all occurrences of each logical variable get replaced by the same term). To satisfy the query in a program P , each goal must have a common instance with a fact in P .

16.2.8 Rules and Horn Clauses

- A rule is a statement of the form

$A \leftarrow B_1, B_2, \dots, B_n$

where $n \geq 0$ and A and each B_i is a goal.

G22.3110 Programming Languages (Honors)

Lecture 16

Logic Programming

16.1. History

- Robinson (1965): Unification and Resolution
- Kowalski (1974): Procedural Interpretation of Horn Clauses (using Robinson's unification)
- Colmerauer: Prolog, a theorem prover, embodied Kowalski's procedural interpretation

16.2. Basic Constructs

16.2.1 Facts

- State a relationship between objects.

```
teacher(goldberg,pl_class).
teacher(cole,algorithms_class).
student(smith,pl_class).
student(smith,algorithms_class).
```

- Terminated with the "." The expression before the "." is called a goal.

16.2.2 Queries

- Asks if a relation holds between objects. Answer is yes or no.

```
teacher(cole,pl_class)?
no
teacher(goldberg,pl_class)?
yes
```

- Syntactically differentiated from facts by the "?", again the expression before the "?" is a goal.

16.2.3 Logical Variables

- Represents an unspecified object. Generally starts with a capital letter.

```
teacher(X,pl_class)?           "Does there exist a teacher of the pl_class?"
yes, X=goldberg
In_WWH(X).                    "All objects are in WWH"
student(Y,algorithms_class)   "All students take the algorithms class"
```

16.2.4 Terms

- A term is the syntactic unit of a logic program, as well is the basic data structure.

term = constant | variable | functor(term, term, ..., term)

- Examples:

```
schonberg, X, cons(a,b), cons(A, cons(B, cons(C, nil)))
```