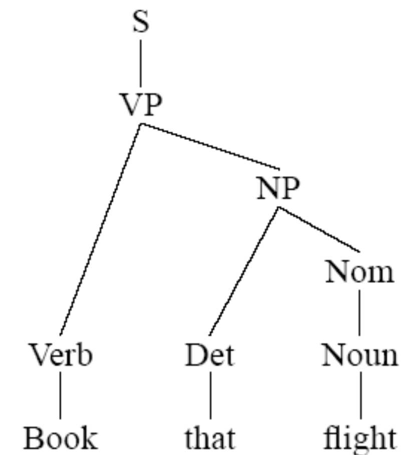
# Unification-based approach to NLP

final parsing and semantics examples; Unification-based approach to NLP; bits of history, First-order predicate logic; unification; Resolution

# CFG Parsing example

| | |
|---|---|
| $S \rightarrow NP\ VP$ | $Det \rightarrow that \mid this \mid a$ |
| $S \rightarrow Aux\ NP\ VP$ | $Noun \rightarrow book \mid flight \mid meal \mid money$ |
| $S \rightarrow VP$ | $Verb \rightarrow book \mid include \mid prefer$ |
| $NP \rightarrow Det\ Nominal$ | $Aux \rightarrow does$ |
| $Nominal \rightarrow Noun$ | |
| $Nominal \rightarrow Noun\ Nominal$ | $Prep \rightarrow from \mid to \mid on$ |
| $NP \rightarrow Proper\text{-}Noun$ | $Proper\text{-}Noun \rightarrow Houston \mid TWA$ |
| $VP \rightarrow Verb$ | |
| $VP \rightarrow Verb\ NP$ | $Nominal \rightarrow Nominal\ PP$ |

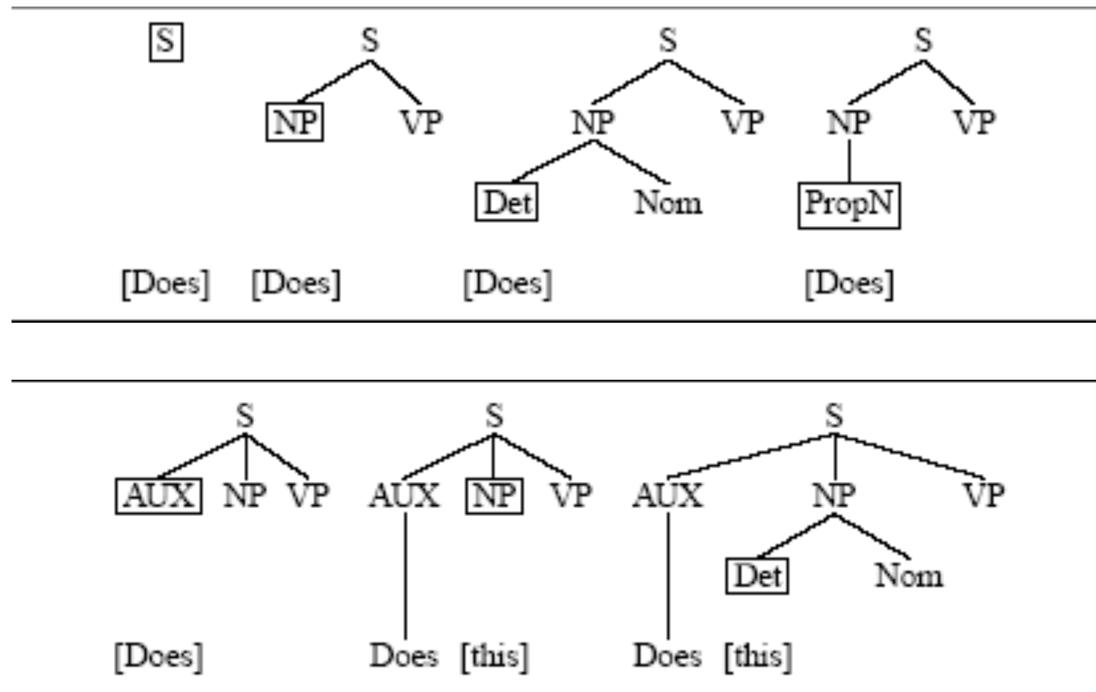


# A top-down, depth-first, left to right parser

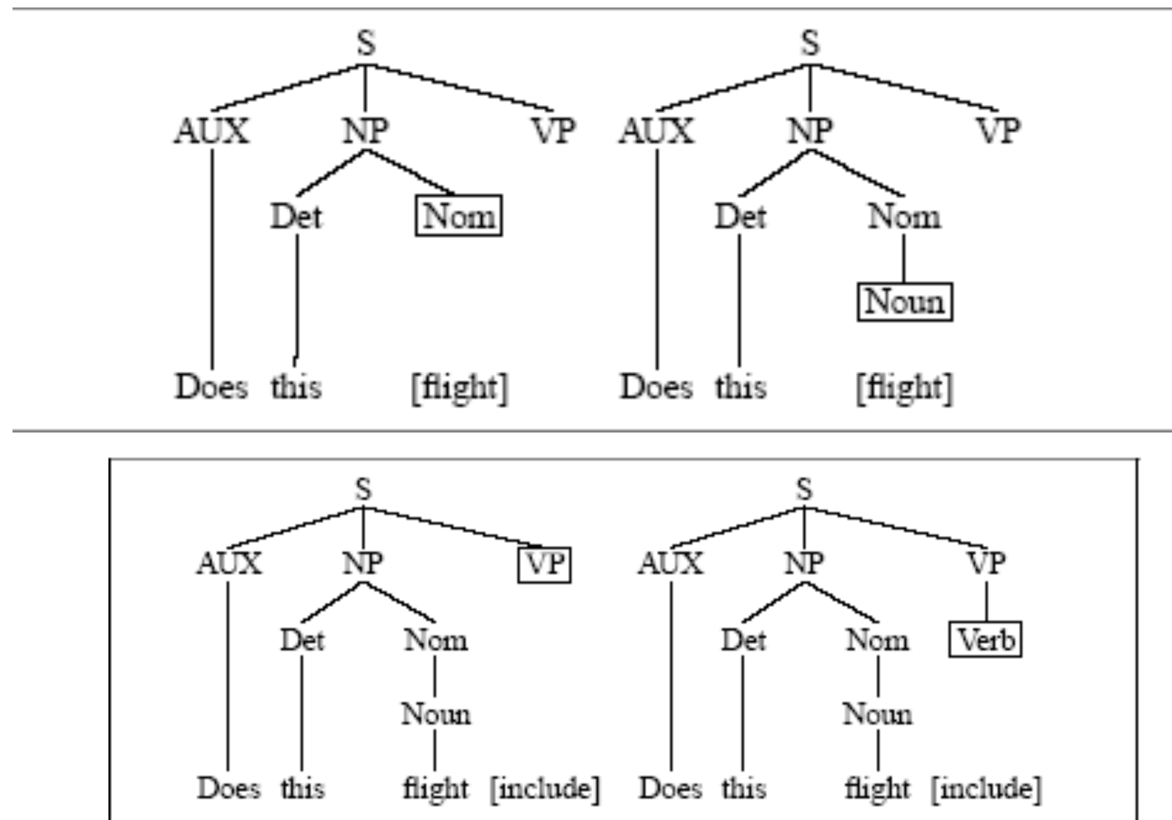
```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

 agenda ← (Initial S tree, Beginning of input)
 current-search-state ← POP(agenda)
 loop
   if SUCCESSFUL-PARSE?(current-search-state) then
     return TREE(current-search-state)
   else
     if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
       if CAT(node-to-expand)
            ⊂
          POS(CURRENT-INPUT(current-search-state)) then
         PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
       else
         return reject
     else
       PUSH(APPLY-RULES(current-search-state, grammar), agenda)
   if agenda is empty then
       return reject
   else
       current-search-state ← NEXT(agenda)
 end
```

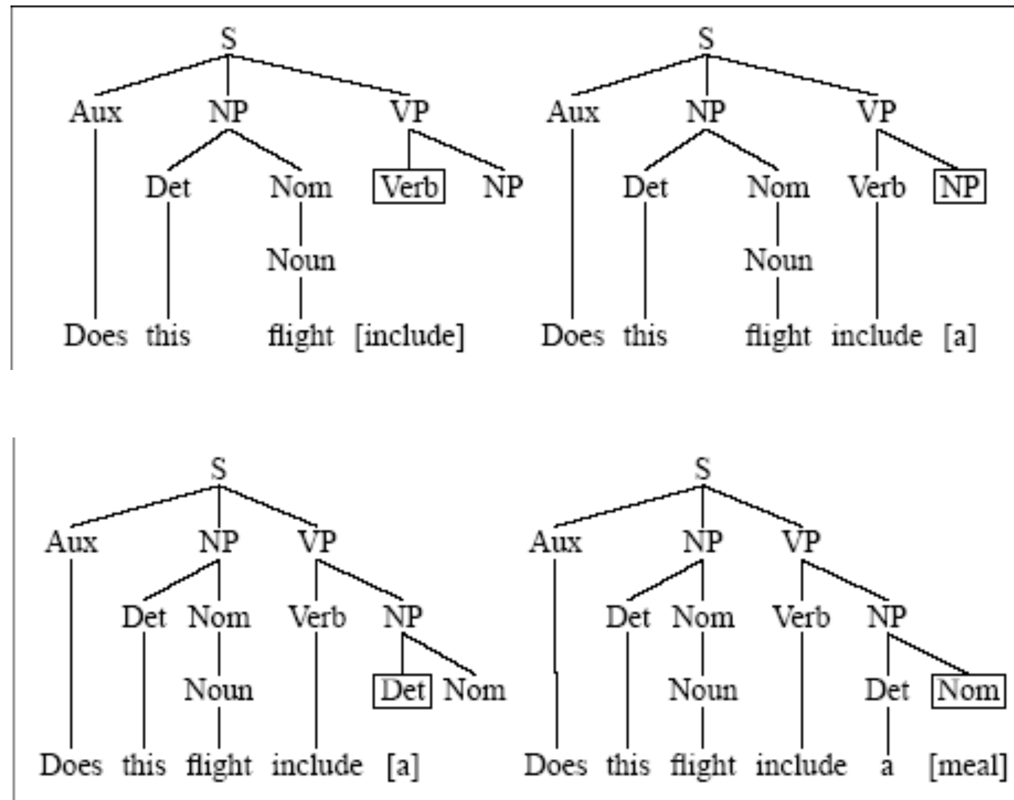
# A top-down, depth-first, left to right parser, example (1)



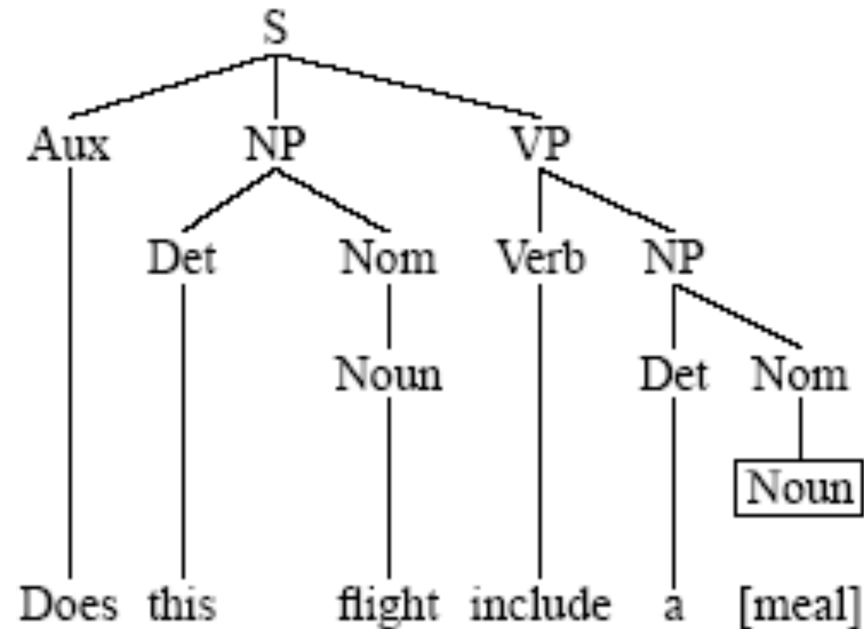
# A top-down, depth-first, left to right parser, example (2)



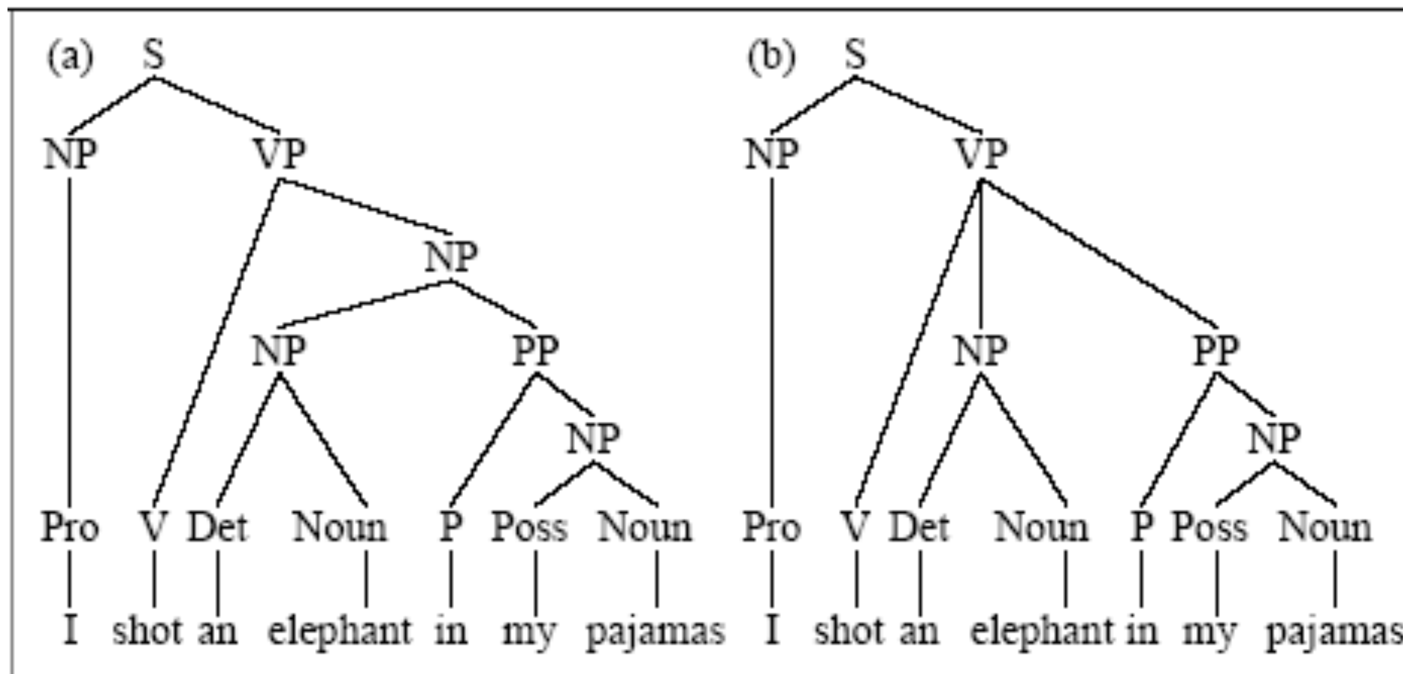
# A top-down, depth-first, left to right parser, example (3)



# A top-down, depth-first, left to right parser, example (3)



# Big Problem: PP attachment ambiguity



# Solutions

- Use a Probabilistic Parser (covered later in class)
- Use semantics

# Example again (w/Earley parser) “Book that flight”

| Chart[0] | | |
|---|---|---|
| $\gamma \rightarrow \bullet S$ | [0,0] | Dummy start state |
| $S \rightarrow \bullet NP\ VP$ | [0,0] | Predictor |
| $NP \rightarrow \bullet Det\ NOMINAL$ | [0,0] | Predictor |
| $NP \rightarrow \bullet Proper\text{-}Noun$ | [0,0] | Predictor |
| $S \rightarrow \bullet Aux\ NP\ VP$ | [0,0] | Predictor |
| $S \rightarrow \bullet VP$ | [0,0] | Predictor |
| $VP \rightarrow \bullet Verb$ | [0,0] | Predictor |
| $VP \rightarrow \bullet Verb\ NP$ | [0,0] | Predictor |

# Example again (w/Earley parser) "Book that flight" (2)

| Chart[1] | | |
|---|---|---|
| $Verb \rightarrow book \bullet$ | [0,1] | Scanner |
| $VP \rightarrow Verb \bullet$ | [0,1] | Completer |
| $S \rightarrow VP \bullet$ | [0,1] | Completer |
| $VP \rightarrow Verb \bullet NP$ | [0,1] | Completer |
| $NP \rightarrow \bullet Det\ NOMINAL$ | [1,1] | Predictor |
| $NP \rightarrow \bullet Proper\text{-}Noun$ | [1,1] | Predictor |

# Example again (w/Earley parser) “Book that flight” (3)

Chart[2]

| | | |
|---|---|---|
| $Det \rightarrow that\bullet$ | [1,2] | Scanner |
| $NP \rightarrow Det\bullet NOMINAL$ | [1,2] | Completer |
| $NOMINAL \rightarrow \bullet\ Noun$ | [2,2] | Predictor |
| $NOMINAL \rightarrow \bullet\ Noun\ NOMINAL$ | [2,2] | Predictor |

# Example again (w/Earley parser) “Book that flight” (4)

Chart[3]

| | | |
|---|---|---|
| $Noun \rightarrow flight \bullet$ | [2,3] | Scanner |
| $NOMINAL \rightarrow Noun \bullet$ | [2,3] | Completer |
| $NOMINAL \rightarrow Noun \bullet NOMINAL$ | [2,3] | Completer |
| $NP \rightarrow Det\ NOMINAL \bullet$ | [1,3] | Completer |
| $VP \rightarrow Verb\ NP \bullet$ | [0,3] | Completer |
| $S \rightarrow VP \bullet$ | [0,3] | Completer |
| $NOMINAL \rightarrow \bullet Noun$ | [3,3] | Predictor |
| $NOMINAL \rightarrow \bullet Noun\ NOMINAL$ | [3,3] | Predictor |

## Parsing with Features

3 views of a context-free rule

- generation (production): S → NP VP
- parsing (comprehension): S ← NP VP
- verification (checking): S = NP VP

- Today you should keep the third, declarative perspective in mind.

- Each phrase has
  - an interface (S) saying where it can go
  - an implementation (NP VP) saying what's in it
- To let the parts of the tree coordinate more closely with one another, enrich the interfaces:

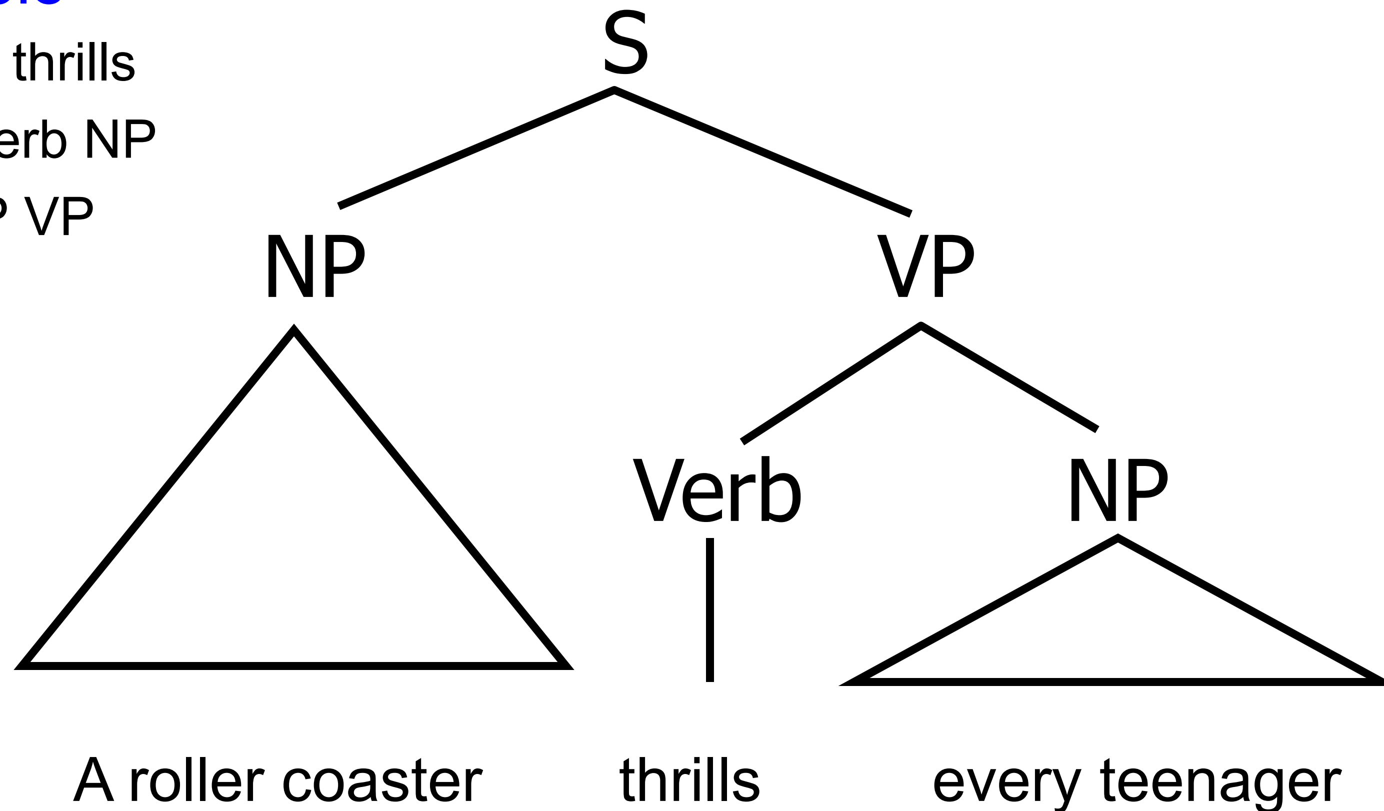
  S[features…] = NP[features…] VP[features…]

## Example

Verb → thrills

VP→ Verb NP

S → NP VP



# 3 common ways to use features

morphology of a single word:

Verb[head=thrill, tense=present, num=sing, person=3,…] → thrills

projection of features up to a bigger phrase

VP[head=$\alpha$, tense=$\beta$, num=$\gamma$…] → V[head=$\alpha$, tense=$\beta$, num=$\gamma$…] NP

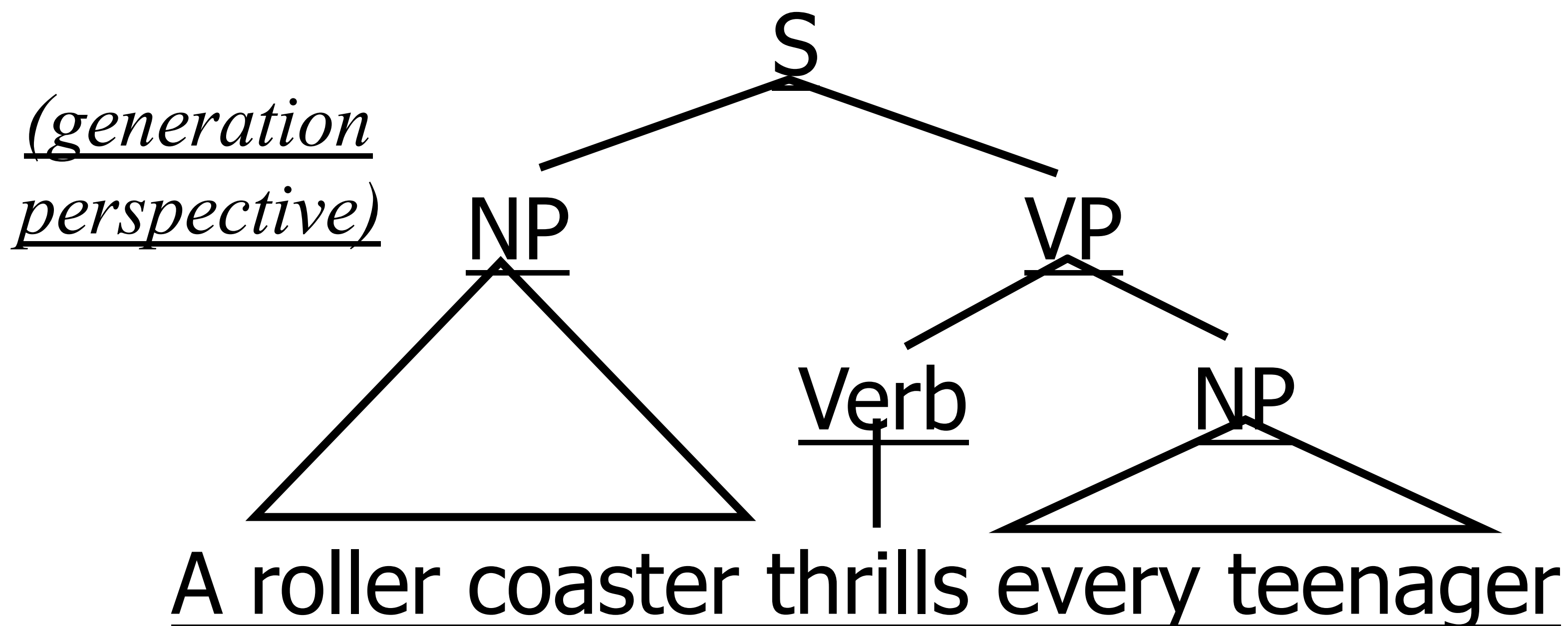
provided $\alpha$ is in the set TRANSITIVE-VERBS

agreement between sister phrases:

S[head=$\alpha$, tense=$\beta$] → NP[num=$\gamma$,…] VP[head=$\alpha$, tense=$\beta$, num=$\gamma$…]

## 3 Common Ways to Use Features

Verb[head=thrill, tense=present, num=sing, person=3,…] → thrills
VP[head=α, tense=β, num=γ…] → V[head=α, tense=β, num=γ…] NP
S[head=α, tense=β] → NP[num=γ,…] VP[head=α, tense=β, num=γ…]



# Uses of Grammar

- Prescriptive - Identify speaker's socioeconomic class & education level; Identify level of formality of a particular usage
- Descriptive - Understand how people produce & understand language; Identify similarities & differences across languages; Development of language technologies

# Competence vs. Performance

The Distinction

- Competence - knowledge of language
- Performance - how the knowledge is used

## Acceptability vs. grammaticality

- A sentence is acceptable if native speakers say it sounds good.
- A sentence is grammatical (with respect to a particular grammar) if the grammar licenses it.
- Linguists are sometimes sloppy about the difference.

## The Generative Revolution

- Noam Chomsky's work in the 1950s radically changed linguistics, making syntax central.
- Chomsky has been the dominant figure in linguistics ever since

# Main Tenets of Generative Grammar

- Grammars should be formulated precisely and explicitly
- Languages are infinite, so grammars must be tested against invented data, not just attested examples.
- The theory of grammar is a theory of human linguistic abilities.

## Some of Chomsky's Controversial Claims

- The superficial diversity of human languages masks their underlying similarity.
- All languages are fundamentally alike because linguistic knowledge is largely innate.
- The central problem for linguistics is explaining how children can learn language so quickly and easily.

# Relationship of Some Syntactic Theories

Other Theories ——— Early Transformational Grammar (1955-1964)

|

Standard Theory (1964-1967)

| GB | GPSG | Realistic TG | Generative Semantics |
|---|---|---|---|
| (1981-1993) | (1979-1985) | (1978-1980) | (1966-1975) |
| \| | \| | \| | \| |
| Other | HPSG (1986-present) | LFG (1980-present) | Other |

# Logic Refresher

## Propositional Calculus

Extensively developed by Whitehead and Russell in their early 20th century classic Principia Mathematica, this system is also known as propositional logic, sentential calculus, and (informally) as symbolic logic.

The basic entities, or primitives, in the propositional calculus are propositions (sentences) which are symbolized p, q, r, s, ... A proposition symbol stands for an assertion (the sky is blue, it is raining, x=y) which may be true (T) or false (F). Propositions may be combined into more complex assertions by the use of operators, analogous to the familiar arithmetic operators of addition, multiplication, and so on. These logical connectives, however, combine propositions into logical expressions whose truth or falsity is a function of the truth value (T or F) of each component proposition.

# Logic Refresher

## Propositional Calculus (cont.)

In general, logical connectives map combinations of n propositions onto the set {T, F}. When n=1, the only mapping of interest reverses the truth variable of a proposition. We symbolize this negation operator with a minus sign and read -p (or ~p) as not p.

When n=2 (which is as high as we need to go) there are 16 possible binary logical connectives (comprising all the distinct ways truth values can be assigned to the four possible pairs of proposition values). The table below shows the mappings for the conjunction (p&q, p and q), disjunction (pvq, p or q), and implication (p→q, p implies q) operators.

# Logic Refresher

## Propositional Calculus (cont.)

| p | q | p&q | pvq | p_ q |
|---|---|---|---|---|
| T | T | T | T | T |
| T | F | F | T | F |
| F | T | F | T | T |
| F | F | F | F | T |

A formal means of determining whether more complex expressions are constructed properly is given by the following recursive definition of *well-formed-formulas* (wffs):

1. A proposition is a wff.
2. If A and B are wffs, then so are (-A), (A&B), (AvB), and (A→B).
3. There are no other wffs.

# Logic Refresher

Predicate Calculus extends propositional calculus permitting individuals, relations between individuals, and properties of individuals and sets of individuals. We continue to denote propositions by p, q, r, ... and to use the same set of unary and binary logical connectives.

To those structures we add individual constants, denoted a, b, c, ..., which symbolically identify particular items of the domain of discourse, D (e.g., people, numbers, days of the week). We use the last letters of the alphabet (z, y, x, ...) to denote individual variables which may range over all the individuals in D. Functions of one or more variables and/or constants will be denoted f, g, h, ... and will map objects or groups of objects in D into other objects in D. Thus in the domain of numbers we might represent negation by g(x), addition by f(x,y), and three way multiplication by h(x,y,z), so that h(g(2), 6, f(3,1)) would denote -48. Any expression of this sort, which evaluates to an object or set of objects in D, is known as a term. Defined recursively, a term is (1) a constant, (2) a variable, or (3) a function of terms.

# Logic Refresher

Predicate calculus gets its name from the entities used to describe or relate terms. Predicates are denoted P, Q, R, ... and map terms onto the truth values T and F. Thus, if D is people, P(a) might assert that individual a has red hair, while R(c,b) might claim that b is a sibling of c. Any predicate of terms (or simple proposition) in the predicate calculus is known as an atomic formula. The last group of PC entities consists of two quantifiers. The universal quantifier, denoted (x) and read for all x, when applied to a formula asserts that the formula is true for all possible substitution instances of the variable x (the entire domain D). The existential quantifier, denoted (∃x) and read there exists an x, asserts that the formula is true for at least one of the possible values of x. In general a quantifier does not apply to all occurrences of its variable in a formula but only to those which fall within its range or scope (delimited if necessary by appropriate parentheses). Such variables are said to be bound by the quantifier while other occurrences of the same variable may be free of quantification.

# Logic Refresher

We define recursively the well-formed-formulas (wffs) of the PC, as follows:

1. Any atomic formula is a wff.
2. If A and B are wffs then so are (-A), (A&B), (AvB), and (A→B).
3. If A is a wff and x is a (free) variable in A, then ((x)A) and ((∃x)A) are wffs.
4. There are no other wffs.

Quantifiers are to be evaluated first, along with negations. Thus the scope of (x) in (x)-P(x)vQ(x) is just -P(x); the x in Q(x) is a free variable.

Interpretation of PC formulas requires specification of the domain, D, an assignment of elements of D to individual constants, and assignments of meanings (mappings) with respect to D to all functions and predicates.

Just as for the propositional calculus, PC formulas are classed as valid (true for all interpretations), satisfiable (true for at least one interpretation) and inconsistent (true for no interpretations). Two predicate formulas are equivalent ⇔ they have identical truth values under all interpretations.

# Prenex normal form

A useful type of formula equivalent to any predicate calculus formula is its prenex normal form. In this form all quantifiers have been swept to the front of the formula, so that each of them has all the rest of the formula (called the matrix) as its scope. The most awkward aspect of converting formulas to prenex normal form can be moving negation through quantifiers where the following (sensible) equivalences apply:

$$-(x)A=(\exists x)-A, \quad -(\exists x)A=(x)-A$$

Since conversion to prenex normal form is an implicit step in preparing formulas for resolution, we will illustrate the method next.

# Clause form

In 1965 the logician J. A. Robinson reported the development of a new inference rule for the predicate calculus. He also proved that his *resolution principle* was sound (producing only valid wffs) and complete (producing all valid wffs). While not especially convenient or intuitive for people, the resolution principle is ideally suited to computer implementation and forms the basis for almost all current research in theorem proving, logic programming and computational linguistics.

## Unification & Resolution

A proof that some formula W logically follows from a set of formulas S is equivalent to the claim that every interpretation satisfying S also satisfies W. If such is the case then no interpretation can satisfy the union of S and -W. Resolution theorem proving tries to show that union is unsatisfiable by deriving a special formula called the null clause or resolvent from it. The method is thus a special form of proof by contradiction.

Before resolution theorem proving can be applied to a theorem, preliminary steps must be executed. Premises and the conclusion to be proved stated in English must be expressed in PC. Second, the conclusion to be proved must be negated. Third, all formulas including the negated conclusion must be converted to *clause form*, a formula in prenex normal form with no quantifiers shown because existential quantifiers have been eliminated and all variables are assumed to be universally quantified. The matrix of a clause consists solely of disjunctions of atomic formulas and their negations, known collectively as *literals*. Conversion to clause form is by the 8 algorithm.

# The Eight-Step Algorithm

Using the unusually complex formula

(x)[P(x)→[(y)Q(x,y)&-(y)(P(y)→R(f(x,y)))]]. [●]

Step 1: Eliminate Implication Signs - Using p→q = -pvq, [●] becomes

(x)[-P(x)v[(y)Q(x,y)&-(y)(-P(y)vR(f(x,y)))]]

Step 2: Reduce Scopes of Negation Signs - We then use equations p&q = -(-pv-q), pvq = -(-p&-q) and -(x)A=(∃x)-A, -(∃x)A=(x)-A to reduce the scopes of negation signs to single predicates:

(x)[-P(x)v[(y)Q(x,y)&(∃y)(P(y)&-R(f(x,y)))]]

Step 3: Standardize Variables - Now we rename quantified variables, if necessary, so that each quantifier has a unique variable:

(x)[-P(x)v[(y)Q(x,y)&(∃z)(P(z)&-R(f(x,z)))]]

## The Eight-Step Algorithm

Using the unusually complex formula

(x)[P(x)→[(y)Q(x,y)&-(y)(P(y)→R(f(x,y)))]]. [🍎]

Step 4: Eliminate Existential Quantifiers - For all such quantifiers which do not fall within the scope of universal quantifiers we may simply replace (∃w)P(w) with P(a) where 'a' is a constant whose existence the quantifier asserts. In a case like (v)(∃w)Q(w), there is some (possibly distinct) w for every v, so we must write (v)Q(h(v)) where h is a function that selects the w which exists for each v. Constants and functions introduced in this step must be new to the formula. [functions introduced here are called Skolem functions]. Our example becomes:

(x)[-P(x)v[(y)Q(x,y)&(P(g(x))&-R(f(x,g(x))))]]

- Step 5: Convert to Prenex Form - This conversion is accomplished by moving all (universal) quantifiers to the front of the formula:
  - (x)(y)[-P(x)v[Q(x,y)&P(g(x))&-R(f(x,g(x))))]]

## The Eight-Step Algorithm

Using the unusually complex formula

(x)[P(x)→[(y)Q(x,y)&-(y)(P(y)→R(f(x,y)))]]. [●]

Step 6: Put Matrix in Conjunctive Normal Form - Converting from prenex form to conjunctive normal form yields

- (x)(y)[(-P(x)vQ(x,y))&(-P(x)vP(g(x)))&(-P(x)v-R(f(x,g(x))))]

- Step 7: Eliminate Universal Quantifiers - Dropping the universal quantifiers (we assume that all variables at this point are universally quantified) leaves us
  - [(-P(x)vQ(x,y))&(-P(x)vP(g(x)))&(-P(x)v-R(f(x,g(x))))]
- Step 8 :Eliminate & Signs - Eliminate the conjunctions by separating the formula into distinct clauses, each of which will be a disjunction of literals:
  - -P(x)vQ(x,y)    -P(x)vP(g(x))    -P(x)v-R(f(x,g(x)))

## Unification Algorithm

- Given a set of clauses derived from the premises and negated conclusion of a theorem, the resolution principle generates new clauses by resolving pairs of clauses in the set. These new clauses are added to the set and may be used in the generation of further resolvents. If the original set of clauses is unsatisfiable (the theorem is provable) resolution will eventually produce a clause containing no literals, the so-called null resolvent.
- To produce a resolvent of two available clauses we require that at least one atomic formula appear with opposite signs in the two parent clauses. The resolvent then consists of a disjunction of all other literals in both parent clauses, after removal of the literal(s) differing only in sign. Thus from the clauses -P(x)vR(x) and -R(x)v Q(x) we may infer the resolvent -P(x)vQ(x) by combining the literals left after removing R(x) and -R(x).

## Unification Algorithm

- In the example the coincidental appearance of R(x) and -R(x) in the parent clauses was fortunate. Usually it is necessary to perform substitutions in the parent clauses. The process of finding suitable substitutions is called unification. If a set of clauses can be unified (i.e., can produce resolvents), a procedure called the unification algorithm can be used to find the simplest substitution (or most general unifier) that does the job. The details of unification are given now.
- The terms of a literal can be variable letters, constant letters, or expressions consisting of function letters and terms. A substitution instance of a literal is obtained by substituting terms for variables in the literal. Thus four instances of P(x,f(y),b) are
  - P(z,f(w),b)
  - P(x,f(a),b)
  - P(g(z),f(a),b)
  - P(c,f(a),b)

# Unification Algorithm

- The first instance is called an alphabetic variant of the original literal because we have merely substituted different variables for the variables appearing in P(x,f(y),b). The last of the four instances mentioned above is called a ground instance or atom since none of the terms in the literal contains variables.
- In general, we can represent any substitution by a set of ordered pairs $\theta = \{(t_1,v_1), (t_2,v_2),\ldots, (t_n,v_n)\}$. The pair $(t_i,v_i)$ means that the term $t_i$ is substituted for variable $v_i$ throughout. We insist that a substitution be such that each occurrence of a variable have the same term substituted for it; that is $i \neq j$ implies $v_i \neq v_j$, $i,j=1,\ldots,n$. The substitutions used above in obtaining the four instances of P(x,f(y),b) are
    - $\alpha = \{(z,x), (w,y)\}$
    - $\beta = \{(a,y)\}$
    - $\gamma = \{(g(z),x), (a,y)\}$
    - $\delta = \{(c,x), (a,y)\}$

# Unification Algorithm

- To denote a substitution instance of a literal P using a substitution θ, we write P:θ. Thus P(z,f(w),b) = P(x,f(y),b):α. The composition of two substitutions α and β is denoted by α|β and is the substitution obtained by applying β to the terms of α and then adding any pairs of β having variables not occurring among the variables of α. Thus
    - {(g(x,y),z)}{(a,x),(b,y),(c,w),(d,z)}={(g(a,b),z),(a,x),(b,y),(c,w)}
- It can be shown that applying α and β successively to a literal P is the same as applying α|β to P, that is, (P:α):β = P:α:β. It can also be shown that the composition of substitutions is associative:
    - (α|β)|γ = α|(β|γ)
- If a substitution θ is applied to every member of a set $\{L_i\}$ of literals, we denote the set of substitution instances by $\{L_i\}$:θ. We say that a set $\{L_i\}$ of literals is unifiable if there exists a substitution q such that $L_1$:θ = $L_2$:θ = $L_3$:θ = etc. In such a case θ is said to be a unifier of $\{L_i\}$ since its use collapses the set to a singleton.

# Unification Algorithm

As an example, θ = {(a,x), (b,y)} unifies {P(x,f(y),b), P(x,f(b),b)} to yield {P(a,f(b),b)}.

- Although θ = {(a,x), (b,y)} is a unifier of the set {P(x,f(y),b), P(x,f(b),b)}, in some sense it is not the simplest unifier. We really did not have to substitute a for x to achieve unification. The most-general (or simplest) unifier [mgu] λ of $\{L_i\}$ has the property that if θ is any unifier of $\{L_i\}$ yielding $\{L_i\}$:θ, then there exists a substitution δ such that $\{L_i\}$:λ|δ = $\{L_i\}$:θ. Furthermore, the common instance produced by a most-general unifier is unique except for alphabetic variants.
- There is an algorithm called the unification algorithm that produces a most-general unifier λ for any unifiable set $\{l_i\}$ of literals and reports failure when the set is not unifiable.

# Unification Algorithm

The algorithm starts with the empty substitution and constructs, step-by-step, a most general unifier if one exists. Suppose at the $k^{th}$ step, the substitution so far produced is $\lambda_k$. If all the literals in the set $\{L_i\}$ become identical after employing the substitution $\lambda_k$ on each of them then $\lambda = \lambda_k$ is a most-general unifier of $\{L_i\}$. Otherwise we regard each of the literals in $\{L_i\}$: $\lambda_k$k as a string of symbols and detect the first symbol position in which not all of the literals have the same symbol. We then construct a *disagreement set* containing the well-formed expressions from each literal that begins with this symbol position. (A well-formed expression is either a term or a literal). Thus, the disagreement set of

{P(a,f(a,**g(z)**),h(x)),P(a,f(a,**u**),g(w))} is {g(z),u}

Now the algorithm attempts to modify the substitution $\lambda_k$ in such a way as to make two elements of the disagreement set equal. This can be done only if the disagreement set contains a variable that can be set equal to one of its terms. (If the disagreement set contains no variables at all, $\{L_i\}$ cannot be unified. For example, we note that at the first step of the algorithm the disagreement set may be $\{L_i\}$ itself, and then certainly then no element is a variable).

# Unification Algorithm

Let $s_k$ be any variable in the disagreement set and let $t_k$ be a term (possibly another variable) in the disagreement set such that $t_k$ does not contain $s_k$. (If no such $t_k$ exists, then again $\{L_i\}$ is not unifiable). Next we create the modified substitution $\lambda_k+1 = \lambda_k\{(t_k,s_k)\}$ and perform another step of the algorithm.

It can be proven (Robinson, 1965) that the unification algorithm finds a most-general unifier of a set of unifiable literals and reports failure when the literals are not unifiable.

As examples, we list the most common substitution instances (those obtained by the mgu) for a few sets of literals.

# Unification Algorithm

| **Set Of Literals** | **Most-general Common Substitution Instances** |
|---|---|
| {P(x), P(a)} | P(a) |
| {P(f(x),y,g(y)), P(f(x),z,g(x))} | P(f(x),x,g(x)) |
| {P(f(x,g(a,y)),g(a,y)),P(f(x,z),z)} | P(f(x,g(a,y)),g(a,y)) |

We consider the legal substitutions that may be made in a pair of clauses without altering their truth values. In order to avoid confusion (and error) from coincidentally identical variable names, substitution should be applied to clauses which have no variable names in common. If this is not already the case we simply rename some or all of the variables in one of the clauses. Now since all variables are understood to be universally quantified, each specifies any object in the domain. We can therefore substitute any new or existing variable name for all of the occurrences of any given name in order to bring literals in the clauses into closer correspondence.

# Unification Algorithm

We can substitute any constant or function for all the instances of any variable in the two clauses, since such substitutions simply limit the range to one or more of the objects for which the variable stood. We cannot however make any substitutions which would change or increase the identified set of objects, since such substitutions could alter the truth value of the clause. Thus we may not substitute variables for functions or constants, nor may we replace any constant or function with any other constant or function.

To illustrate how substitution can be used in producing resolvents, consider the clauses

(1) -P(a) v Q(f(x),y,c) v R(y)

(2) S(x,y) v P(x) v -Q(y,b,c).

Renaming variables, by application of primes to variables in (2) which also happen to appear in (1), gives us

(2a) S(x',y') v P(x') v -Q(y',b,c).

Now we can substitute a for x' in (2a) producing

(2b) S(a,y') v P(a) v -Q(y',b,c).

# Unification Algorithm

which can be resolved with (1) to give

(3) Q(f(x),y,c) v R(y) v S(a,y') v -Q(y',b,c)

Alternatively we might substitute b for y in (1) and f(x) for y’ in (2a), giving the different resolvent

(4) -P(a) v R(b) v S(x',f(x)) v P(x')

Thus different substitutions can give different resolvents. It should also be noted that (3) and (4) can be further resolved against the original formulas, with appropriate further substitutions.

## Unification Algorithm - example

Consider the following theorem: If there are no compassionate professors, and if all competent professors are compassionate, then no competent professor exists. If we let S(x) indicate that x is compassionate, and P(x) that x is competent, then the predicate calculus formulas for the premise are

(1) $-(\exists x)S(x)$

(2) $(y)(P(y) \rightarrow S(y))$,

while the denial of the conclusion is $- -(\exists z)(P(z))$ or just

(3) $(\exists z)(P(z))$.

(Note that we have avoided duplication of variable names to reduce the necessity for renaming prior to substitution.) In clause form,

(1') $-S(x)$ (2') $-P(y) \vee S(y)$ (3') $P(a)$

# Unification Algorithm - another example

Substitution can be used in producing resolvents; consider the two clauses

(1) -P(a) v Q(f(x),y,c) v R(y)

(2) S(x,y) v P(x) v -Q(y,b,c).

Renaming variables, by application of primes to variables in (2), which also happen to appear in (1), gives us

(2a) S(x',y') v P(x') v -Q(y',b,c).

Now we can substitute a for x' in (2a) producing

(2b) S(a,y') v P(a) v -Q(y',b,c).

which can be resolved with (1) to give

(3) Q(f(x),y,c) v R(y) v S(a,y') v -Q(y',b,c)

# Unification Algorithm - another example

Alternatively, we might substitute b for y in (1) and f(x) for y' in (2a), giving the different resolvent

(4) -P(a) v R(b) v S(x',f(x)) v P(x')

Thus different substitutions can give different resolvents. It should also be noted that (3) and (4) can be further resolved against the original formulas with appropriate further substitutions.

With substitution of x for y in (2'), resolution of (1') and (2') yields just -P(x). Substituting a for x in this resolvent and using (3') as the other parent yields the null resolvent, proving the theorem.

# Answer extraction using unification

Consider the following: If Marcia goes wherever John goes, and John is at school, where is Marcia? The facts might simply be translated into the set S of wffs

1. $(x)\{AT(John,x) \rightarrow AT(Marcia,x)\}$ and 2. $AT(John,school)$

where the predicate letter AT is interpreted obviously. The question where is Marcia? could be answered if we could first prove that the wff

$$(\exists x)AT(Marcia,x)$$

followed from S and could then find an instance of the x that exists. If the question can be answered from the facts given, the wff created in this manner will logically follow from S. After obtaining a proof, we then try to extract an instance of the existentially quantified variable to serve as an answer.

# Answer extraction using unification

The proof is obtained by first negating the wff to be proved, adding this negation to the set S, converting all of the members of this enlarged set to clause form, and then, by resolution, showing that this set of clauses is unsatisfiable. A refutation tree for our example is shown in Figure 3-1. The wff to be proved is called the conjecture and the clauses resulting from the wffs in S are called axioms. Note that the negation of (∃x)AT(Marcia,x) produces (x)[-AT(Marcia,x)] whose clause form is simply -AT(Marcia,x).

## Answer extraction using unification

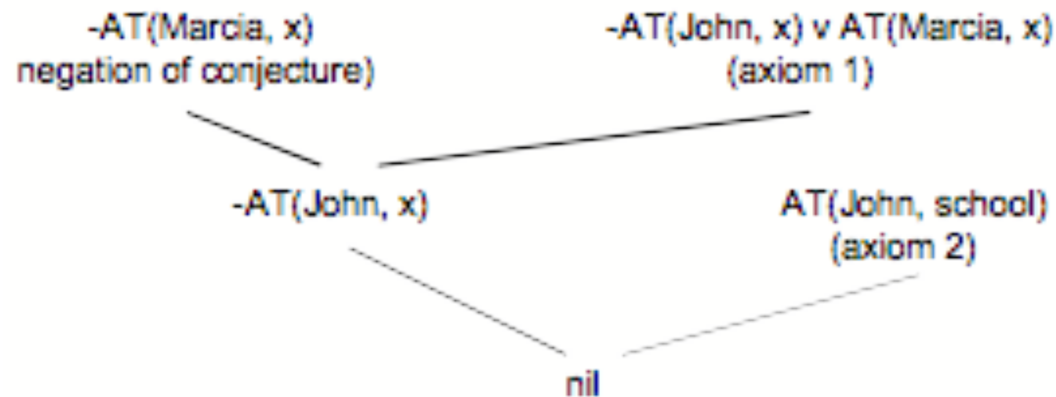



*Figure 3-1. Refutation Tree for example problem.*

Next, we must extract an answer to the question Where is Marcia? from this refutation tree. The process for doing so in this case is as follows:

# Answer extraction using unification

1. Append to each clause arising from the negation of the conjecture its own negation. Thus -AT(Marcia,x) becomes the tautology -AT(Marcia,x)vAT(Marcia,x).
2. Following the structure of the refutation tree, perform the same resolution as before until some clause is obtained at the root. In our example this process produces the proof tree shown in the figure below with the clause AT(Marcia,school) at the root.
3. Convert the clause at the root back to the conventional predicate calculus form and use it as an answer statement. This wff can then be translated back into English, say, as an answer to the question. In our example it is obvious that AT(Marcia,school) is the appropriate answer to the problem

# Answer extraction using unification

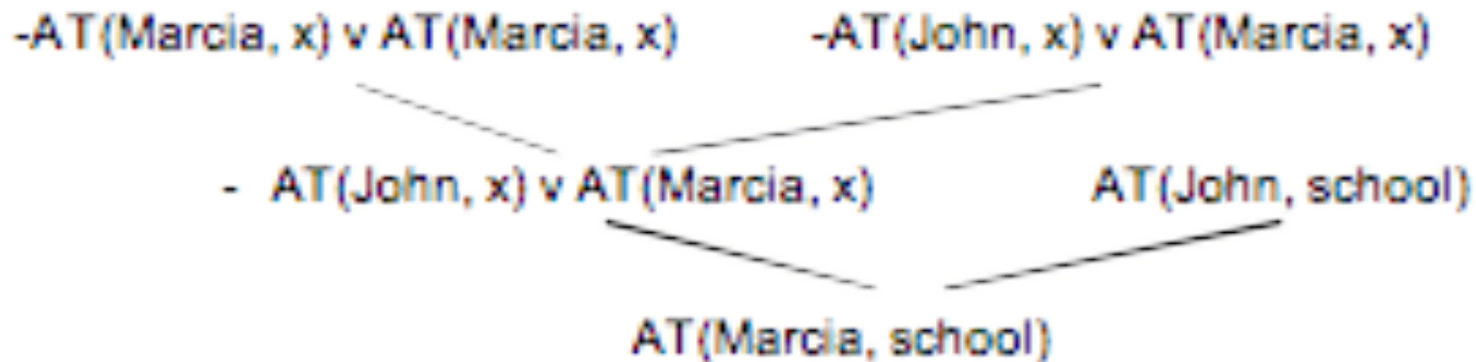



*Figure 3-2. Modified Proof Tree for example problem.*

## Concluding Remarks

Out of time

My old clock used to tell the time

and subdivide diurnity;

but now it's lost both hands and chime

and only tells eternity.