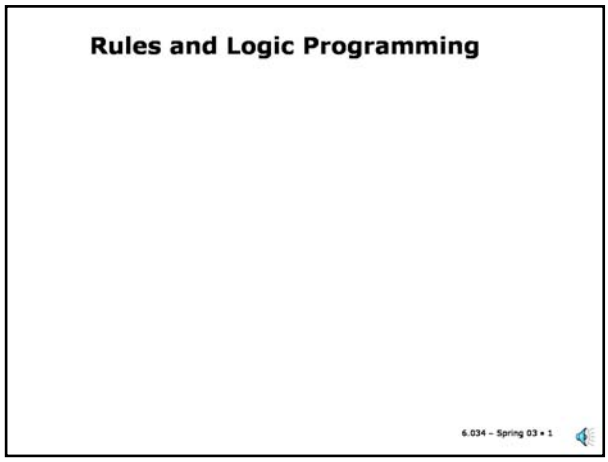
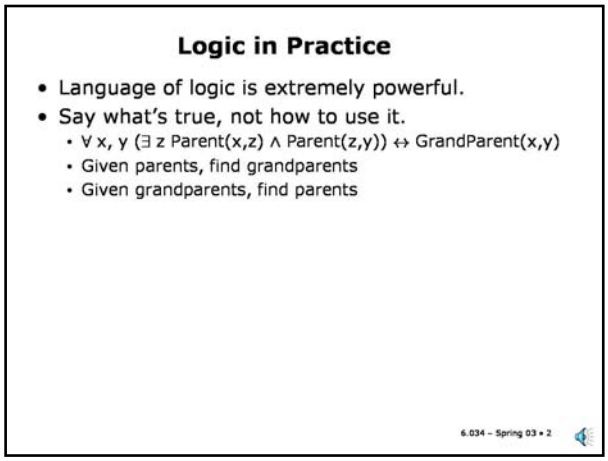
6.034 Notes: Section 11.1

Slide 11.1.1
We've now spent a fair bit of time learning about the language of first-order logic and the mechanisms of automatic inference. And, we've also found that (a) it is quite difficult to write first-order logic and (b) quite expensive to do inference. Both of these conclusions are well justified. Therefore, you may be wondering why we spent the time on logic.

We can motivate our study of logic in a variety of ways. For one, it is the intellectual foundation for all other ways of representing knowledge about the world. As we have already seen, the Web Consortium has adopted a logical language for its Semantic Web project. We also saw that airlines use a language not unlike FOL to describe fare restrictions. We will see later when we talk about natural language understanding that logic also plays a key role.

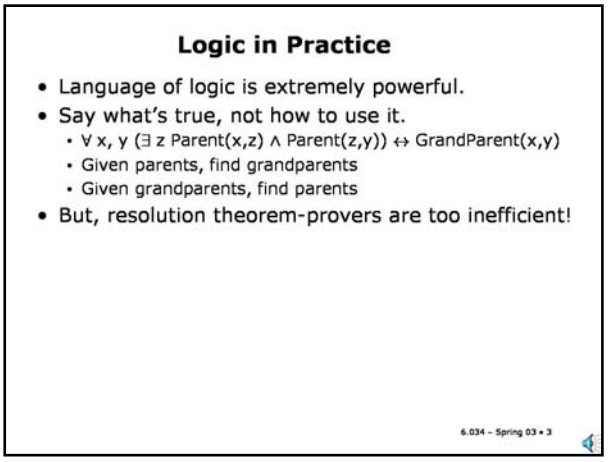
There is another practical application of logic that is reasonably widespread namely **logic programming**. In this section, we will look briefly at logic programming. Later, when we study natural language understanding, we will build on these ideas.

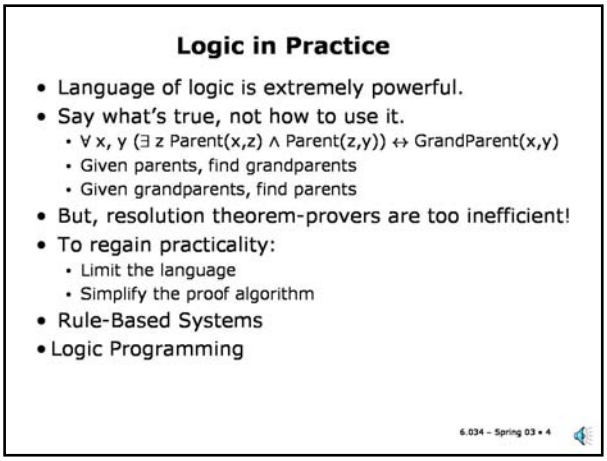




Slide 11.1.2
We have seen that the language of logic is extremely general, with much of the power of natural language. One of the key characteristics of logic, as opposed to programming languages but like natural languages, is that in logic you write down what's true about the world, without saying how to use it. So, for example, one can characterize the relationship between parents and grandparents in this sentence without giving an algorithm for finding the grandparents from the grandchildren or a different algorithm for finding the grandchildren given the grandparents.

Slide 11.1.3
However, this very power and lack of specificity about algorithms means that the general methods for performing computations on logical representations (for example, resolution refutation) are hopelessly inefficient for most practical problems.



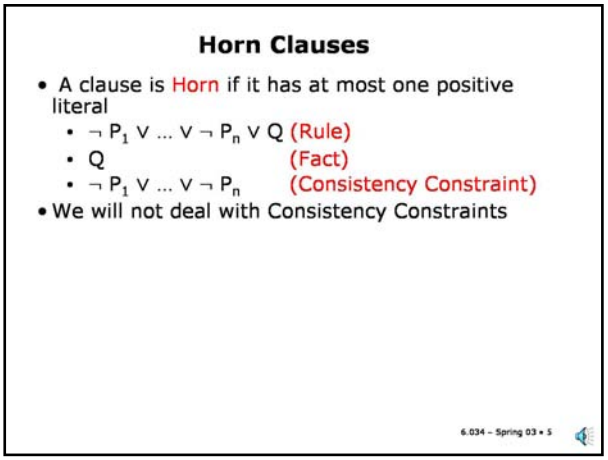


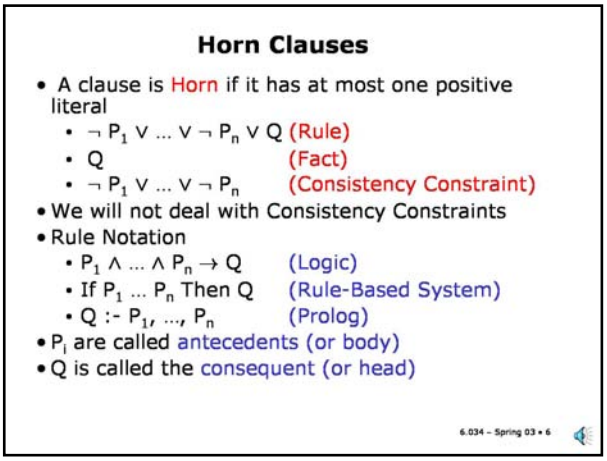
Slide 11.1.4
There are, however, approaches to regaining some of the efficiency while keeping much of the power of the representation. These approaches involve both limiting the language as well as simplifying the inference algorithms to make them more predictable. Similar ideas underlie both **logic programming** and **rule-based systems**. We will bias our presentation towards logic programming.

Slide 11.1.5
In logic programming we will also use the clausal representation that we derived for resolution refutation. However, we will limit the type of clauses that we will consider to the class called **Horn clauses**. A clause is Horn if it has at most one positive literal. In the examples below, we show literals without variables, but the discussion applies both to propositional and first order logic.

There are three cases of Horn clauses:

- A **rule** is a clause with one or more negative literals and exactly one positive literal. You can see that this is the clause form of an implication of the form P_1 ^ ... ^ P_n -> Q, that is, the conjuction of the P's implies Q.
- A **fact** is a clause with exactly one positive literal and no negative literals. We generally will distinguish the case of a **ground fact**, that is, a literal with no variables, from the general case of a literal with variables, which is more like an unconditional rule than what one would think of as a "fact".
- In general, there is another case, known as a **consistency constraint** when the clause has no positive literals. We will not deal with these further, except for the special case of a **conjunctive goal clause** which will take this form (the negation of a conjuction of literals is a Horn clause with no positive literal). However, goal clauses are not rules.

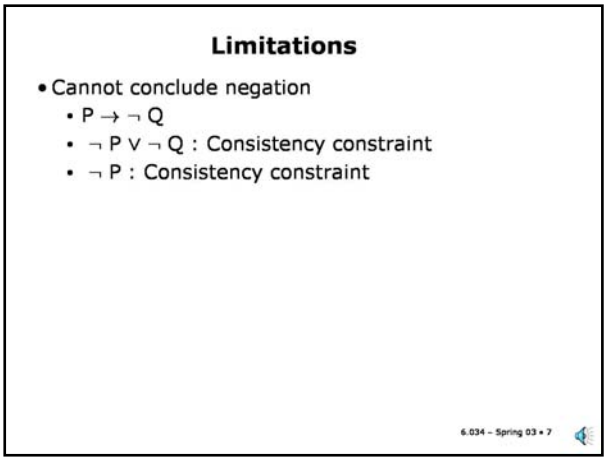


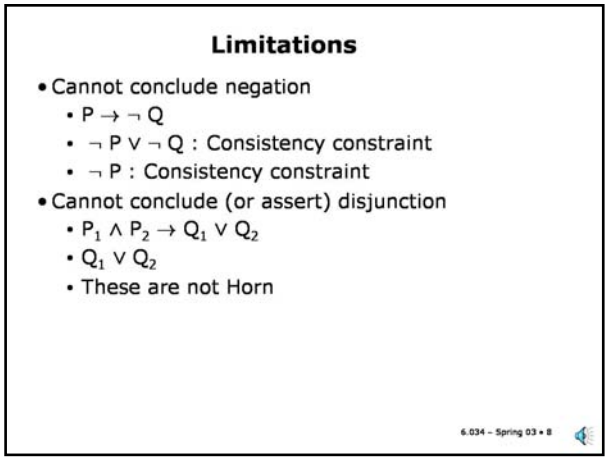


Slide 11.1.6
There are many notations that are in common use for Horn clauses. We could write them in standard logical notation, either as clauses, or as implications. In rule-based systems, one usually has some form of equivalent "If-Then" syntax for the rules. In Prolog, which is the most popular logic programming language, the clauses are written as a sort of reverse implication with the ":-" instead of "<-".

We will call the Q (positive) literal the **consequent** of a rule and call the P_i (negative) literals the **antecedents**. This is terminology for implications borrowed from logic. In Prolog it is more common to call Q the **head** of the clause and to call the P literals the **body** of the clause.

Slide 11.1.7
Note that not every logical statement can be written in Horn clause form, especially if we disallow clauses with zero positive literals (consistency constraints). Importantly, one cannot have a negation on the right hand side of an implication. That is, we cannot have rules that conclude that something is not true! This is a reasonably profound limitation in general but we can work around it in many useful situations, which we will discuss later. Note that because we are not dealing with consistency constraints (all negative literals) we will not be able to deal with negative facts either.



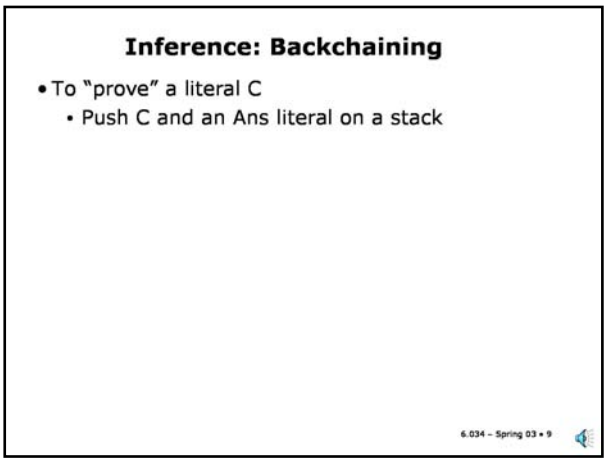


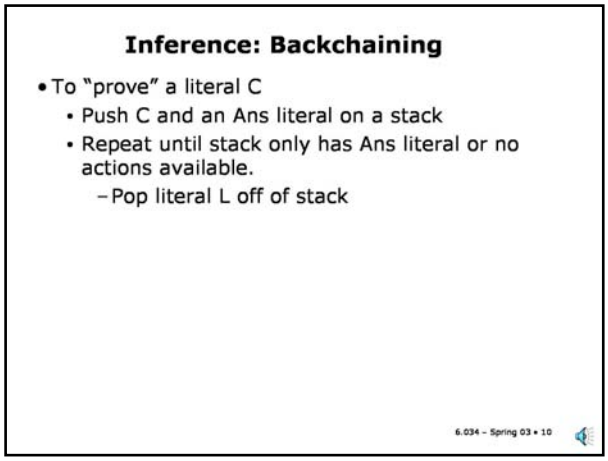
Slide 11.1.8
Similarly, if we have a disjuction on the right hand side of an implication, the resulting clause is not Horn. In fact, we cannot assert a disjunction with more than one positive literal or a disjuction of all negative literals. The former is not Horn while the latter is a consistency constraint.

Slide 11.1.9
It turns out that given our simplified language, we can use a simplified procedure for inference, called **backchaining**, which is basically a generalized form of Modus Ponens (one of the "natural deduction" rules we saw earlier).

Backchaining is relatively simple to understand given that you've seen how resolution works. We start with a literal to "prove", which we call C. We will also use Green's trick (as in Chapter 6.3) to keep track of any variable bindings in C during the proof.

We will keep a stack (first in, last out) of goals to be proved. We initialize the stack to have C (first) followed by the Answer literal (which we write as `Ans`).



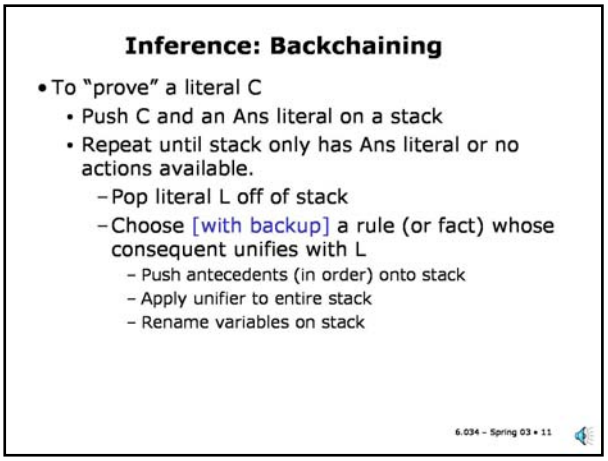


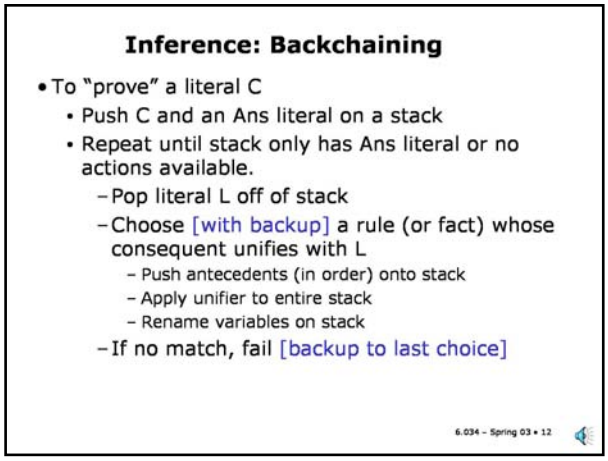
Slide 11.1.10
The basic loop is to pop a literal (L) off the stack until either (a) only the Ans literal remains or (b) there are no further actions possible. The first case corresponds to a successful proof; the second case represents a failed proof.

A word of warning. This loop does not necessarily terminate. We will see examples later where simple sets of rules lead to infinite loops.

Slide 11.1.11
Given a literal L, we look for a fact that unifies with L or a rule whose consequent (head) unifies with L. If we find a match, we push the antecedent literals (if any) onto the stack, apply the unifier to the entire stack and then rename all the variables to make sure that there are no variable conflicts in the future. There are other ways of dealing with the renaming but this one will work.

In general, there will be more than one fact or rule that could match L; we will pick one now but be prepared to come back to try another one if the proof doesn't work out. More on this later.

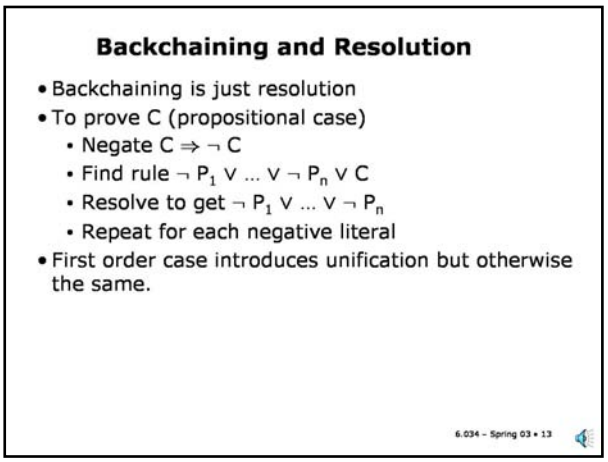


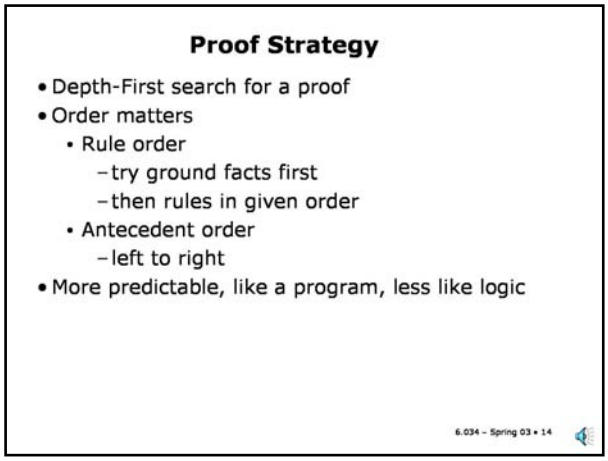


Slide 11.1.12
If no match can be found for L, we fail and backup to try the last choice that has other pending matches.

Slide 11.1.13
If you think about it, you'll notice that backchaining is just our familiar friend, resolution. The stack of goals can be seen as negative literals, starting with the negated goal. We don't actually show literals on the stack with explicit negation but they are implicitly negated.

At every point, we pair up a negative literal from the stack with a positive literal (the consequent) from a fact or rule and add the remaining negative literals (the antecedents) to the stack.





Slide 11.1.14
When we specified backchaining we did it with a particular search algorithm (using the stack), which is basically depth-first search. Furthermore, we will assume that the facts and rules are examined in the order in which they occur in the program. Also that literals from the body of a rule are pushed onto the stack in reverse order, os that the one that occurs first in the body will be the first popped off the stack.

Given these ordering restrictions, it is much easier to understand what a logic program will do. On the other hand, one must understand that what it will do is not what a general theorem prover would do with the same rules and facts.

Slide 11.1.15
Time for an example. Let's look at the following database of facts and rules. The first two entries are ground facts, that A is Father of B and B is Mother of C. The third entry defines a grandparent rule that we would write in FOL as:

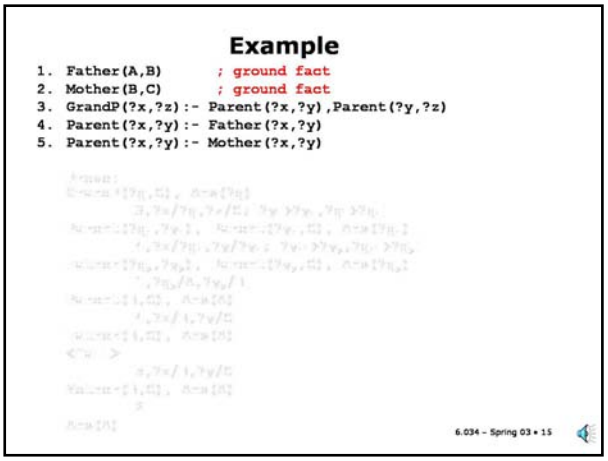
```
@x . @y. @z. P(x,y) ^ P(y,z) -> GrandP(x,z)
```

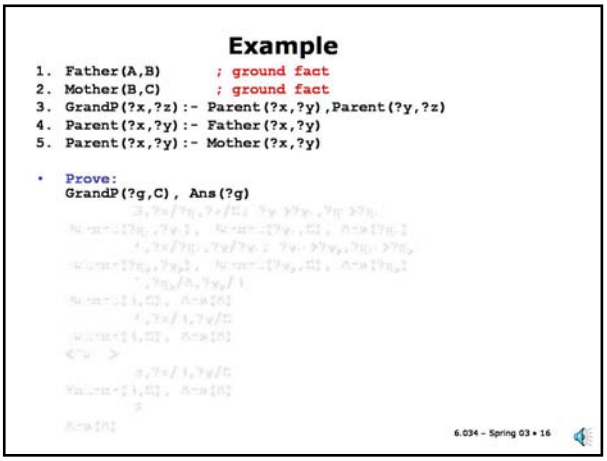
Our rule is simply this rule written with the implication pointing "backwards". Also, our rule language does not have quantifiers; all the variables are implicitly universally quantified.

In our rule language, we will modify our notational conventions for FOL. Instead of identifying constants by prefixing them with $, we will indicate variables by prefixing them with ?. The rationale for this is that in our logic examples we had lots more variables than constants, but that will be different in many of our logic-programming examples.

The next two rules specify that a Father is a Parent and a Mother is a parent. In usual FOL notation, these would be:

```
@x . @y. F(x,y) -> P(x,y)
@x . @y. M(x,y) -> P(x,y)
```





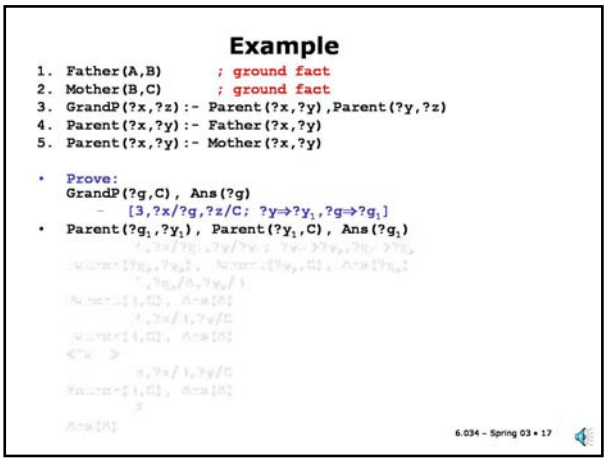
Slide 11.1.16
Now, we set out to find the Grandparent of C. With resolution refutation, we would set out to derive a contradiction from the negation of the goal:

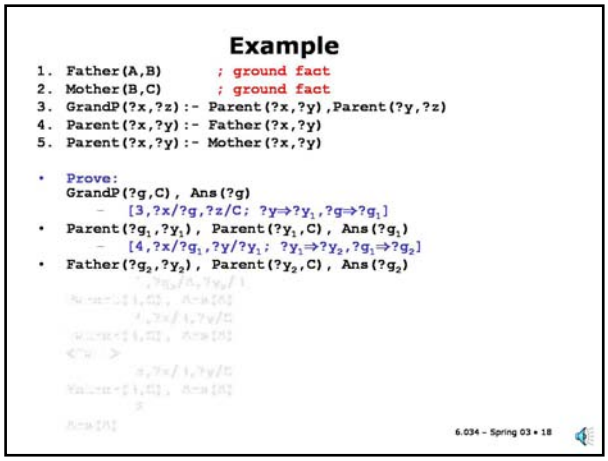
```
~ ]g . GrandP(g,C)
```

whose clause form is `~GrandP(g,C)`. The list of literals in our goal stack are implicitly negated, so we start with `GrandP(g,C)` on the stack. We have also added the Ans literal with the variable we are interested in, ?g, hopefully the name of the grandparent.

Now, we set out to find a fact or rule consequent literal in the database that matches our goal literal.

Slide 11.1.17
You can see that the grandparent goal literal unifies with the consequent of rule 3 using the unifer `{ ?x/?g, ?z/C }`. So, we push the antecedents of rule 3 onto the stack, apply the unifier and then rename all the remaining variables, as indicated. The resulting goal stack now has two Parent literals and the Ans literal. We proceed as before by popping the stack and trying to unify with the first Parent literal.

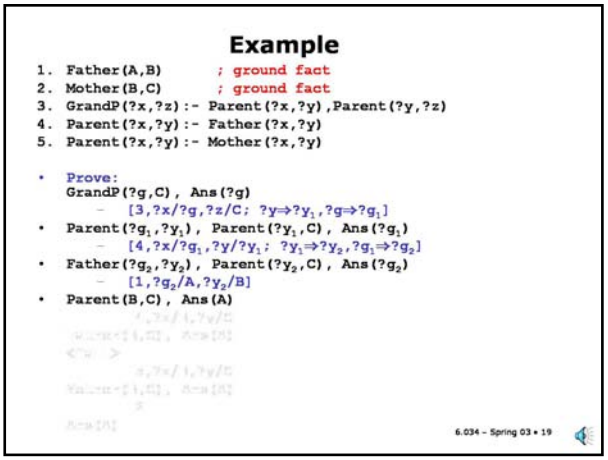


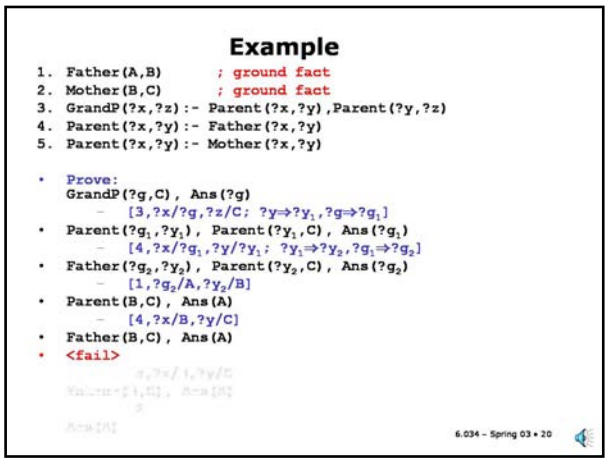


Slide 11.1.18
The first Parent goal literal unifies with the consequent of rule 4 with the unifier shown. The antecedent (the `Father` literal) is pushed on the stack, the unifier is applied and the variables are renamed.

Note that there are two Parent rules, we use the first one but we have the other one available should we fail with this one.

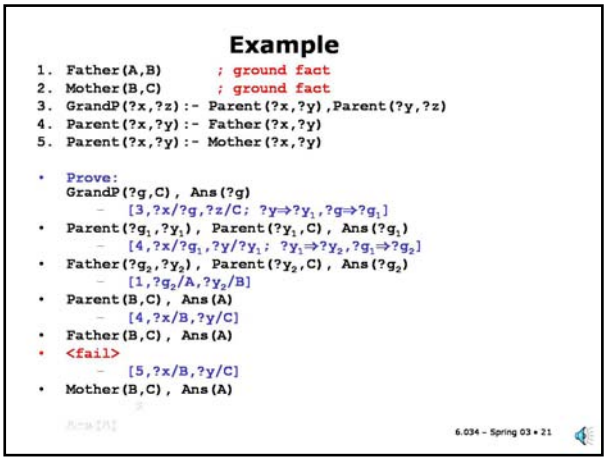
Slide 11.1.19
The `Father` goal literal matches the first fact, which now unifies the ?g variable to A and the ?y variable to B. Note that since we matched a fact, there are no antecedents to push on the stack (as in resolution with a unit-length clause). We apply the unifier, rename and proceed.

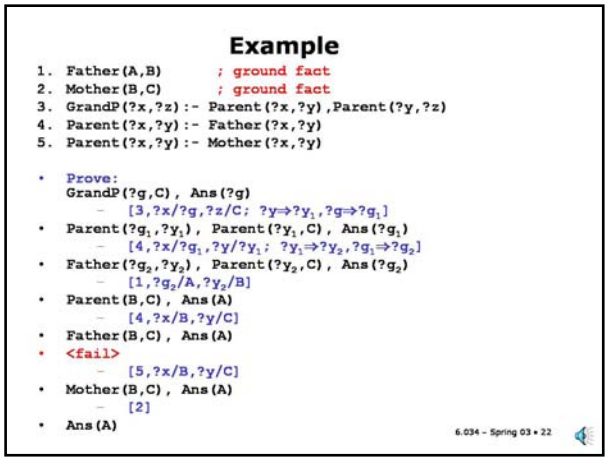




Slide 11.1.20
Now, we can match the Parent(B,C) goal literal to the consequent of rule 4 and get a new goal (after applying the substitution to the antecedent), `Father(B,C)`. However we can see that this will not match anything in the database and we get a failure.

Slide 11.1.21
The last choice we made that has a pending alternative is when we matched Parent(B,C) to the consequent of rule 4. If we instead match the consequent of rule 5, we get an alternative literal to try, namely `Mother(B,C)`.





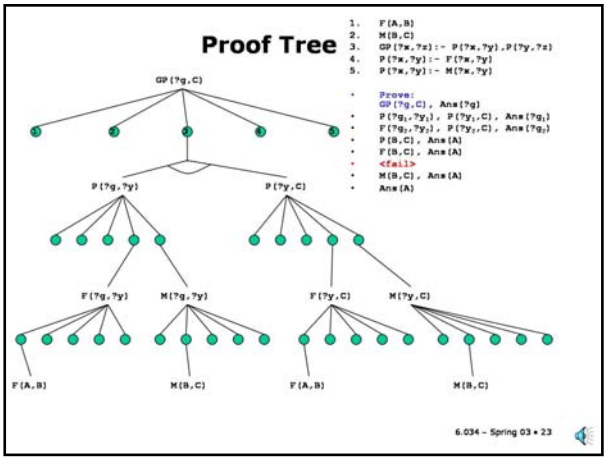
Slide 11.1.22
This matches fact 2. At this point there are no antecedents to add to the stack and the Ans literal is on the top of the stack. Note that the binding of the variable ?g to A is in fact the correct answer to our original question.

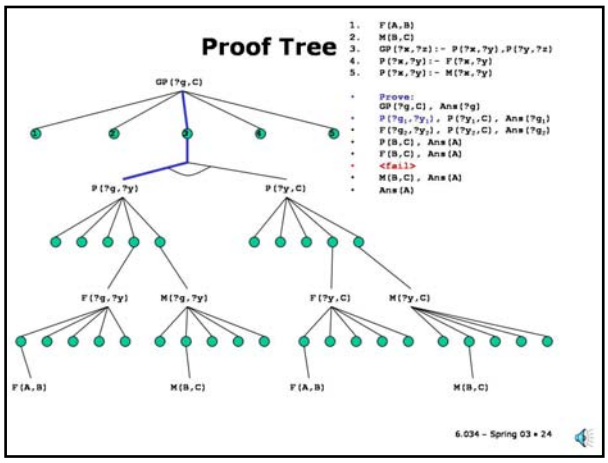
Slide 11.1.23
Another way to look at the process we have just gone through is as a form of tree search. In this search space, the states are the entries in the stack, that is, the literals that appear on our stack. The edges (shown with a green dot in the middle of each edge) are the rules or facts. However, there is one complication: a rule with multiple antecedents generates multiple children, each of which must be solved. This is indicated by the arc connecting the two descendants of rule 3 near the top of the tree.

This type of tree is called an AND-OR tree. The OR nodes come from the choice of a rule or fact to match to a goal. The AND nodes come from the multiple antecedents of a rule (all of which must be proved).

You should remember that such a tree is **implicit** in the rules and facts in our database, once we have been given a goal to prove. The tree is not constructed explicitly; it is just a way of visualizing the search process.

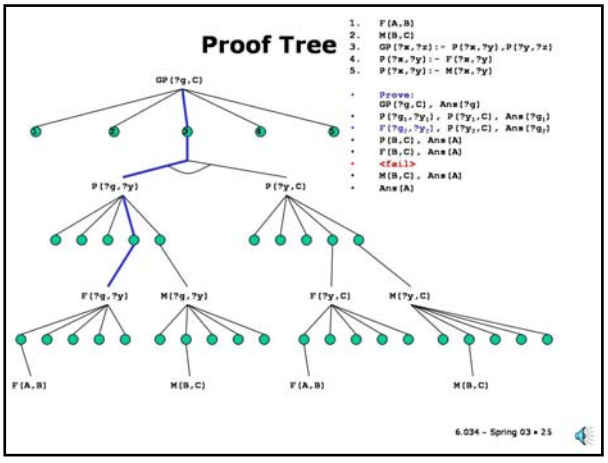
Let's go through our previous proof in this representation, which makes the choices we've made more explicit. We start with the GrandP goal at the top of the tree.

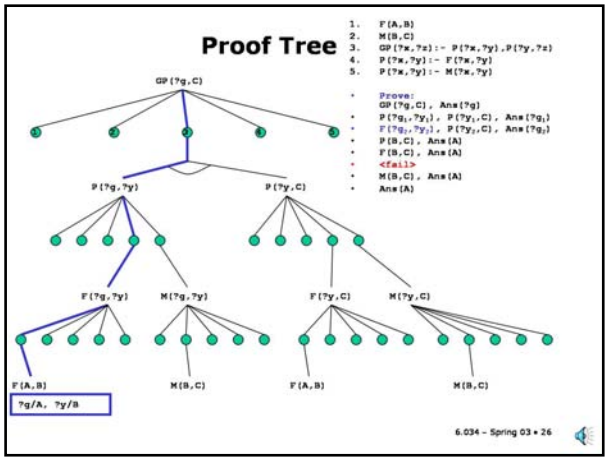




Slide 11.1.24
We match that goal to the consequent of rule 3 and we create two subgoals for each of the antecedents (after carrying out the substitutions from the unification). We will look at the first one (the one on the left) next.

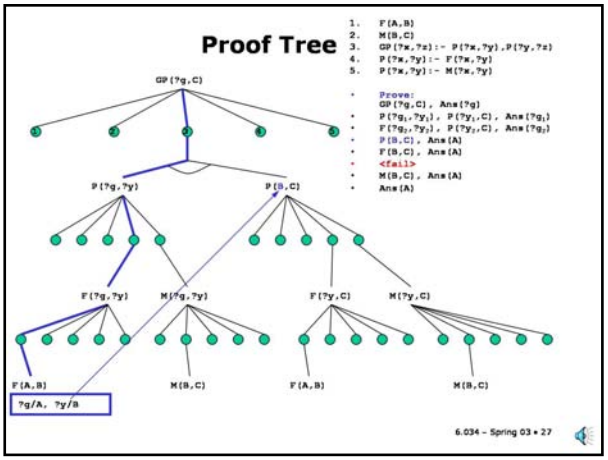
Slide 11.1.25
We match the Parent subgoal to the rule 4 and generate a `Father` subgoal.

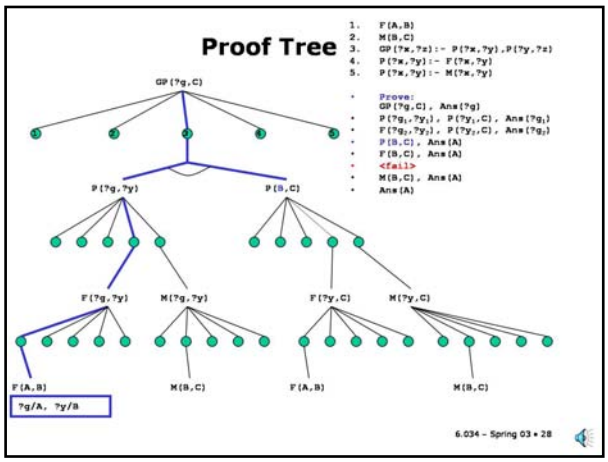




Slide 11.1.26
Which we match to fact 1 and create bindings for the variables in the goal. In all our previous steps we also created variable bindings but they were variable to variable bindings. Here, we finally match some variables to constants.

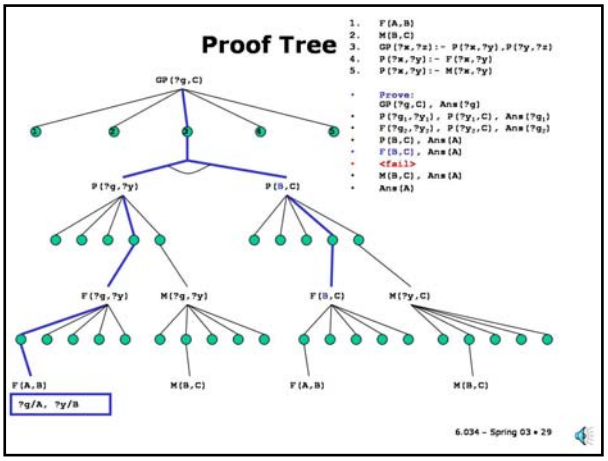
Slide 11.1.27
We have to apply this unifier to all the pending goals, including the pending Parent subgoal from rule 3. This is the part that's easy to forget when using this tree representation.

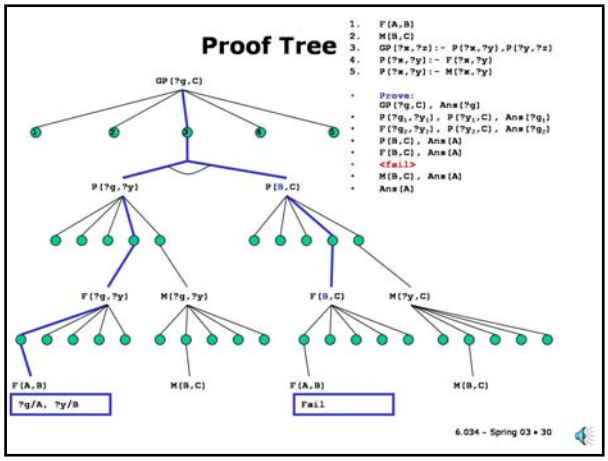




Slide 11.1.28
Now, we tackle the second Parent subgoal ...

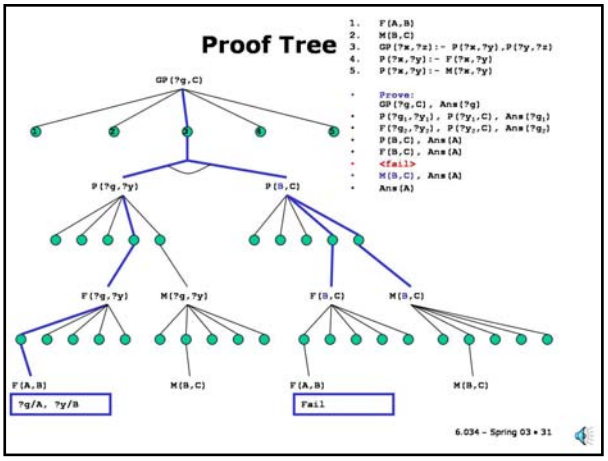
Slide 11.1.29
... which proceeds as before to match rule 4 and generate a `Father` subgoal, `Father(B,C)` in this case.

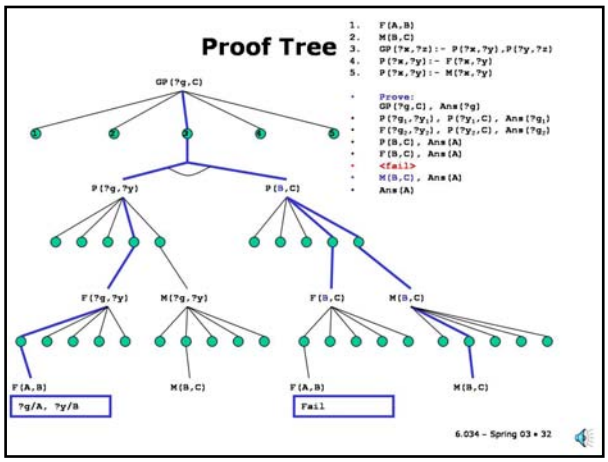




Slide 11.1.30
But, as we saw before that leads to a failure when we try to match the database.

Slide 11.1.31
So, instead, we look at the other alternative, matching the second Parent subgoal to rule 5, and generate a `Mother(B,C)` subgoal.



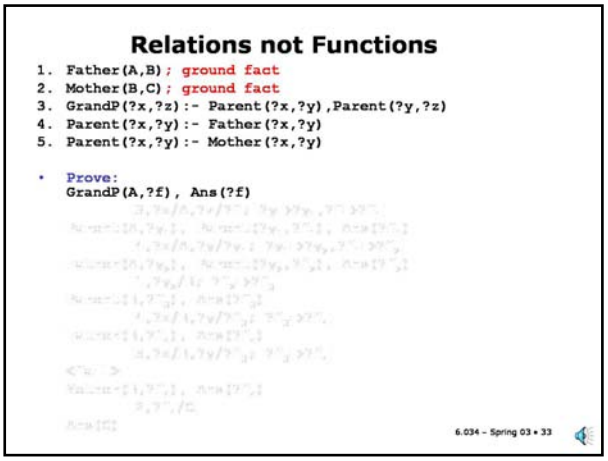


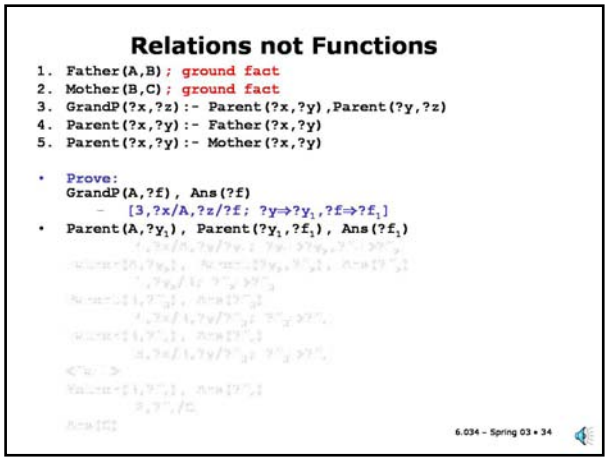
Slide 11.1.32
This matches the second fact in the database and we succeed with our proof since we have no pending subgoals to prove.

This view of the proof process highlights the search connection and is a useful mental model, although it is too awkward for any big problem.

Slide 11.1.33
At the beginning of this section, we indicated as one of the advantages of a logical representation that we could define the relationship between parents and grandparents without having to give an algorithm that might be specific to finding grandparents of grandchildren or vice versa. This is still (partly) true for logic programming. We have just seen how we could use the facts and rules shown here to find a grandparent of someone. Can we go the other way? The answer is yes.

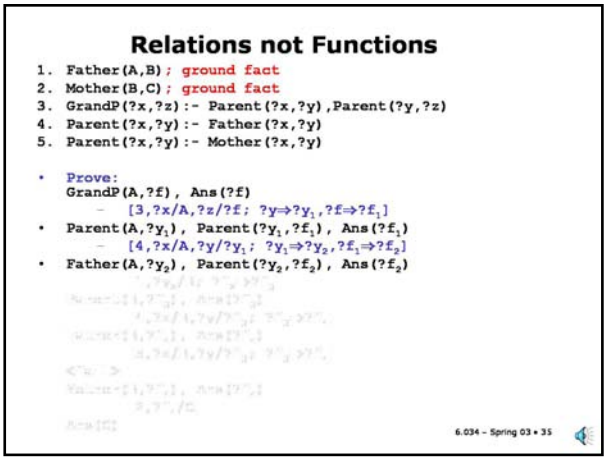
The initial goal we have shown here asks for the grandchild of A, which we know is C. Let's see how we find this answer.

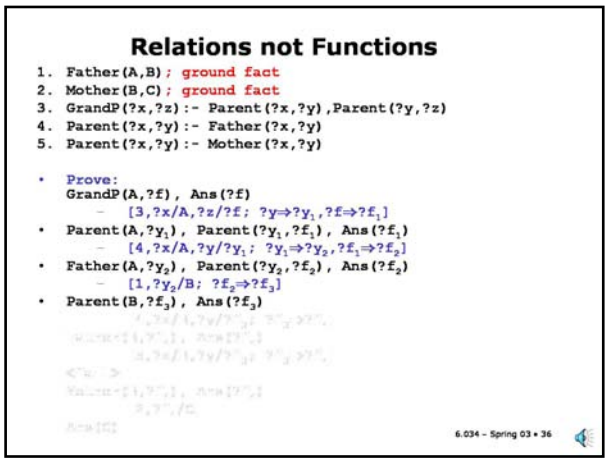




Slide 11.1.34
Once again, we match the GrandP goal to rule 3, but now the variable bindings are different. We have a constant binding in the first Parent subgoal rather than in the second.

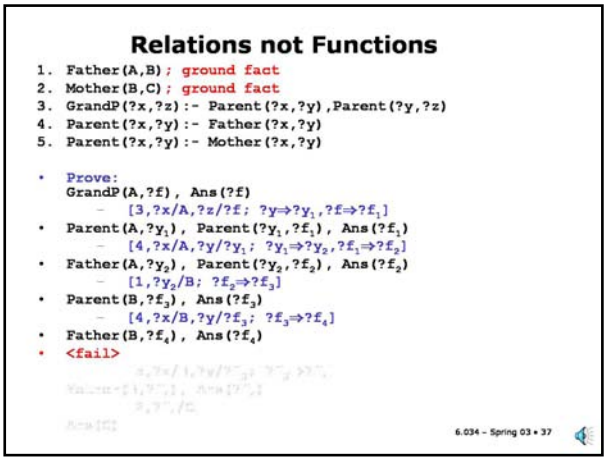
Slide 11.1.35
Once again, we match the Parent subgoal to rule 4 and get a new `Father` subgoal, this time involving A. We are basically looking for a child of A.

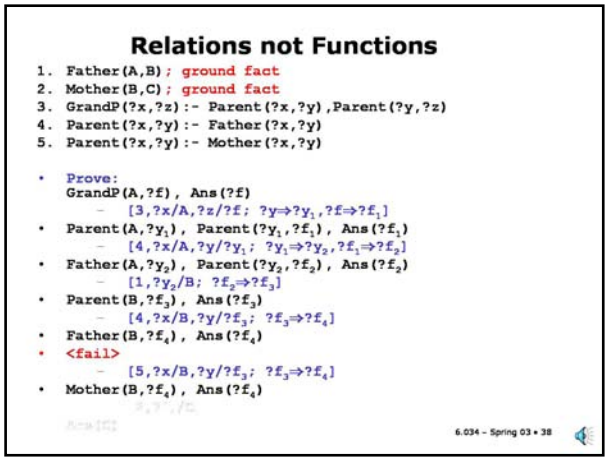




Slide 11.1.36
Then, we match the first fact, namely `Father(A,B)`, which causes us to bind the ?x variable in the second Parent subgoal to B. So, now, we look for a child of B.

Slide 11.1.37
We match the Parent subgoal to rule 4 and generate another `Father` subgoal, which fails. So, we backup to find an alternative.





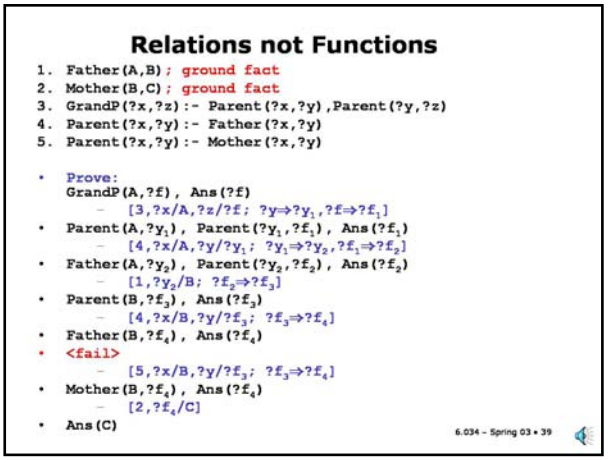
Slide 11.1.38
We now match the second Parent subgoal to rule 5 and generate a `Mother(B,?f)` subgoal.

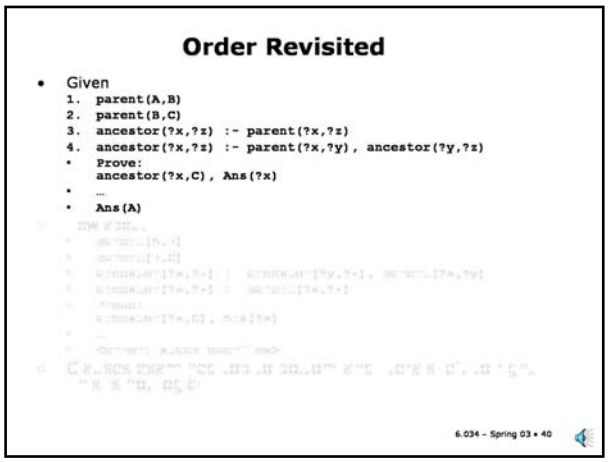
Slide 11.1.39
...which succeeds and binds ?f (our query variable) to C, as expected.

Note that if we had multiple grandchildren of A in the database, we could generate them all by continuing the search at any pending subgoals that had multiple potential matches.

The bottom line is that we are representing **relations** among the elements of our domain (recall that's what a logical predicate denotes) rather than computing functions that specify a single output for a given set of inputs.

Another way of looking at it is that we do not have a pre-conceived notion of which variables represent "input variables" and which are "output variables".

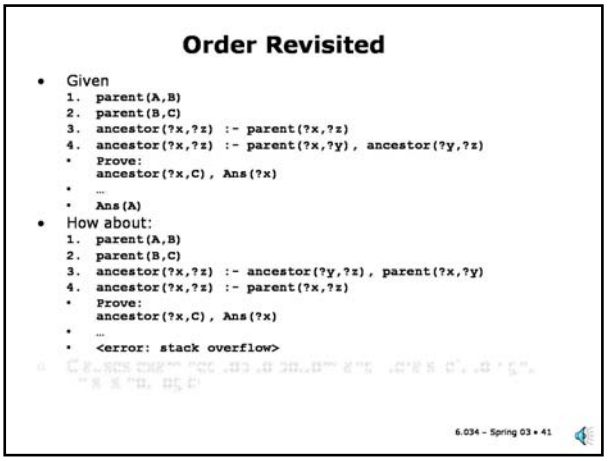


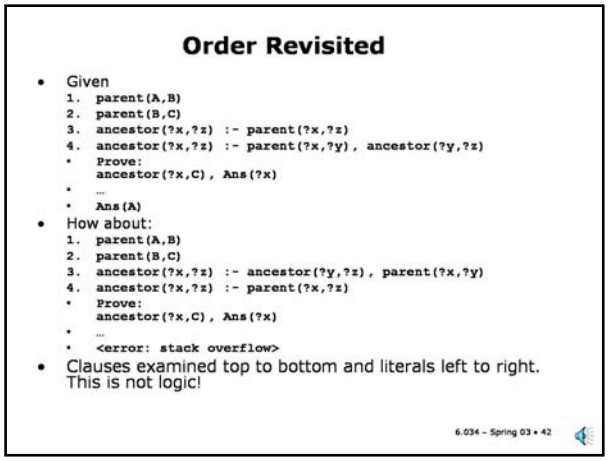


Slide 11.1.40
We have seen in our examples thus far that we explore the underlying search space in order. This approach has consequences. For example, consider the following simple rules for defining an ancestor relation. It says that a parent is an ancestor (this is the base case) and that the ancestor of a parent is an ancestor (the recursive case). You could use this definition to list a person's ancestors or, as we did for grandparent, to list a person's descendants.

But what would happen if we changed the order a little bit?

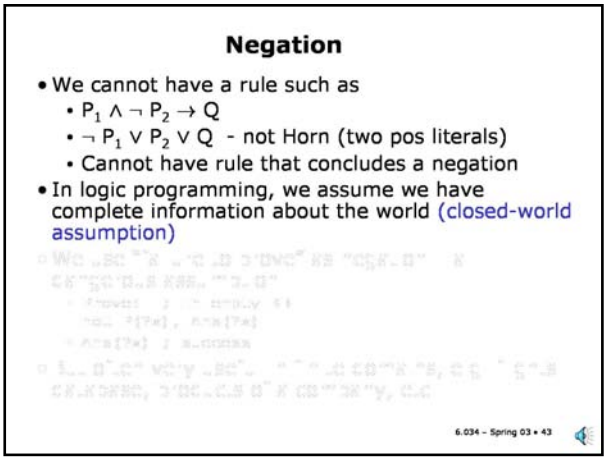
Slide 11.1.41
Here we've switched the order of rules 3 and 4 and furthermore switched the order of the literals in the recursive ancestor rule. The effect of these changes, which have no logical import, is disastrous: basically it generates an infinite loop.

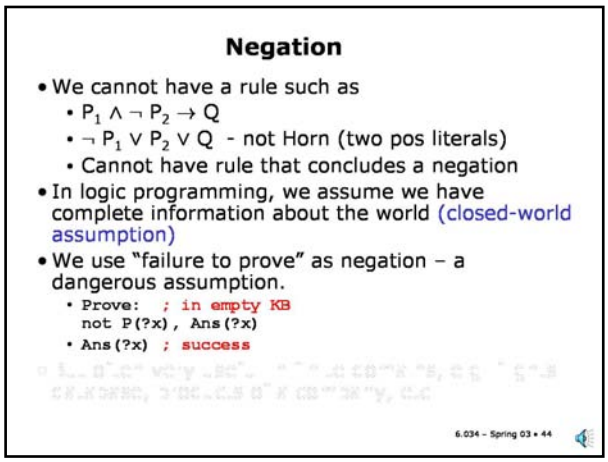




Slide 11.1.42
This type of behavior is what you would expect from a recursive program if you put the recursive case before the base case. The key point is that logic programming is half way between traditional programming and logic and exactly like neither one.

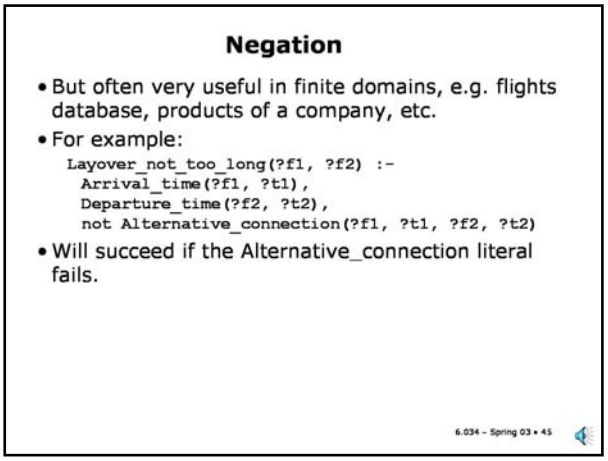
Slide 11.1.43
It is often the case that we want to have a condition on a rule that says that something is not true. However, that has two problems, one is that the resulting rule would not be Horn. Furthermore, as we saw earlier, we have no way of concluding a negative literal. In logic programming one typically makes a **closed world** assumption, sometimes jokingly referred to as the "closed mind" assumption, which says that we know everything to be known about our domain. And, if we don't know it (or can't prove it), then it must be false. We all know people like this...





Slide 11.1.44
Given we assume we know everything relevant, we can simulate negation by failure to prove. This is very dangerous in general situations where you may not know everything (for example, it's not a good thing to assume in exams)...

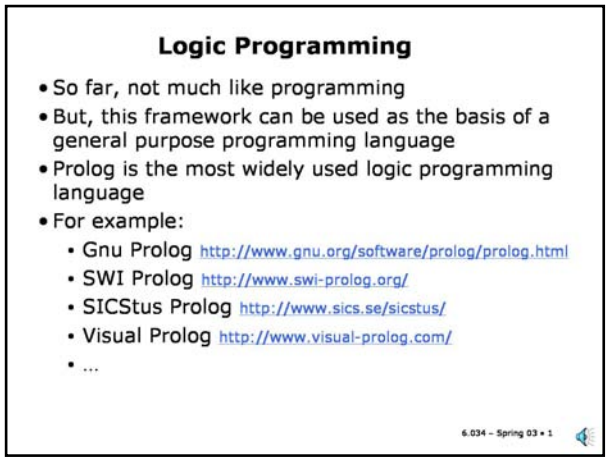
Slide 11.1.45
... but very useful in practice. For example, we can write rules of the form "if there are no other acceptable flights, accept a long layover" and we establish this by looking over all the known flights.

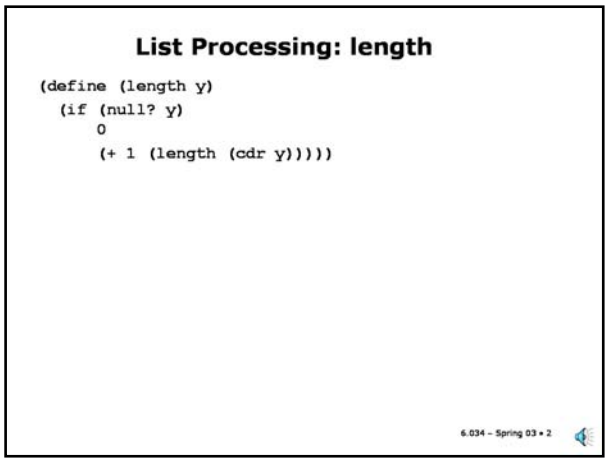


6.034 Notes: Section 11.2

Slide 11.2.1
So far, what we have seen of logic programming may not seem much like programming. Now, we will look at a number of list processing examples that will look more like the examples that you are used to writing in Scheme.

What we will see is essentially a subset of the logic programming language Prolog, which is used fairly widely. There are a number of open-source and commercial versions of Prolog available. We will use a very simple home-brew system implemented in Scheme rather that one of these systems so that there are no mysteries in the implementation. However, we will pay a substantial performance penalty for this choice.





Slide 11.2.2
Let's start with a very simple Scheme program to compute the length of a list. It's composed of two "cases", the base case when the list is null and the recursive case, in which we reduce the problem into a simpler instance of the same problem (getting the length of the cdr of the list) and compute the final result by adding one to the result of the recursive call.

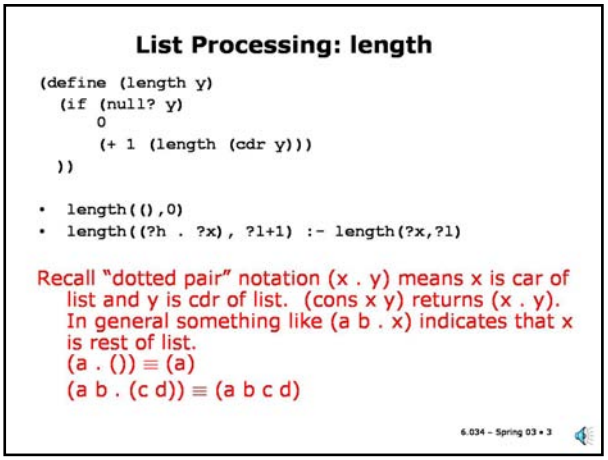
Slide 11.2.3
This would be a Prolog-like solution to the same problem. It has essentially the same structure as the Scheme program. We use a predicate "length" that has two arguments, one is the list and the other its length.

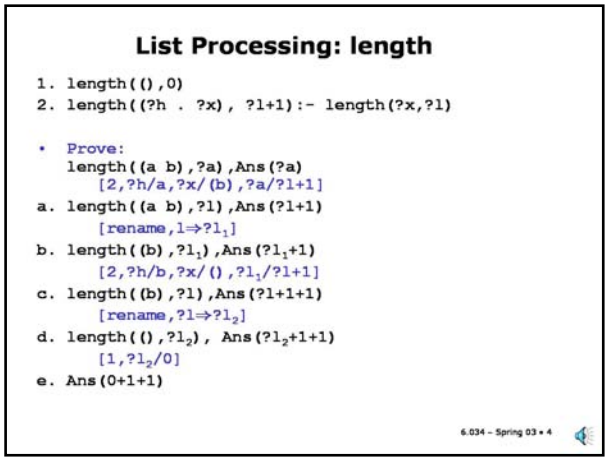
The first "fact" handles the base case; it defines the length of the null list as 0.

The second rule handles the recursive case. The consequent of the rule (the left-hand-side) is what will match a pending subgoal. Note the form of the first argument of the consequent: it is a Scheme dotted pair. It is set up to match the variable ?h to the car of a list and the variable ?x to the cdr of the list. The second argument of the consequent expresses the length of the list as a function of the length of the cdr of the list.

The right hand side of the rule is the IF part. It sets up a simpler subgoal to solve. Once we solve it, we will have bound ?l to the length of the cdr and we will know the length of the full list (including the car).

Let's look at an example.



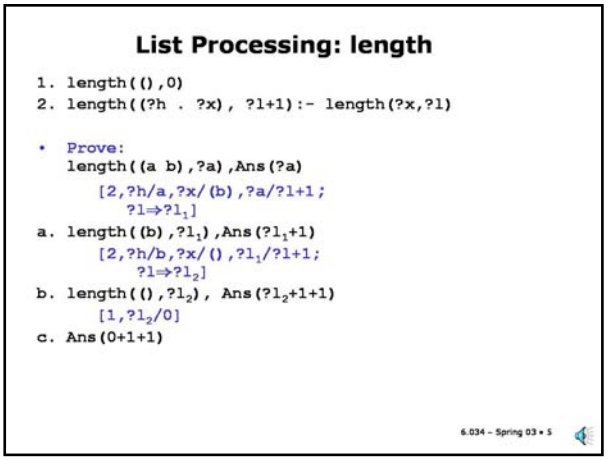


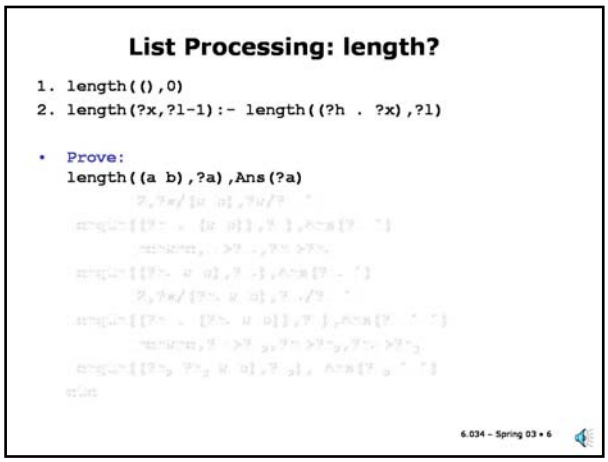
Slide 11.2.4
You can see the operation of this little program here. The operation is very like that of the corresponding Scheme program. The sequence of subgoals corresponds to the recursive calls to the program.

We have separated the unifier substitution step from the renaming to make things a little clearer. Note that without the renaming we would be hopelessly confused with the bindings of ?l.

In practice, in a Prolog system, the arithmetic expressions would be evaluated by the system and we would get Ans(2).

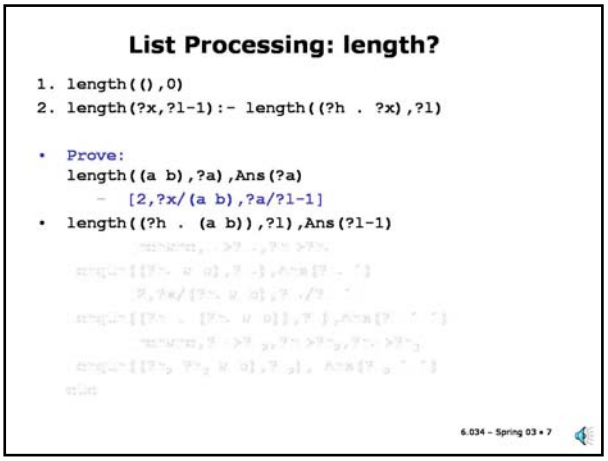
Slide 11.2.5
This is the same operation but combining the unfier substitution and renaming steps. You can see the sequence of subgoals more clearly here.

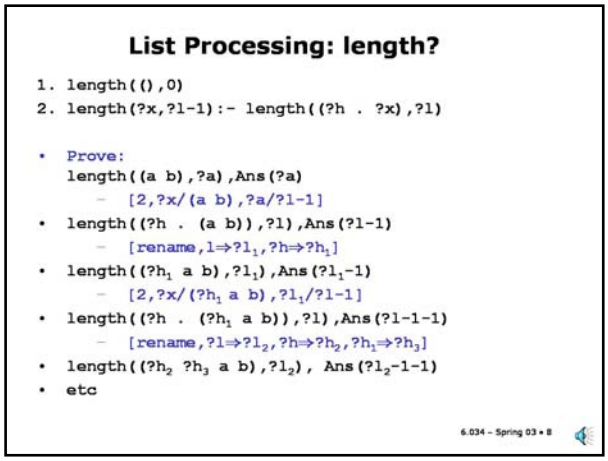




Slide 11.2.6
You may be wondering whether this formulation of length would also work. Certainly, it seems just as valid as the one we used. Let's trace it through. We start with the same goal as before, finding the length of the list (a b).

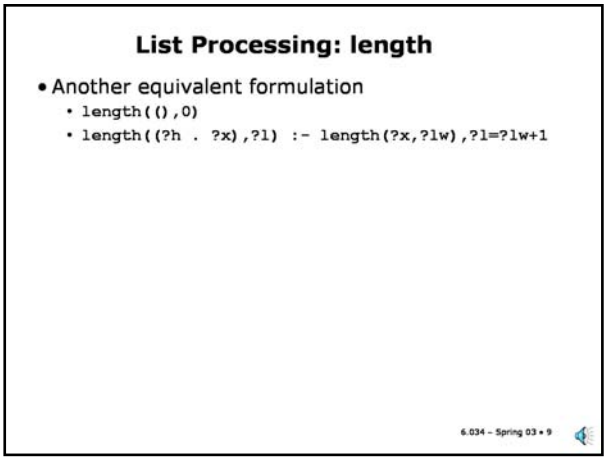
Slide 11.2.7
We match the goal to the consequent of rule 2 and do the substitution to get a new subgoal. Note, however, that this is not a simpler subgoal. It's actually trying to find the length of a longer, not completely specified, list. If we knew the length of such a list then we could know the length of our input list. Can you smell trouble brewing?

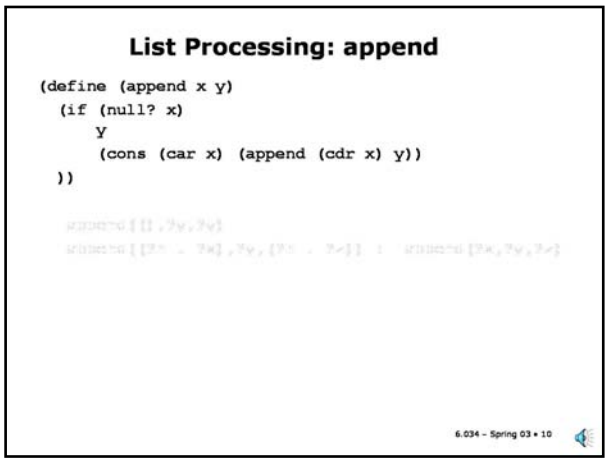




Slide 11.2.8
Sure enough; we've coded an infinite loop. The moral of the story is that you want to write these recursive rules in the form of "complex consequent :- simple antecedent" and not the other way around.

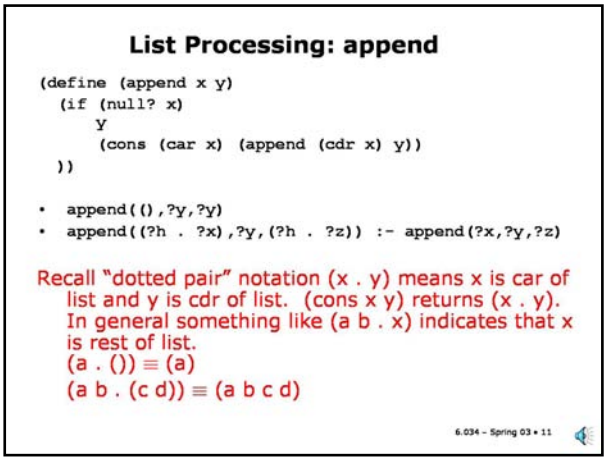
Slide 11.2.9
Here's another formulation of length that does work. Here we've separated the updating of the length into a separate equality statement. The software that we will be using will require this particular form.





Slide 11.2.10
Let's look at another example, which is quite parallel to length. Here is a Scheme implementation of an append function. It too consists of two cases. The base case handles the case of the first argument being null, in which case the answer is simply the second list. The recursive case involves computing the solution to a simpler case (append of the cdr of x to y) and updating it to the final answer by consing the car of x to the result.

Slide 11.2.11
The logic program is completely analogous. The append predicate has three arguments, the lists to be appended and the result list. The first fact just says that the output of appending the null list and any list is just the second list. The second rule looks more complicated but it is just like the Scheme program. We pick out the car and the cdr in the consequent (note the use of dotted pair notation) and bind them to ?h and ?x respectively. Then we define a subgoal involving ?x and ?y and bind the result to ?z. We can then construct the result for the original list by consing ?h to ?z (using dotted pair notation).



List Processing: append

```
1. append((),?y,?y)
2. append((?h . ?x),?y,(?h . ?z)):- append(?x,?y,?z)
```

- Prove:
  append((a b),(c d),?l),Ans(?l)

  [2,?h/a,?x/(b),?y/(c d),?l/(a . ?z); $?z \Rightarrow ?z_1$]

a. append((b),(c d),$?z_1$),Ans((a . $?z_1$))

  [2,?h/b,?x/(),?y/(c d),$?z_1$/(b . ?z); $?z \Rightarrow ?z_2$]

b. append((),(c d),$?z_2$), Ans((a . (b . $?z_2$)))

  [1,?y/$?z_2$,$?z_2$/(c d)]

c. Ans((a . (b . (c d))))

  – Note that (a . (b . (c d))) is (a b c d)

6.034 – Spring 03 • 12

Slide 11.2.12
Here you can trace the operation of these rules in a very simple example. Once again note that the renaming is crucial for keeping things straight.

Slide 11.2.13
Thus far, the structure of the logic programs we have seen is very similar to that of the natural Scheme programs. But that's not always the case. Let's look at an alternative way of representing lists that leads to very different looking programs. The representation is called **difference lists**. You can see some examples of this representation of a simple list with three elements here. The most important one is diff((a b c . ?x), ?x), which says that any list starting with (a b c) and followed by anything can be used to represent the list.

Difference Lists

- A list can be represented as the difference between two lists, which we will write `diff(L1,L2)`
- For example, (a b c) can be written as
  - `diff( (a b c), () )`
  - `diff( (a b c d), (d) )`
  - `diff( (a b c d e), (d e) )`
  - `diff( (a b c . ?x), ?x )`
- The empty list is any list of the form
  - `diff(?x, ?x)`

6.034 – Spring 03 • 13

Difference Lists

- A list can be represented as the difference between two lists, which we will write `diff(L1,L2)`
- For example, (a b c) can be written as
  - `diff( (a b c), () )`
  - `diff( (a b c d), (d) )`
  - `diff( (a b c d e), (d e) )`
  - `diff( (a b c . ?x), ?x )`
- The empty list is any list of the form
  - `diff(?x, ?x)`
- In `diff(L1,L2)` think of `L1` as a pointer to the beginning of the list and `L2` as a pointer to the end of the list.

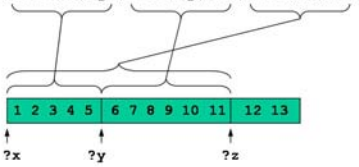
6.034 – Spring 03 • 14

Slide 11.2.14
The basic idea is that we can represent a list by a pair of pointers into a bigger list, one to the beginning and the other to the end of the list.

Slide 11.2.15
In this representation we can code append as a single fact! The picture shows the intuition behind the definition. On first viewing this seems like we're cheating. It's easy to see that this statement is true, but how does it actually compute anything? Partly, one has to think carefully about the representation of the input.

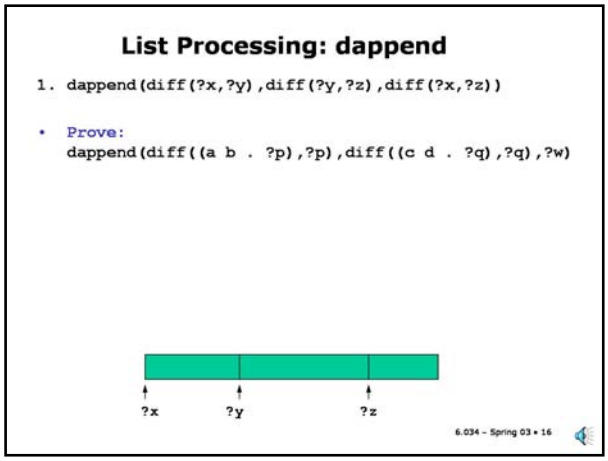
List Processing: dappend

```
1. dappend(diff(?x,?y),diff(?y,?z),diff(?x,?z))
```




```
?x = (1 2 3 4 5 6 7 8 9 10 11 12 13)
?y = (6 7 8 9 10 11 12 13)
?z = (12 13)
```

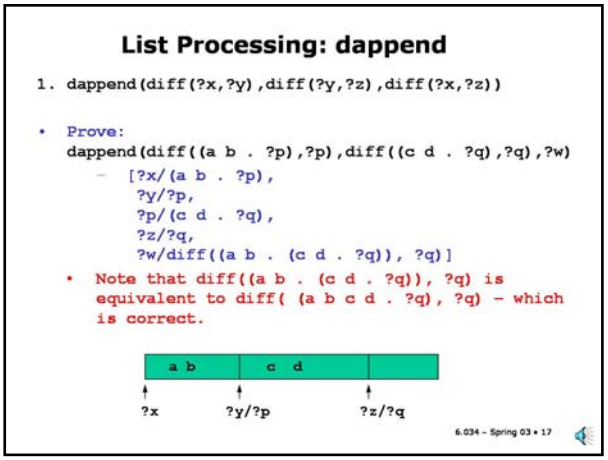
6.034 – Spring 03 • 15

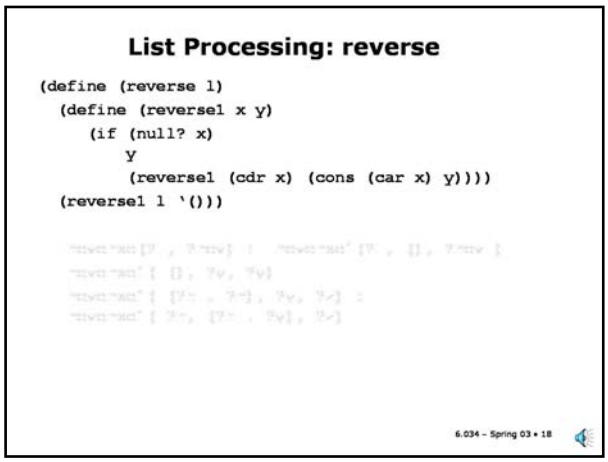


Slide 11.2.16
Here we see how a goal would be phrased in this representation. We have used the most general representation of the input lists.

Slide 11.2.17
Now, we unify. Note that unification ends up equating ?y, the end of the first list in the rule, with ?p, the end of the first input list. Then ?p (and therefore ?y) is matched to the start of the second list. This is basically what carries out the append operation.

Yes, it looks like magic. We will be using difference lists a fair bit when we do natural language processing, so it is worth spending a bit of time understanding them.



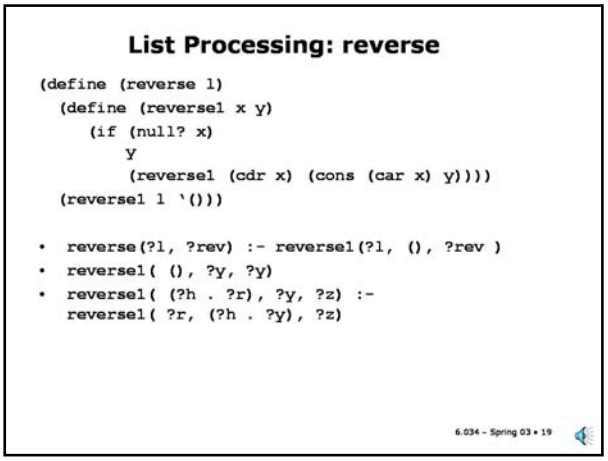


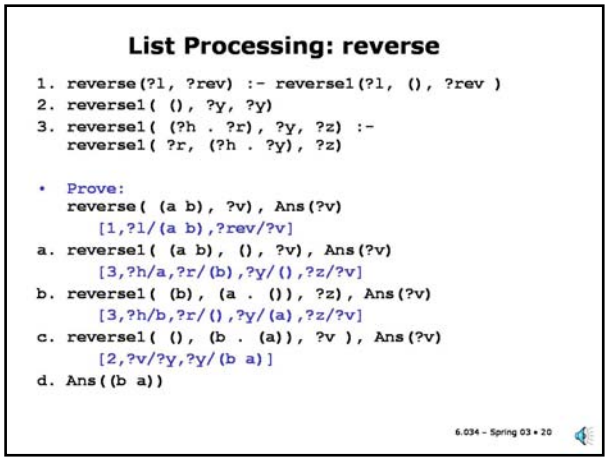
Slide 11.2.18
Let's look at another example, first without difference lists and then with.

This is Scheme for a list reverse operation. It's a bit more complicated than the cases we've seen so far. To reverse the list, we need a temporary value to serve as an accumulator for the reversed list. That's what the y argument to the inner procedure is. y starts with the null list and we cons each of the elements of the input onto this list. When the first argument is null, we return the accumulated list.

Slide 11.2.19
We follow the same pattern in the logic program. We define the predicate reverse, with two arguments, the input and output lists, in terms of a three-place auxiliary predicate reverse1, which introduces the accumulator and initializes it to nil. Note that if you reverse the first argument of reverse1 and append it to the second argument of reverse1 then that gives the answer to the original query.

Reverse1 is defined by two rules: in the base case when the first argument is nil, we simply equate the output list to the accumulator. In the general case, we set up a recursive subgoal with the cdr of the list, but we cons the car of the input list to the accumulator.





Slide 11.2.20
You can see the operation on a simple example here.

Slide 11.2.21
Here is the implementation using a difference list instead of an explicit accumulator. Note that the end of the difference list essentially behaves as the accumulator variable did in the previous implementation.

Hopefully, this has given you some flavor for logic programming. We will see more examples of these types of rules in the next chapter.