Chapter 1
A Meta-Rule Treatment for English Wh-Constructions

Lynette Hirschman[1]

Paoli Research Center
Unisys Defense Systems

Abstract

This paper describes a general meta-rule treatment of English wh-constructions (relative clauses, and questions) in the context of a broad-coverage logic grammar that also includes an extensive meta-rule treatment of co-ordinate conjunction. Wh-constructions pose difficulties for parsing, due to their introduction of a dependency between the wh-word (e.g., *which*) and a corresponding *gap* in the following clause: *This is the book which I thought you told me to refer to ( )*. The gap can be arbitrarily far away from the wh-word, but it *must* occur within the clause, or the sentence is not well-formed, as in **The book which I read it.*

A meta-rule treatment has several advantages over an Extraposition Grammar-style treatment: a natural delimitation of the gap scope, the ability to translate/compile the grammar rules, and ease of integration with conjunction. Wh-constructions are handled by annotating those grammar rules that license a gap or realize a gap. These annotations are converted, via the meta-rule component, into parameterized rules. A set of paired input/output parameters pass the need for a gap from parent to child and left sibling to right sibling until the gap is realized; once the gap is realized, the parameter takes on a *no_gap* value, preventing further gaps from being realized. This 'change of state' in the paired parameters ensures that each gap is filled exactly once. The conjunction meta-rule operates on the parameterized wh-rules to link gaps within conjoined structures by unification, so that any gap within a conjoined structure is treated identically for all conjuncts.

[1]This work has been supported in part by DARPA under contract N00014-85-C-0012, administered by the Office of Naval Research; and in part by internal Unisys funding.

1.1 Introduction

Wh-constructions are one of the classically difficult parsing problems, because a correct treatment requires interaction of non-adjacent constituents, namely the wh-word, which introduces a constituent in clause-initial position, and the following construction which is missing a constituent (the *gap*). The gap can be arbitrarily far from the introducing wh-word (an *unbounded dependency*); in particular, it can appear within deeply embedded constructions, such as *the person that [I had hoped [Jane would tell [( ) to get the books]]]*, where there are three levels of embedded structure. It is possible, in principle, to write a rule for each case where a gap can appear. However, since the number of constructions which can accommodate a gap is very large (e.g., most complement types), this is both extremely labor-intensive and unmaintainable from the grammar writer's point of view.

It is also possible to write general rules for gap-realization, e.g., a noun phrase can be realized as a gap. If this approach is taken, then these rules must be carefully constrained to accept gaps only when inside a wh-construction; in addition, the wh-construction must contain exactly *one* gap. These restrictions involve complex and expensive search up and down the parse tree, to determine whether a gap is occurring inside a wh-construction.

In many ways, the wh-problem parallels the problem of co-ordinate conjunction that has also been a major obstacle for natural language systems. Both constructions involve gaps, both affect large portions of the grammar, and both require a major modification to the grammar and/or to the parsing mechanism to handle the linguistic phenomena.

There have been two basic approaches to conjunction and wh-constructions in the computational linguistics literature: modification of the parser (interpreter) and meta-rules. Of these, the first approach has been far more common. For conjunction, a number of variants on the 'interrupt' driven approach have been presented, both in conventional natural language processing systems [13, 12, 14], and in the context of logic grammars [4]. The same is true for logic grammar implementations of wh-constructions: the most generally used treatment is the interpreter-based treatment of Extraposition Grammar (XG) [10].

Meta-rules offer an appealing alternative to interpreter-based approaches, both for conjunction and for wh-expressions. Meta-rules are particularly well-suited to phenomena that range over a variety of syntactic structures, where the linguistic description would otherwise require regular changes to a large set of grammar rules. The use of meta-rules turns out to be efficient computationally. It also preserves compactness of the underlying grammar, so that the grammar is still maintainable from the point of view of the grammar-writer. Finally, the meta-rule approach avoids additional interpretive overhead and permits translation/compilation of grammar rules for efficient execution [5].

For conjunction, the meta-rule approach forms the basis for a comprehensive

treatment of co-ordinate conjunction in Restriction Grammar [7]. Abramson has provided a generalization of this approach, formulating meta-rules as a specialized case of meta-programming [2]. Other researchers have also examined a meta-rule approach to related phenomena; Banks and Rayner, for example, have proposed a meta-treatment of the comparative [3].

For wh-constructions, we propose here an approach based on parameterization of the grammar rules. This is similar in spirit to the GPSG notion of 'slash categories' [6], but in the framework of logic grammar. The use of parameterized rules to pass gap information has previously been proposed in a logic grammar framework, specifically as *gap-threading* [10, 11]. Our approach differs from Pereira's in several ways, the most important of which is the use of meta-rules. The meta-rule approach provides a much cleaner user interface, making it possible for the grammar writer to use linguistically motivated annotations to indicate gap license and gap realization for the unparameterized BNF definitions in the grammar. The meta-rules process these annotations to generate parameterized grammar rules which, in turn, can be translated and compiled for efficient execution. The meta-rule treatment also has the property of combining seamlessly with a meta-rule treatment of co-ordinate conjunction.

1.2 Wh-Constructions: The Linguistic Issues

Wh-constructions are one instance of a class of problems referred to as *unbounded dependencies* – that is, constructions where the interdependent entities may be arbitrarily far apart. In the case of wh-constructions, we have a wh-expression which begins the clause (e.g., *who, what, which, whose book, how*, etc.) followed by a *gap* at some later point in the clause. The wh-expression may take the place of a noun phrase, an adjective phrase or an adverbial phrase. These may appear in the subject, object or sentence adjunct positions.

As the sentences of Figure 1.1 illustrate, there are a variety of wh-constructions, namely, relative clauses (including the zero-complementizer case, where an overt wh-word is absent, as in *the person I saw*), indirect questions (*I don't know what they mean*), wh-questions (*What do you want?*), and headless relatives (*You get what you deserve*). In addition to these basic types of wh-construction, there are also some constructions where the wh-expression is embedded inside a noun phrase (*this is the person whose mother I met*), with the wh-word *whose* modifying a noun phrase; the subsequent gap is filled by the noun phrase (*the person's mother*) of which the wh-word is a part. There are also wh-constructions embedded in prepositional phrases, as in *the person from whom I learned it* or *the door the key to which is missing.*

A wh-construction involves (1) a *wh-word* (e.g., *who*) contained in a clause-initial *wh-expression*; and (2) a *gap*: a constituent omitted in the clause following the wh-word, e.g., *the book which I bought ( )*. Relative clauses also have an antecedent for the relative pronoun (the wh-word); for questions, the wh-word

marks the questioned item.

Wh = who; gap = subject NP
The person who () was here
Wh = who(m); gap = object NP
Who did you see ()?
Wh = that; gap = object NP
The time that I spent ()
Wh = that; gap = sentence adjunct adverbial
The time I visited them ()
Wh = who(m); gap = embedded object
The person who they told me they had tried to visit ()
Wh = who; gap = embedded subject
Who did they tell you () had visited them?
Wh = how; gap = sentence adjunct adverbial
Do you know how they did it ()?

Figure 1.1: Wh-constructions in English

To regularize a wh-construction, the wh-expression fills in the gap, and the wh-word is replaced by its antecedent (if in a relative clause).[1] For example, in the phrase *the movie which I saw ( )*, the wh-expression is *which* and the gap is after *saw*. Moving the wh-expression into the gap, we get *the movie [I saw which]*. Then, replacing *which* by its antecedent (*the movie*), we get: *the movie [I saw the movie]*. Similarly for questions, we get *what did you see ( )?* regularized as *did you see what?*. In some cases, however, the wh-word is not identical to the whole wh-expression, as in *the bird whose nest I found ( )*. Here, the wh-expression is *whose nest*, and the wh-word is *whose*. Again, we replace the gap (the object of *found*) by the wh-expression, to get *the bird [I found whose nest]*. Then we replace the wh-word by its antecedent, namely *the bird*: *the bird [I found the bird's nest]*, preserving the possessive marker from *whose*. Similarly in a question, we get: *which book did you read ( )* regularized as *did you read which book?*. To summarize, wh-expressions are introduced by a phrase containing a wh-word; following a wh-expression, there must be a gap, and this gap is understood as the wh-expression, after it has had the antecedent of the wh-word word filled in (if in a relative clause).

[1]The expression *replace by its antecedent* is used loosely here. What is really meant is replacing the relative pronoun by a pointer to the antecedent. This preserves co-referentiality of the relative pronoun and its antecedent, and avoids the dangers of copying quantifier and other modifier information.

The need for a gap can be captured very simply by associating with each grammar definition a set of paired input/output parameters. The input parameter signals whether or not a gap is need when the node is about to be constructed, at rule invocation time. The output parameter signals whether that need has been satisfied once the node is completed, at rule exit. Thus an assertion in a relative clause has as its input parameter the need for a gap (**need_gap**) and on exit, that need must have been satisfied (indicated by a **no_gap** output parameter). These parameters, once set, are simply passed along from parent to child, and sibling to sibling, via unification through linked input/output parameters.

However, an assertion may also occur as the main clause, where it is not licensed for a gap. This is illustrated in Figure 1.2 by the (simplified) definition for a sentence, as having two alternatives: an assertion or a question. The definition for *assertion* itself therefore must be neutral with respect to gaps, since that depends on where it is called from (relative clause or sentence). The parameters in the assertion definition simply pass along the information from parent to child and sibling to sibling. If the assertion is in a relative clause, then the need for a gap is passed along until some node (*nullwh* in Figure 1.2) realizes the gap (that is, accepts the empty string), at which point its output parameter is set to **no_gap**; this is passed along and finally, back up to assertion. If the assertion occurs as the main clause of a sentence, it has no need for a gap and in fact, cannot unify with the gap realization rule, which requires an input parameter of **need_gap**.

This mechanism enforces the constraint that only a node with the parameter pair (**need_gap/no_gap**) can dominate a gap. Any node whose input and output parameters are equal has not 'changed state' – that is, whatever it needed (or didn't need) on rule entry, it will still need at rule exit. Procedurally, any rule whose input and output parameters are equal cannot unify with the gap realization rule. The flow of information through the tree is illustrated in Figure 1.3.

1.3 The Framework: Restriction Grammar

The proposed solution is presented in the context of Restriction Grammar [8], which is the syntactic portion of the PUNDIT text processing system [9]. However, this solution is only dependent on a few general properties of Restriction Grammar, which it shares with other formalisms (e.g., Definite Clause Translation Grammars [1]). A Restriction Grammar is written in terms of context-free BNF definitions, augmented with constraints (*restrictions*) on the well-formedness of the resulting derivation tree. Constraints operate on the derivation tree, which is constructed automatically during parsing; restrictions traverse and examine this tree, to determine well-formedness.

One of the significant characteristics of Restriction Grammar is the *absence* of parameters. Context sensitivity is enforced by the restrictions, which obtain information from the derivation (parse) tree, rather than via parameter passing.

```
% Simplified BNF definitions before parameterization for
 wh-constructions:

sentence     ::= assertion; question.
rel_clause   ::= wh, assertion.
assertion    ::= subject, verb, object.
subject      ::= noun_phrase.
verb         ::= *v.                  % * indicates terminal lexical
 category
object       ::= noun_phrase; assertion;....

noun_phrase ::= lnr; *pro.
noun_phrase ::= nullwh.
lnr          ::= ln, *n, rn.       % noun with left, right adjuncts.
rn           ::= null; pp; rel_clause.

null         ::= % .      % empty string (for empty adjunct slots)
nullwh       ::= % .      % empty string for gap realization.

wh           ::= [who]; [which]; ....

% Parameterized BNF definitions for handling relative clause:
% Where parameters pass no information, input = output parameter.

sentence(X/X)       ::= assertion(no_gap/no_gap); question
(need_gap/no_gap).
rel_clause(X/X)     ::= wh(Y/Y), assertion(need_gap/no_gap).
assertion(In/Out)   ::= subject(In/Subj), verb(Subj/Verb), object
(Verb/Out).
subject(In/Out)     ::= noun_phrase(In/Out).
verb(In/In)         ::= *v.
object(In/Out)      ::= noun_phrase(In/Out); assertion(In/Out); ...

noun_phrase(In/In) ::= lnr(In/In); *pro.
noun_phrase(need_gap/no_gap)
                    ::= *nullwh.                    % empty string
                                                    %      for gap
lnr(In/In)          ::= ln(In/In), *n, rn(In/In).   % noun + left,
                                                    % right adjuncts
rn(In/In)           ::= null(In/In); pp(In/In);
                          relative_clause(In/In).

null(In/In)         ::= % . % empty string (for empty adjunct slots)
nullwh(need_gap/no_gap)
                    ::= % . % empty string for gap realization.
wh(In/In) ::= [who]; [which]; ....
```

Figure 1.2: Simplified Rules with Parameters for Wh.

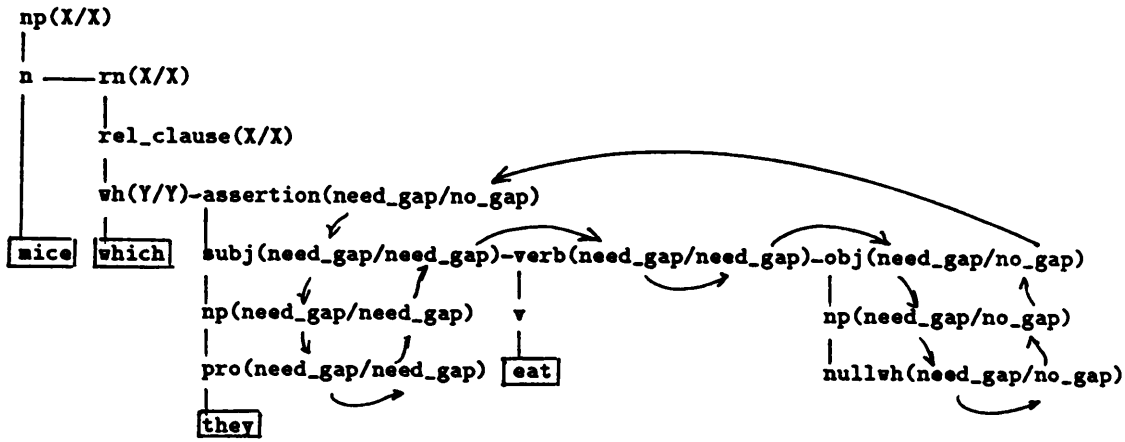

Figure 1.3: Flow of Information in *...mice which they eat.*

Restriction Grammar is implemented as a form of logic grammar which includes parameters not just for the word stream, as in DCG's, but also for the automatic construction of the derivation tree as well. In addition, each grammar rule is augmented with an associated regularization rule (indicated by a right hand arrow $\rightarrow$), which incrementally constructs an *Intermediate Syntactic Representation (ISR).* The ISR is an operator/operand notation that represents a canonical, regularized form of the parse tree. The regularization rule composes the ISRs of the daughter nodes in the derivation tree into the ISR of the current node, using lambda reduction. Computation of the ISR for wh-constructions is discussed in greater detail in section 6.

1.4 The Solution: Meta-Rules

Although parameterization is an elegant and efficient solution, it presents a major problem – it obscures the declarative aspect of the BNF rules, and correct parameterization of rules can be tedious and error prone, especially since there are some 40 object types in our current broad-coverage grammar of English.

The solution is to define a set of annotations to express the required linguistic constraints: gap introduction via wh-word, gap realization, and gap prohibition. Figure 1.4 shows a grammar using annotations defined as prefix operators applied to the node names in BNF definitions. Gap introduction is written as <<, gap realization as >>, and gap prohibition as <>. These are used, for example, to flag the need for a wh-word in a wh-expression, followed by the need to realize a

gap:

```
rel_clause ::= <<wh, >>assertion.
```

Annotations can appear on either the left-hand side or the right-hand side of BNF definitions. By introducing the gap-requirement on the right-hand side of a BNF definition, we create a conditional gap requirement. For example, *assertion* requires a gap in the context of a relative clause, but not as the normal realization of a sentence (main clause) option. Thus we do not want to annotate the definition for *assertion*, but the call to *assertion* in *rel_clause*. However, the definition for *nullwh* is always a gap realization rule, hence it is annotated on the left-hand side (see Figure 1.4).

In certain cases, we need to define a special gap-requirement rule. For example, we define a special case for noun-phrase gap realization. This enables us to *block* transmission of gap parameters in all other options of noun phrase. To do this, we use the third annotation <> to set input parameter equal to output parameters. This annotation is also used to show that the verb can never license a gap. Similarly, the determiner (*det*) and pre-nominal adjective (*adjs*) rules cannot license a gap.

The remainder of the rules require no annotation; their parameters simply transmit whatever gap information is passed in. Figure 1.5 shows the parameterized definitions corresponding to the annotated definitions used in Figure 1.4.

1.5 The Meta-Rule Component

The meta-rule component for parameterization is implemented as a general procedure which adds parameters to each production in the grammar. At grammar read-in time, each rule is parsed and parameterized appropriately, depending on its annotation. The basic case is no annotation, in which case the following rules apply (*Label* is the left-hand side of the BNF definition; *Rule* is the right-hand side):

```
% Basic case:
wh_params(Label,Rule,NewLabel,NewRule) :-
    check_head_params(Label,InParam/OutParam,NewLabel),
    take_apart(Rule,NewRule,InParam,OutParam),!.

check_head_params(Label,Params,NewHead) :-
        insert_param(Head,Params,NewHead).
insert_param(Head,Params,NewHead) :-
        NewHead =..[Head,Params].

% Conjunction
take_apart((A,B),(NewA,NewB),InParam,OutParam) :- !,
```

```
% ANNOTATED SOURCE RULES

% Operator definitions
:- op(500,fx,[<<,>>,<>]).

% Relative clause requires  gap in assertion.
rel_clause  ::= <<wh, >>assertion.

assertion   ::= subject, verb, object.
subject     ::= noun_phrase.
<>verb      ::= *v.          % verb can't have gap.
object      ::= noun_phrase; assertion..

% Regular noun phrase rule.
% Parameters block gap
<>noun_phrase ::= lnr.
% Gap realization rule
>>noun_phrase ::= nullwh.

lnr         ::= ln, *n, rn.  % lnr =
                             % left-adjunct + noun + right-adjunct

% Left noun adjunct rules
ln          ::=  det, adjs.
<>det       ::= *t; null.    % t = determiner
<>adjs      ::= null; *adj, adjs.

% Right noun adjunct rules
rn          ::= null; pp; rel_clause.
pp          ::= *prep, noun_phrase.

% Normal empty string rule
null        ::= % .
% Gap realization rule
>>nullwh    ::= %.

% Wh-word Rules
<<wh        ::= [which]; [who].
```

Figure 1.4: Illustration of Annotated Rules for Wh

```
% PARAMETERIZED VERSION OF ANNOTATED RULES

% Operator definitions
:- op(500,fx,[<<,>>,<>]).

% rel_clause        ::= <<wh, >>assertion.
rel_clause(In/In)   ::= wh(need_wh/no_wh),
                        assertion(need_gap/no_gap).

% Gap propagation rules, generated via Meta-Rule
assertion(In/Out)   ::=
        subject(In/Subj), verb(Subj/Verb), object(Verb/Out).
subject(In/Out)     ::= noun_phrase(In/Out).
verb(In/In)         ::= *v.                          % <>verb ::= *v.
object(In/Out)      ::= noun_phrase(In/Out);
                        assertion(In/Out).

%  Regular noun phrase rule.
%  Annotation blocks gap: <>noun_phrase ::= lnr.
noun_phrase(In/In) ::=          lnr(In/In).
% Gap realization rule: >>noun_phrase ::= nullwh.
noun_phrase(need_gap/no_gap)
                    ::= nullwh(need_gap/no_gap). % the empty string
lnr(In/Out)         ::= ln(In,Out1), *n, rn(Out1/Out).
% Left noun adjunct rules
ln(In/Out)          ::=  det(In/Out1), adjs(Out1/Out).
det(In/In)          ::= *t; null(In/In).    % <> det ::= *t; null.
adjs(In/In)         ::= null(In/In); *adj, adjs(In/In).
                                            % <>adjs ::= null; *adj,
                                            %                 adjs.
% Right noun adjunct rules
rn(In/Out)          ::= null(In/Out); pp(In/Out);
                        rel_clause(In/Out).
pp(In/Out)          ::= *prep, noun_phrase(In/Out).
% Normal empty string rule
null(In/In)         ::= % .
% Gap realization rule
nullwh(need_gap/no_gap) ::= %.
% Wh-word Rules
wh(need_wh/no_wh)   ::= [which];[who].  % <<wh ::= [which];[who].
```

Figure 1.5: Illustration of Parameterized Rules for Wh

```
    take_apart(A,NewA,InParam,OutParamA),
    take_apart(B,NewB,OutParamA,OutParam).

% Non-Terminal
take_apart(Def,NewDef,InParam,OutParam) :-
    insert_param(Def,InParam/OutParam,NewDef),!.
```

If there is a terminal symbol, indicated by *, there is clearly no gap, and input and output parameters are equal; terminal symbols do not get parameterized:

```
% Terminal
take_apart(*Atom,*Atom,InParam,InParam) :- !.
```

Parameterization of embedded disjunction poses a problem, because there is a possibility that different disjuncts could instantiate the *In/Out* parameters differently. To catch this problem, each disjunct is computed separately, and a routine *same_params* checks the results, to make sure that they are consistent, before instantiating the parameters. If they are inconsistent, it issues a warning message. Otherwise, it unifies the inputs of the disjunctions; likewise, it unifies the outputs. It is always possible to avoid the warning message by splitting embedded disjunctions into separate rules, as was done for the noun_phrase definition (see Figure 1.4).

```
% Disjunction
take_apart((A;B),(NewA;NewB),InParam,OutParam) :-
    take_apart(A,NewA,InParamA,OutParamA),
    take_apart(B,NewB,InParamB,OutParamB),
    same_params((A;B),
        InParamA,InParamB,InParam,OutParamA,OutParamB,OutParam),!.

take_apart((A;B),_,_,_) :- !,
    print('$$$ Warning:  disjunct '),
    print(A), print(';'), print(B),
    print(' may not be parameterized correctly!').
```

There is also a special case for each annotation, for both the left-hand side of the rule (*check_head_params/3*) and the right-hand side (*take_apart/4*). For the case of the wh-word <<), the rule is shown below. In this case, the wh-word functions independently of other nodes in the definition and so it does not hook up to the input/output parameters.[2]

[2]In actuality, the wh-expression needs to pass some information about the wh-word to the expression containing the gap, and this rule will be revised in section 6.

```
% << Needs wh-word
check_head_params(<<Head,(need_wh/no_wh),NewHead) :- !,
    insert_param(Head,(need_wh/no_wh),NewHead).

take_apart(<<Def,NewDef,InParam,InParam) :- !,
    insert_param(Def,(need_wh/no_wh),NewDef).
```

The gap-requirement annotation >> also disconnects the phrase containing the gap from its parent and siblings.

```
% >> Requires gap
check_head_params(>>Head,(need_gap/no_gap),NewHead) :- !,
    insert_param(Head,(need_gap/no_gap),NewHead).

take_apart(>>Def,NewDef,InParam,InParam) :-!,
    insert_param(Def,(need_gap/no_gap),NewDef).
```

Finally, the annotation <> enforces sameness of input and output, precluding realization of a gap:

```
% <> Rules out gap
check_head_params(<>Head,In/In,NewHead) :- !,
    insert_param(Head,In/In,NewHead).

take_apart(<>Def,NewDef,InParam,InParam) :- !,
    insert_param(Def,InParam/InParam,NewDef).
```

This small set of annotations is sufficient to describe the wh-constructions in English with one minor addition, needed to handle conjunction correctly (see section 7). Using this small set of annotations, the grammar writer can control the flow of gap information, without having a grammar cluttered with parameters. Unification is used to control generation of gaps only where required, so the technique is also efficient, avoiding extensive search to determine presence/absence of a gap.

1.6 Refinements

The treatment described above leaves open several issues. The first of these is the proper generation of a regularized syntax (*Intermediate Syntactic Representation*) for these constructions. It is clear that the compositional representation of a gap-containing expression can readily be described as a lambda expression. First, the wh-expression itself can be viewed as a lambda expression, which produces a *filler* when applied to the referent of the relative pronoun. Then, the gap-containing

expression is a lambda expression, which when applied to the filler, produces a completed clause:

the book which I bought ( ):

```
=>     book1, lambda(Filler, [I bought Filler]),
                                        [lambda(Wh,[Wh]),book1]
=>     book1, lambda(Filler, [I bought Filler]), [book1]
=>     book1, [I bought book1].
```

Only wh-words in relative clauses have antecedents, namely the head noun to which the relative clause is attached. Wh-words in questions stand for the questioned element in the clause; in this case, we insert a dummy element **wh_gap** to mark the questioned element.

what did you buy ( )?

```
=>    lambda(Filler,[you buy Filler]), [lambda(Wh, [Wh]), wh\_gap]
=>    lambda(Filler,[you buy Filler]), [wh_gap]
=>    [you buy wh_gap]
```

This treatment extends very nicely to complex wh-expressions, such as *the person whose book I borrowed*, which can be represented as follows:

the person whose book I borrowed ( )

```
=>    person1,
       lambda(Filler,[I borrow Filler]), [lambda(Wh, [Wh 's book]),
 person1]
=>    person1, lambda(Filler, [I borrow Filler]), [person1 's book]
=>    person1, [I borrow person1's book]
```

This treatment has one unfortunate property: at the time that the lambda variable is generated for the relative clause, there is no way of knowing where it should be placed in the lambda expression, that is, where the associated gap in the assertion will be. For example, the lambda variable associated with a noun phrase gap could be realized as the subject or the object of the clause. Our solution is to pass the lambda variable along, embedded in the gap parameters, until the noun-phrase gap-realization rule rule is reached, at which point the lambda variable becomes the representation of the gap, shown in Figure 1.6.

We implement this by treating **need_gap** as a functor with an argument for the lambda variable. To avoid explicit mention of parameters in the source rules, we again use an annotation. Access to the lambda variable embedded in the

need_gap parameter is given by the annotation **//lambdaVar(Var)**, where **//** is a binary operator.

Figure 1.6 shows several wh-constructions, with their associated ISR rules. The ISR rule appears on the right hand side of the arrow '->'. The node names within the ISR rule access the ISRs associated with the named node.

There are other complications in covering wh-constructions. One issue is that several different types of gap can occur, specifically noun phrase gaps, adverbial phrase gaps, and even adjective gaps:

The book that I bought (np gap)
What did you buy? (np gap)
The place I put it (adverb gap)
Where did you get it? (adverb gap)
The way I did it (adverb gap)
How big is it? (adjective gap)

In many cases, the wh-word signals the type of gap to be expected. This information is critical to determining what element should be realized as the gap. A simple way of handling this is to add a 'gap-type' argument to the information being passed in the parameters. Thus for *noun_phrase* to realize a gap, the gap type must be *np*; for an *adverb* to realize the gap, we must have gap type *adv*. This information is computed in the handling of the wh-expression and passed along to the gap-licensing construction. Thus the wh-expression does in fact have to communicate information to the construction dominating the gap. This is implemented by a slight complication to the meta-rule. The gap type information is accessed by the annotation **//type(GapType)** on the appropriate rule.

Finally, the semantics needs to know what the wh-word word was, so that it can distinguish between location expressions (*the place where I left it*) from temporal expressions (*the month I left*). Of course, there is not always an overt wh-word to indicate gap type: *the time I spent* (np gap) vs. *the time I visited* (adverb gap). All of these complications can be handled by embedding an additional parameter to carry the actual wh-word (if present) into the basic *need_gap* expression. This information is accessed by the annotation **//wh_word(WH)** and is used in constructing the final ISR.

1.7 Wh and Conjunction

A major complication occurs in the interaction between wh-constructions and conjunction. For example, with conjunction, it is not longer true that there is exactly one gap per wh-expression. Within a conjoined construction, the conjuncts must be identical with respect to gaps – if one conjunct has a gap, the other must have one as well, as in:

```
% ANNOTATED RULES with ISR

:- op(500,fx,[<<,>>,<>,><]).
:- op(400,xfy,//).

<>rn      ::=  null -> lambda(N, [N]);
               pn -> lambda(N, [N, !pn]);
               rel_clause
                     -> lambda(N, [N, [-rel_clause, [+rel_clause,
                                                  copy(N)]]]).
% +Def extracts the head of a list; -Def get the tail of the list.
rel_clause ::= <<wh, >>assertion//lambdaVar(Gap)
                        -> [wh, lambda(Gap, [assertion])].

assertion ::=  subject, verb, object -> [verb, object, subject].
subject   ::=  noun_phrase -> noun_phrase.
<>verb    ::=  *v -> lambda(Obj, [lambda(Subj, [v, Subj, Obj])]).
object    ::=  noun_phrase -> noun_phrase; assertion -> assertion.
<>noun_phrase
          ::= lnr -> lnr.
% gapped noun_phrase -- lambda variable is ISR.
>>noun_phrase//lambdaVar(Gap)
          ::= nullwh -> Gap.
<<wh      ::= [which];[who] -> lambda(Filler, [Filler]).

% PARAMETERIZED VERSION
rn(In/In) ::= null(In/In) -> lambda(N, [N]).
              pn(In/In) -> lambda(N, [N,!pn]);
              rel_clause(In/In)
                   -> lambda(N, [N, [-rel_clause, [+rel_clause,
                                                 copy(N)]]]).
% +Def extracts the head of a list; -Def gets the tail of the list.
rel_clause(In/In)
          ::= wh(need_wh/no_wh), assertion(need_gap(Gap)/no_gap)
                   -> [wh, lambda(Gap, [assertion])].
noun_phrase(In/In)
          ::= lnr(In/In) -> lnr.
noun_phrase(need_gap(Gap)/no_gap)
          ::= nullwh(need_gap(Gap)/no_gap) -> Gap.
% omitted defs for assertion, subject, verb, object.
wh(need_wh/no_wh)
          ::= [which];[who] -> lambda(Filler, [Filler]).
```

Figure 1.6: The ISR Rules for Wh-Constructions

the books which I bought ( ) and you read ( )

or

the letter that ( ) arrived yesterday and I sent ( ) on to you.

but not

**the books which I bought and you read them*

In an interpreter-based treatment of wh-constructions, the interpreter needs to make a special provision for 'resetting' its state to handle gaps over conjoined structure – that is, it must account for the fact that there may be two gaps if the wh-construction has scope over a conjoined structure. One advantage of a meta-rule treatment is that this interaction occurs in an extremely natural way: the meta-rules for wh-expressions are applied first; then the conjunction meta-rules simply copy the parameters of the first conjunct for the second conjunct. Since these are implemented as logical variables, the parameters of the two conjuncts are unified, and hence are guaranteed to have exactly the desired behavior.

The ability to factor syntax and semantics cleanly makes it possible to implement a simple meta-rule treatment of conjunction [7]. The basic idea is to use meta-rules to generate all possible conjoinings as explicit rules. Because there are no parameters in the rules, and because the ISR rules are compositional and cleanly factored from the BNF definitions, the meta-rules are very simple. Figure 1.7 shows the transformation of a (simplified) assertion definition into a conjoined assertion definition. The resultant rule is a disjunction; one branch allows a conjunction, followed by a recursive call to assertion; the other branch terminates after application of a restriction. Each branch has associated with it a separate regularization rule. The original meta-rules, though simple, are low-level operations, concerned with maintaining and updating a recorded database of rules. Recently, a general mechanism to support the statement of meta-rules has been proposed [2] this treatment would provide a more elegant statement of the meta-rules used to handle conjunction in Restriction Grammar.

The current conjunction meta-rules will handle correctly the interaction of wh-expressions and conjunction. However, one minor problem is the preservation of a source form of the rules, for inspection and editing by the grammar-writer. This requires that conjunction operate on the source (unparameterized) form of the rules. By introduction of one addition annotation $><$, it is possible to apply conjunction to the source rule, which can then be parameterized via the wh meta-rule component. The $><$ annotation simply ensures that its operand receives the parameters associated with the left-hand side of the definition. Thus the conjunction meta-rule generates:

```
% Conjunction meta-rule (using '=>' as infix operator):
    (LHS ::= Body -> ISR) =>
    (LHS ::= Body,
        (*conj_wd, LHS  -> [conj_wd, ISR, LHS]
        ; {wconj_restr} -> ISR)).

% Example of meta-rule applied to assertion definition:

assertion ::= subject, verb, object -> [verb, subject, object].
        =>
assertion ::= subject, verb, object,
               (*conj_wd, assertion
                   -> [conj_wd, [verb, subject, object],
                                                assertion]
               ; {w_conj_restr} -> [verb, subject, object]).
```

Figure 1.7: Generation of Conjunction via Meta-Rule

```
assertion ::= subject, verb, object,
               ( *conj_wd, ><assertion
                   -> [conj_wd, [verb, subject, object],
                                                assertion]
                 {w_conj_restr} -> [verb, subject, object]).
```

The introduction of the >< annotation requires a slight complication to the code for **take_apart**, but is very straight-forward. The result is the following parameterized definition for assertion:

```
assertion(In/Out) ::=
     subject(In/Subj), verb(Subj/Verb), object(Verb/Out),
        (*conj_wd, assertion(In/Out)
              -> [conj_wd, [verb, subject, object], assertion]
        ; {w_conj_restr} -> [verb, subject, object]).
```

Combining treatment of conjunction and wh-constructions, the system is able to parse sentences such as:

The disk which he repaired and she installed is working.

The disk which she repaired and installed has failed.
What disk did she repair and he install?
The disk which was installed but not repaired has been removed.
Which disks and drives are failing?
What does she believe they will repair and the engineer will maintain?

1.8 Conclusion

It is clear that meta-rules provide a powerful approach to a range of grammatical problems. In particular, meta-rules are very well-suited to the handling of phenomena that require a regular change to a large range of grammatical constructions. Although the parameterization approach outlined here for wh-constructions differs considerably from the meta-rule treatment of conjunction, they both share the property of enabling the grammar writer to capture a high-level generalization that applies to many rules in the grammar. Perhaps even more interesting is the fact that these meta-rule treatments appear to combine gracefully, in a way that does not appear to be readily available through the extended interpreter approach of Extraposition Grammar.

We plan to investigate other possible applications of parameterized grammar rules. These include the use of parameters to propagate feature information, and the use of parameters to 'compile' restrictions into unification of parameters, for greater efficiency and greater *locality* within subtrees. By using parameters instead of explicit constraints on tree-shape, it may be possible to save well-formed subtrees. The problem of putting together well-formed subtrees can then be captured via unification, rather than by constraints which must look outside the local subtree.

As we continue to develop meta-rule approaches to various grammatical phenomena, it will be important to abstract from the specifics outlined here and move towards a more general language for the statement of grammatical meta-rules. Just as grammar examples provide fertile ground for the application of meta-programming techniques, we expect meta-programming techniques to provide more elegant and efficient ways of capturing the meta-rules.

1.9 Acknowledgements

A number of people have made important contributions to this work. The basic grammatical insights come from Marcia Linebarger, the principal developer of our broad-coverage English grammar. The implementation of the ISR has been done by John Dowding, who also suggested the appropriate treatment of gaps and of complex wh-expressions. Deborah Dahl, Rebecca Passonneau and François Lang provided a number of helpful suggestions on the organization of the paper. I am

also indebted to Dale Miller for interesting insights about the meta-programming aspects of the wh-problem.

References

[1] Harvey Abramson. Definite clause translation grammar. In *Proc. 1984 International Symposium on Logic Programming*, pages 233–241, Atlantic City, NJ, 1984.

[2] Harvey Abramson. Metarules and an approach to conjunction in definite clause translation grammars: some aspects of grammatical metaprogramming. In *Logic Programming: Proc. of the 5th International Conf. and Symposium*, pages 233–248, MIT Press, Cambridge, MA, 1988.

[3] Amelie Banks and Manny Rayner. Comparatives in logic grammars – two viewpoints. In *Proc. of the 2nd International Workshop on Natural Language Understanding*, Simon Fraser University, Vancouver, B.C, August, 1987.

[4] Veronica Dahl and Michael McCord. Treating co-ordination in logic grammars. *American Journal of Computational Linguistics*, 9(2):69–91, 1983.

[5] John Dowding and Lynette Hirschman. Dynamic translation for rule pruning in restriction grammar. In *Proceedings of the 2nd International Workshop On Natural Language Understanding and Logic Programming*, Vancouver, B.C., Canada, 1987.

[6] G. Gazdar. Unbounded dependencies and co-ordinate structure. *Linguistic Inquiry*, 12:155–184, 1981.

[7] Lynette Hirschman. Conjunction in meta-restriction grammar. *Journal of Logic Programming*, 4:299–328, 1986.

[8] Lynette Hirschman and Karl Puder. Restriction grammar: a prolog implementation. In D.H.D. Warren and M. VanCaneghem, editors, *Logic Programming and its Applications*, pages 244–261, Ablex Publishing Corp., Norwood, N.J., 1986.

[9] Martha S. Palmer, Deborah A. Dahl, Rebecca J. [Schiffman] Passonneau, Lynette Hirschman, Marcia Linebarger, and John Dowding. Recovering implicit information. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, August 1986.

[10] F.C.N. Pereira. Extraposition grammars. *American Journal of Computational Linguistics*, 7:243–256, 1981.

[11] Fernando Pereira and Stuart Shieber. *Prolog and Natural-Language Analysis.* University of Chicago Press, Chicago, Illinois, 1987.

[12] C. Raze. A computational treatment of coordinate conjunctions. *American Journal of Computational Linguistics*, 1976. Microfiche 52.

[13] Naomi Sager. Syntactic analysis of natural language. In *Advances in Computers*, pages 153–188, Academic Press, New York, NY, 1967.

[14] William A. Woods. Progress in natural language understanding: an application to lunar geology. In *AFIPS Conference Proceedings*, 1973.

Appendix

Meta-Rules for Parameterization of BNF Definitions

```
% Do parameter generation at rule read-in time,
% to capture meta-rules.
% Pick up rule, add parameters, and record new rule.
Label ::= Rule :-
  wh_params(Label,Rule,ParamLabel,ParamRule),
  record(ParamLabel,(ParamLabel ::= ParamRule),_).

% Generate wh-parameters
wh_params(Label,Rule,NewLabel,NewRule) :-
  make_head_params(Label,InParam/OutParam,NewLabel),
  take_apart(Rule,NewRule,InParam,OutParam).

% Parameterize head of rule
%       Case 1 -- head cannot dominate gap; close off parameters
make_head_params(Head,InParam/InParam, NewHead) :-
  cant_pass_on(Head),!,
  insert_param(Head,InParam/InParam,NewHead).
%       Case 2 -- parameterize head.
make_head_params(Head,InParam/OutParam, NewHead) :-
  insert_param(Head,InParam/OutParam,NewHead).

% TAKE APART RULE BODY to insert params
% Semantics Rule
take_apart((A -> B), (NewA -> B), InParam,OutParam) :- !,
  take_apart(A,NewA,InParam,OutParam).
% Conjunction
take_apart((A,B),(NewA,NewB),InParam,OutParam) :- !,
```

```
  take_apart(A,NewA,InParam,OutParamA),
  take_apart(B,NewB,OutParamA,OutParam).
% Disjunction
take_apart((A;B),(NewA;NewB),InParam,OutParam) :-
  take_apart(A,NewA,InParamA,OutParamA),
  take_apart(B,NewB,InParamB,OutParamB),
  same_params(
(A;B),InParamA,InParamB,InParam,OutParamA,OutParamB,OutParam),!.
take_apart((A;B),_,_,_) :- !,
  print('$$$ Error:  disjunct '), print(A), print(';'), print(B),
  print(' cannot be parameterized correctly!').
% Literal
take_apart([Word|MoreWords],[Word|MoreWords],InParam,InParam) :- !.
% Restriction
take_apart({Restr},{Restr},InParam,InParam) :- !.
% Prune
take_apart(prune(Name,Defs),prune(Name,NewDefs),InParam,
                                                    OutParam) :- !,
  take_apart(Defs,NewDefs,InParam,OutParam).
% Literal
take_apart(*Atom,*Atom,InParam,InParam) :- !.
% Empty node
take_apart(% , % , InParam,InParam) :- !.
% Non-Terminal
take_apart(Def,NewDef,InParam,OutParam) :-
  check_params(Def,InParam,OutParam),
  insert_param(Def,InParam/OutParam,NewDef),!.
take_apart(Def,Def,InParam,InParam).

% cant_pass_on sets two parameters equal in the head of the
% name rule;
%       this means that the automatically generated
%       nodes of this type
%       CANNOT contain a gap or a wh word.  These nodes have
 hand-generated
%       parameterized rules that permit gap or wh-words.
cant_pass_on(sa).
cant_pass_on(rn).
cant_pass_on(nstg).
cant_pass_on(thats).

insert_param(Head,X/Y,NewHead) :-
  Head =.. [Head|Args],
  NewHead =..[Head,X/Y|Args].
```