

author

Michal Měchura

assignment title

(none – multiple questions)

degree

M.Phil. in Speech and Language Processing

module

LI 7872 Formal Foundations of Linguistic Theories

term

Michaelmas 2007

word count

4,003 words

Task 1

“Detail, assisting the discussion with positive and negative examples, what it means to provide a finite recursive syntax and corresponding compositional semantics to an infinite logical language.”

This essay is subdivided into two sections. The first section deals with recursive syntax definitions, the second with compositional semantics. For reason of length and simplicity, both sections draw mainly on propositional logic to illustrate the concepts dealt with, although references first-order predicate logic is also referred to throughout.

1. Syntax

In computational linguistics, a language is a set of all sentences that can be expressed in that language. A logical language is then a set of all well-formed logical expressions, including the expressions $(a \wedge b)$, $((a \wedge \neg b) \rightarrow c)$, $\forall x(Hx \rightarrow Px)$, and so on. If the language we wish to define is finite (i.e. the number of well-formed expressions in that language is finite), then the set can be defined by simply listing all the well-formed expressions. More typically, logical languages are infinite (i.e. there is an unlimited number of well-formed expressions) and such sets can be defined by supplying a recursive membership function.

Defining a set recursively involves declaring that a collections of some primitive expressions are members of the set, and supplying some rules for deriving further expressions from the primitives. Expressions such derived are also members of the set, and as such, the same rules can be applied to them again to derive further expressions. This process can continue indefinitely, and thus an infinite set has been defined. The following is an example of a (rather minimalist) logical language defined recursively:

- (1) 1. The expression a is a member of the language L .
2. If ϕ is a member of the language L , then so is $\neg\phi$.

3. No other expressions than those sanctioned by the rules above are members of the language L .

Even such a simple definition can generate an infinite set with the members ϕ , $\neg\phi$, $\neg\neg\phi$, and so on. It is important to note, at this stage, that the definition in (1) simply specifies well-formed formulas regardless of their meaning. We may know that the expressions ϕ and $\neg\neg\phi$ have the same meaning in propositional logic but that is irrelevant here. From a syntactical point of view, both are distinct, well-formed expressions in the logical language we are defining. What they represent from a semantic point of view is another matter and will be discussed later.

In practice, recursive definitions of logical languages tend to be more complex than (1), involving a larger number of derivation rules – such as rules for two-place operators like *and*, *or* and *implies* – and a larger number of primitives. The collection of primitives may even be an infinite set itself and may be defined recursively (for example: if p is a primitive, then so are p' , p'' , and so on). Example (2) below demonstrates a more realistic definition of a language L of propositional logic.

- (2) 1. (a) P is a set of atomic formulas: $P = \{a, b, c, \dots\}$.
 - (b) L is a set of well-formed formulas.
 - (c) $P \subseteq L$.
2. (a) If $\phi \in L$, then $\neg\phi \in L$.
 - (b) If $\phi \in L$ and $\psi \in L$, then $(\phi \wedge \psi) \in L$.
 - (c) If $\phi \in L$ and $\psi \in L$, then $(\phi \vee \psi) \in L$.
 - (d) If $\phi \in L$ and $\psi \in L$, then $(\phi \rightarrow \psi) \in L$.
3. No other formulas than those sanctioned by the rules above are members of the set L .

If the language we wish to define is a language of first-order predicate logic, then the primitives and the rules will be specified with reference to several different types of objects, including variables, constants and predicates.

In definitions like (2), it is important to distinguish between the object language and the metalanguage. The object language is the language we are defining, in this

case logic. The metalanguage is the language in which the definitions are expressed: in this case a combination of English and notation from set theory.

The significance of a recursive syntax is not only its ability to generate an infinite number of well-formed expressions, but also to decide whether any given expression is well-formed or not. Determining whether a formula is well-formed is a matter of conducting a proof, using the recursive definition as an axiomatic system. Example (3) below is a proof that $((a \wedge b) \vee c)$ is a member of the propositional logic language L defined in (2).

(3)	1.	$a \in L$	given
	2.	$b \in L$	given
	3.	$c \in L$	given
	4.	$\forall \phi \forall \psi ((\phi \in L) \wedge (\psi \in L) \rightarrow ((\phi \wedge \psi) \in L))$	given
	5.	$\forall \phi \forall \psi ((\phi \in L) \wedge (\psi \in L) \rightarrow ((\phi \vee \psi) \in L))$	given
	7.	$(a \in L) \wedge (b \in L)$	1, 2, \wedge I
	8.	$(a \wedge b) \in L$	4, 7, \rightarrow E
	9.	$((a \wedge b) \in L) \wedge (c \in L)$	3, 8, \wedge I
	10.	$((a \wedge b) \vee c) \in L$	5, 9, \rightarrow E

Notice that predicate logic is being used in (3) in two incarnations: once as the object language, and once as a subset of the metalanguage. Expressions belonging to the object language (and variables that represent them) are highlighted in red in (3).

2. Semantics

As mentioned above, a recursive definition of syntax simply provides a means of obtaining well-formed formulas, regardless of their meaning. To have access to the semantics of a logical language, an additional definition must be provided that allows us to determine the meaning of any well-formed formula compositionally.

In a compositional semantics, the meanings of non-atomic formulas are “composed” from the meanings of the atomic formulas. So, for example, if the semantic value of the atomic formula a is *true* and the semantic value of the atomic

formula b is *false*, then our compositional semantics will conclude that the non-atomic formula $(a \vee b)$ has *true* as its semantic value. The definition of compositional semantics must be recursive – similarly to the syntax – so that the meanings of complex non-atomic formulas can be derived from the meanings of less complex non-atomic formulas. It follows that our compositional semantics must have a rule for each rule the syntactic definition has, so that the meanings of *all* possible well-formed formulas can be determined.

The following is an example definition of compositional semantics for the propositional logical language whose syntax was defined in (2). The notation $\llbracket x \rrbracket$ means “the semantic value of x ”.

(4) For every well-formed formula π in the language L :

(a) If π has the form $\neg\phi$, then:

$\llbracket \pi \rrbracket = \text{true}$ if $\llbracket \phi \rrbracket = \text{false}$, otherwise $\llbracket \pi \rrbracket = \text{false}$.

(b) If π has the form $(\phi \wedge \psi)$, then:

$\llbracket \pi \rrbracket = \text{true}$ if $\llbracket \phi \rrbracket = \text{true}$ and $\llbracket \psi \rrbracket = \text{true}$, otherwise $\llbracket \pi \rrbracket = \text{false}$.

(c) If π has the form $(\phi \vee \psi)$, then:

$\llbracket \pi \rrbracket = \text{true}$ if $\llbracket \phi \rrbracket = \text{true}$ or $\llbracket \psi \rrbracket = \text{true}$, otherwise $\llbracket \pi \rrbracket = \text{false}$.

(d) If π has the form $(\phi \rightarrow \psi)$, then:

$\llbracket \pi \rrbracket = \text{true}$ if $\llbracket \phi \rrbracket = \text{false}$ or $\llbracket \psi \rrbracket = \text{true}$, otherwise $\llbracket \pi \rrbracket = \text{false}$.

Once again, this definition and others like it makes use of two languages: the object language is propositional logic and the metalanguage is English enhanced with notational devices such as the equal sign. Additionally, there are the semantic values *true* and *false*: these give the truth or falsity of the formula with respect to some model.

The semantic rules in (4) are recursive, just like the syntactic rules in (2). To find the meaning of, for example, $((a \wedge b) \vee c)$, one first applies **semantic** rule (c), effectively decomposing the formula into the subformulas $\phi = (a \wedge b)$ and $\psi = c$, as if using **syntactic** rule (2c) from example (2). Supposing that $\llbracket c \rrbracket$ is known, one still needs to find $\llbracket (a \wedge b) \rrbracket$. This can be found by applying rule (b), again effectively

decomposing the formula into its subformulas, which are now atomic formulas whose semantic values are known. In this way, the semantic value of the whole formula can be “composed” from the semantic values of the atomic formulas.

The semantic definition corresponds to the syntactic one in more than just their recursiveness. There is exactly one semantic rule for each syntactic rule. This means that the meanings of all well-formed formulas can be determined, and that all well-formed formulas have meanings. A compositional semantics thus provides a way of declaring recursively composed formulas as true or false in relation to a particular model.

Task 2

“Discuss the notion of semantic expressivity in formal logics demonstrating where the language and logic provided by Prolog fits into a semantic expressivity hierarchy.”

This essay is subdivided into two sections. The first section deals with the notion of semantic expressivity generally and sets up a semantic expressivity hierarchy among logical and natural languages. The second section discusses where Prolog fits in this hierarchy.

1. Semantic expressivity

As Partee et al. (1993, p. 95) explain, logical languages have the facility to prove certain arguments as valid or invalid without recourse to their meaning. For example, the language of first-order predicate logic can demonstrate the argument in (1) as valid, regardless of the meanings of the predicate of A, B and C.

$$(1) \quad \frac{A_j}{B_j} \quad \forall x(Ax \rightarrow Bx)$$

This is a powerful feature of logical languages but it comes at a price. Aspects that are not part of the vocabulary or the grammar of the language cannot be exploited to demonstrate the validity or invalidity of arguments. The argument in (1) can only be expressed in propositional logic if we ignore certain details. Because propositional logic has neither quantifiers nor variables nor predicates, the three formulas A_j , B_j and $\forall x(Ax \rightarrow Bx)$ would all need to be reduced to simple atomic formulas, say as a , b and u , gaining the argument in (2).

$$(2) \quad \frac{a}{b} \quad u$$

This argument cannot be proven as valid or invalid in propositional logic alone because the language of propositional logic lacks the means to express all relevant aspects of the argument.

We can therefore conclude that first-order predicate logic is a superset of propositional logic, as far as their expressivity is concerned: it can express everything propositional logic can, but can express more. But even first-order predicate logic itself can be surpassed by other logical languages in its semantic expressivity, and eventually those can all be surpassed by the semantic expressivity of natural language. For example, a natural language like English can be demonstrated as more expressive than first-order predicate logic. A natural-language sentence like “John gave Peter a book” (Gamut 1991, p. 80) appears to be expressible in first-order predicate logic as the formula $\exists x(Bx \rightarrow Gjxp)$ – where Bx means “ x is a book”, $Gabc$ means “ a gave b to c ”, j is John and p is Peter. The problem is that the predicate-logic formula is true even if John gave Peter more than one book, while the natural-language sentence makes the subtle suggestion (without saying so explicitly) that just one book was given. Natural language is able to express such vagueness but predicate logic lacks the means to.

If natural languages are more expressive than logical languages, then we could perhaps question the point of inventing logical languages in the first place. Setting aside the fact that many logical languages are created expressly to *avoid* the vagueness of natural languages, the main reason why logical languages exist has already been stated at the start: it is to be able to prove the validity or invalidity of certain arguments without recourse to their meaning. It is not at all certain whether a logical language could be created which would be as expressive as natural languages while retaining this facility (Partee et al. 1993, p. 96).

2. Prolog

Of all the well-formed formulas of first-order predicate logic, only a subset of those can be expressed as a Prolog knowledge base. There are two important constraints

that cause this: firstly, Prolog knowledge bases can only contain *definite clauses*, and secondly, all such clauses are implicitly universally qualified. I will now briefly review these two constraints.

Following the definitions in Pereira and Shieber (1987, p. 14–16), a *definite clause* is a type of *Horn clause*, and a *Horn clause* is a type of *clause*. A clause is a type of formula in first-order predicate logic which is a disjunction of any number of *literals*. A literal is either an atomic formula or a negated atomic formula. For example, the formula in (3) is a clause where each of the positive literals p_1 , p_2 and the negative literals n_1 , n_2 stands for any valid predicate of first-order predicate logic, such as *Rabx*.

$$(3) \quad p_1 \vee p_2 \vee \neg n_1 \vee \neg n_2$$

Clauses can be re-expressed as implications, as in (4).

$$(4) \quad \begin{array}{ll} 1. & p_1 \vee p_2 \vee \neg n_1 \vee \neg n_2 \qquad \text{given} \\ 2. & (p_1 \vee p_2) \vee \neg(n_1 \wedge n_2) \qquad 1, (\neg P \vee \neg Q) \Leftrightarrow \neg(P \wedge Q) \\ 3. & (n_1 \wedge n_2) \rightarrow (p_1 \vee p_2) \qquad 2, (\neg P \vee Q) \Leftrightarrow (P \rightarrow Q) \end{array}$$

Further, a Horn clause is a clause that has at most one positive literal. The three clauses in (5) are all Horn clauses, while the clause in (3) and (4) is not.

$$(5) \quad \begin{array}{l} (a) \quad p \\ (b) \quad (n_1 \wedge n_2) \rightarrow p \\ (c) \quad n_1 \wedge n_2 \end{array}$$

Of these, (5a) and (5b) are definite clauses. (5a) can be expressed in a Prolog knowledge base as the fact (6a), and (5b) can be expressed as the rule (6b):

$$(6) \quad \begin{array}{l} (a) \quad p. \\ (b) \quad p :- n1, n2. \end{array}$$

The Horn clause in (5c) is not a definite clause and cannot appear in a Prolog knowledge base, but can be used as a query (or goal) in a Prolog proof.

All this means that some formulas of first-order predicate logic cannot be expressed in a Prolog knowledge base. For example, the formula $b \vee c$, even though it is a clause, cannot be expressed in Prolog because it has two positive literals, and

is therefore not a Horn clause. Similarly, the formula $a \rightarrow (b \wedge c)$ is not a clause at all, and cannot be re-expressed as one, and so cannot be expressed in Prolog either. Sometimes, a formula like (7a) cannot be re-expressed as a definite clause, but can be re-expressed as a conjunction of those (7b), and that conjunction can then be translated as a Prolog knowledge base containing two clauses (7c).

- (7) (a) $(a \vee b) \rightarrow c$
 (b) $(a \rightarrow c) \wedge (b \rightarrow c)$
 (c) $c :- a.$
 $c :- b.$

The second constraint that Prolog places on first-order predicate logic is the absence of quantifiers. In fact, all Prolog clauses are understood implicitly to be preceded by a universal quantifier for all variables that occur in it. For example, the rule “ $A(x) :- B(x).$ ” is really a Prolog equivalent of the formula $\forall x(Bx \rightarrow Ax)$. There is no existential quantification in Prolog, so it is impossible to express formulas like $\exists x(Bx \rightarrow Ax)$ in Prolog.

To conclude, a Prolog knowledge base is a conjunction of definite clauses, which are a subtype of first-order predicate logic formulas. As in the case of logical languages in general, the question may now be asked what is the point of Prolog if its logical language is limited. The answer is, once again, utility. As definite clauses are implications, they lend themselves easily to automated proof search (which in Prolog is essentially a recursive series of attempts to apply Modus Ponens). Also, any conjunction of definite clauses is guaranteed to be consistent (Pereira and Shieber 1987, p. 17), making sure that no Prolog knowledge base contains contradictions and that no contradictions can be derived from it.

3. Conclusion

This essay has demonstrated how propositional logic and the logic of Prolog are both (different) subsets of first-order predicate logic with respect to their expressivity. Logical languages can be arranged in a hierarchy of semantic

expressivity in which languages higher up in the hierarchy can express details which languages lower down cannot. The point of creating a logical language is to be able to conduct automated reasoning tasks, even if that means that the logical language has smaller expressivity than natural language.

Task 3

Explain the relationship between proof and entailment.

Following Partee et al. (1993, p. 90 – 91), a *formal system* is defined by (i) a set of objects called *primitives*, (ii) a set of statements about the primitives called *axioms*, and (iii) a set of rules for deriving further statements from the axioms. In the case of a logical system, (i) the primitives are all the well-formed formulas of the given logical language (such as propositional logic), (ii) the axioms are any formulas which are *given*, and (iii) the derivation rules are all rules of natural deduction, such as Modus Ponens, Modus Tollens and others.

The *proof* of a statement is the process of finding out whether the statement is derivable from the axioms by applying (repeatedly if necessary) the derivation rules. A proof may conclude that a statement is indeed derivable but that does not automatically imply that the statement is true. Proof is a purely syntactic activity, a mechanic manipulation of symbols without regard for their meaning.

The truth or falsity of a statement can only be stated with respect to a *model*. A model for a formal system contains a *discourse domain* and an *interpretation* which “translates” the system’s primitives into objects in the domain. The truth or falsity of a statement with respect to a model can only be found out by applying the interpretation to the statement and then “looking inside” the domain to see whether the interpreted statement is true there. This – and only this – is the way to find the truth or falsity of a statement. Strictly speaking, proof cannot reveal the truth or falsity of a statement, as it does not have access to a model.

Entailment, on the other hand, does have access to the model(s) of a formal system. Entailment is a relation which may or may not hold between two sets of statements, for example between the axioms of a formal system and some statements derived from them. If, upon applying an interpretation and examining a domain, we observe that the axioms are true in the domain and so are the derived

statements, then we can conclude that the axioms *entail* the derived statements in this particular model.

Many formal systems are set up such that the axioms and all derivable statements are guaranteed (or at least believed) to be true in a particular model. In such a system, any statement that can be *proven* from the axioms is also guaranteed (or at least believed) to be *entailed* by the axioms. This is when the utility of proof becomes apparent, as it can be employed to discover the truth or falsity of a statement “by proxy” – that is, without having to apply any interpretations and without having to examine the domain.

To summarize, proof and entailment are similar but different concepts. Proof is a syntactic process concerned with the relationships of statements to one another within a formal system, while entailment is a semantic relation concerned with the relationship between the statements of a formal system and the model of a formal system.

Task 4

Assuming a representation of letters of the alphabet as Prolog atoms and words as lists of letter representations, construct a general theory in Prolog which effectively defines rules of English pluralization. The theory should be effective for the below examples and other words in their category. It should not consist solely of finite enumeration, but general rules, where possible.

(a) *book/books*

(d) *hobby/hobbies*

(b) *fish/fish*

(e) *child/children*

(c) *dish/dish*

(f) *hoof/hooves*

This report is subdivided into three parts. In the first part, I will present the general design principles of my Prolog knowledge base. In the second part I will briefly discuss how extensively the knowledge base covers the pluralization of English. Finally, the third part, I will consider the issue of the reversibility.

1. Designing the knowledge base

I have decided to organize the rules of English pluralization at three levels, as follows:

- **The general rule.**

At the most general level, one can state that English plurals are formed by appending *-s* to the singular. This rule pluralizes correctly the noun *book* and others like it.

- **Patterns.**

Some families of nouns constitute exceptions to the general rule. Those can be generally recognized by their ending, and the plural is formed by replacing the ending with a different ending:

- *-y* → *-ies*, for example *hobby* → *hobbies*
- *-sh* → *-shes*, for example *dish* → *dishes*
- *-oof* → *-ooves*, for example *hoof* → *hooves*

- **Exceptions.**

Some nouns constitute exceptions to both the general rule and the patterns, and form their plurals in completely irregular ways:

- *fish* → *fish*
- *child* → *children*

My Prolog knowledge base implements these rules so that the correct plural can be derived for any of the nouns (a) – (f) as well as many others. The knowledge base is shown in Listing 1. It is invoked by posing the query `plural (S, P)` where `S` is the singular and `P` is the plural, such as `plural ([b, o, o, k], P)`. The relation `plural /2` holds for all pairs of arguments where the second argument is the plural of the first, and the first is the singular of the second. The relation `plural /2` searches for a solution to the query by posing the goal `plural _rule/2` with the same arguments. While attempting to unify the goal `plural _rule/2` with rules in the knowledge base, the program first attempts to match the singular with one of the exceptions, then with one of the patterns, and finally with the general rule.

As soon as a match is found, control is passed back to `plural /2`. This rule contains a cut (!) at the end of its body, causing the inference algorithm to stop searching for further matches. If the cut were not present, the knowledge base would return incorrect regular plurals for irregular nouns, in addition to the correct irregular ones. For example, the query `plural ([f, i, s, h], X)` would assign `X` in turn to `[f, i, s, h]`, `[f, i, s, h, es]` and `[f, i, s, h, s]`. Because of the cut, the query stops searching for further solutions after the first match, and returns only `[f, i, s, h]`.

2. Language coverage

The model of English pluralization used here seems to provide good coverage of the language's pluralization behaviour, and can be extended with new pattern exception and individual exceptions if necessary.

One possible limitation is that the knowledge base only returns at most one possible plural, when in fact several plurals might be equally valid. For example, both *fishes* and *fish* are (arguably) valid plurals of the singular *fish* as they are both attested in usage.

3. Reversibility

A few words are in order about the mechanism the knowledge base uses to match nouns with patterns. The relations on lines 13–15 succeed if there exists a stem such that, upon appending the ending `[y]`, `[s, h]` or `[o, o, f]` to the stem, the result is the singular noun queried for. If so, then the ending is discarded and a new one is appended to the stem to obtain the plural.

This method works well but it causes the relation to be irreversible in some cases. When searching for the singular of a plural noun that is not an exception or the first of the patterns, the inference algorithm will enter into an infinite loop. For example, when posing the query `plural(X, [d, i, s, h, e, s])`, the algorithm will attempt to match it against the pattern on line 13, indefinitely trying to create progressively longer and longer stems and trying to append `[i, e, s]` to them to obtain `[d, i, s, h, e, s]`. Obviously, this will never succeed. Prolog will loop on this rule indefinitely (or until the stack overflows) and will never even get to the rule on line 14, which is where the singular for *dishes* should really be obtained from.

In other words, some of the `plural_rule/2` relations are left-recursive and this causes them to be irreversible. The procedural meaning of the knowledge base here defeats its declarative meaning. To solve the problem, I have re-engineered the matching strategy to take advantage of Prolog's inherent feature of matching lists by their heads. The new knowledge base is shown in Listing 2 and works in the following way. When `plural/2` is invoked, the singular is reversed and then matched against one of the rules in `plural_rule/2`, which are now specified back-to-front (e.g. *foo-* instead of *-oof*). Once a match is found, the plural is created by discarding the head and pre-pending a new one, thus creating the (reversed) plural.

Finally, control is passed back to `plural/2` where the plural is reversed back and returned.

As this method is not left-recursive, it is reversible. A search for the singular of *dishes* will now return the correct *dish* immediately. However, it is still irreversible when given an invalid plural, such as when posing the query `plural(X, [a, b, c])` – the inference algorithm will indefinitely keep on inventing progressively longer and longer singulars that could possibly be turned into the plural `[a, b, c]`. This can be solved by adding a new rule at the bottom of the knowledge base, `plural_rule([], _)` (commented out in Listing 2) which simply means that, if no other singular can be found for a plural, then the empty list should be returned as the singular. This will stop the infinite loop.

4. Conclusion

While the overall design of the knowledge base appears valid, its detailed implementation in Prolog has posed some challenges. It is obvious that the procedural meaning of Prolog cannot be ignored, especially if one wishes his relations to be reversible.

Listing 1

```
1  %append relation:
2  appnd([], L, L).
3  appnd([H|T], L2, [H|L3]) :- appnd(T, L2, L3).
4
5  %relation for plurals:
6  plural(S, P) :- plural_rule(S, P), !.
7
8  %exceptions:
9  plural_rule([f,i,s,h], [f,i,s,h]).
10 plural_rule([c,h,i,l,d], [c,h,i,l,d,r,e,n]).
11
12 %patterns:
13 plural_rule(Sg, Pl) :- appnd(Stem, [y], Sg),
14                        appnd(Stem, [i,e,s], Pl).
15 plural_rule(Sg, Pl) :- appnd(Stem, [s,h], Sg),
16                        appnd(Stem, [s,h,e,s], Pl).
17 plural_rule(Sg, Pl) :- appnd(Stem, [o,o,f], Sg),
18                        appnd(Stem, [o,o,v,e,s], Pl).
19
20 %general rule:
21 plural_rule(Sg, Pl) :- appnd(Sg, [s], Pl).
```

Listing 2

```
1  %reverse, using an accumulator:
2  revrs(In, Out) :- revrs_acc(In, [], Out).
3  revrs_acc([], Acc, Acc).
4  revrs_acc([H|T], Acc, Ret) :- revrs_acc(T, [H|Acc], Ret).
5
6  %relation for plurals:
7  plural(S, P) :- revrs(S, RS),
8                  plural_rule(RS, RP),
9                  revrs(RP, P),
10                 !.
11
12 %exceptions:
13 plural_rule([h,s,i,f], [h,s,i,f]).
14 plural_rule([d,l,i,h,c], [n,e,r,d,l,i,h,c]).
15
16 %patterns:
17 plural_rule([y|T], [s,e,i|T]).
18 plural_rule([h,s|T], [s,e,h,s|T]).
19 plural_rule([f,o,o|T], [s,e,v,o,o|T]).
20
21 %general rule:
22 plural_rule(L, [s|L]).
23
24 %invalid plural rule:
25 plural_rule([], _).
```

References

- Gamut, L. T. F. (1991) *Logic, Language, and Meaning - Volume 1: Introduction to Logic*
London and Chicago: The University of Chicago Press
- Partee, B. H.; A. ter Meulen; R. E. Wall (1993) *Mathematical Methods in Linguistics*
London: Kluwer Academic Publishers
- Pereira, F. C. N.; S. M. Shieber (1987) *Prolog and Natural Language Analysis* Stanford:
Center for the Study of Language and Information