A computational approach to
the syntax of displacement and the semantics of scope

Published by LOT

Janskerkhof 13
3512 BL Utrecht
The Netherlands

phone: +31 30 253 6006
fax: +31 30 253 6406
e-mail: lot@let.uu.nl
http://www.lotschool.nl

Cover illustration: © 2007 Zach VandeZande
(http://www.animalshaveproblemstoo.com)

ISBN: 978-94-6093-021-8 NUR 616

A computational approach to the syntax of displacement and the semantics of scope

Een computationele benadering
van de syntaxis van beweging
en de semantiek van bereik
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. J. C. Stoof,
involge het besluit van het college voor promoties
in het openbaar te verdedigen
op woensdag 31 maart 2010
des ochtends te 10.30 uur

door

Andrea Christina Unger
geboren op 5 mei 1982
te Leipzig, Duitsland

Promotoren: Prof. dr. D. J. N. van Eijck
Prof. dr. E. J. Reuland

Contents

List of Figures

Acknowledgements

Science is made by friends.
(Haj Ross)

I have been among many marvellous people, without whom this thesis would not be. And although I would like to thank some of them more than I should, I am afraid I will not thank half of them half as well as I should like and certainly not all of them more than half as well as they deserve. Nevertheless I wish to express my gratitude to all of them.

Rightfully first in this list are my supervisors. Jan, for pushing and pulling me at all possible times in all necessary ways and far beyond this thesis. And Eric, for being the L in UiL OTS.

I am also deeply indebted to Gereon Müller, who had been an invaluable source for everything syntax-related, and without whom I might have neither started nor finished this dissertation.

Among the people that inspired me are Klaus Abels, Ulf Brosziewski, Alexis Dimitriadis, Philippe de Groote, Greg Kobele, Andres Löh, Michael Moortgat, Rick Nouwen, Eddy Ruys, Chung-chieh Shan, and Craig Thiersch.

Among my fellow PhDs there are two that I want to thank in particular. Gianluca Giorgolo, for an innumerable amount of things, especially for his computer science influence and a lot of rock'n'roll. Andreas Pankau, for sharing the cultural background and an apartment, and for providing me with beer and football. And both of them for sharing and discussing ideas, for their friendship, and for all the fun.

And I am no less grateful to all other colleagues and friends that made life inside and outside the UiL OTS smashing: Min Que, Sander van der Harst, Bert Le Bruyn, Anna Volkova, Roberta Tedeschi, Gaetano Fiorin, Clizia Welker, Bettina Gruber, Xiaoli Dong, Berit Gehrke, Nino Grillo, Giorgos Spathas, Jakub Dotlačil, Marieke Schouwstra, Anna Chernilovskaya, Lizet van Ewijk,

Ana Aguilar Guevara, Nadya Goldberg, Matteo Capelletti, Linda Badan, Marijke de Belder, Dagmar Schadler, Diana Apoussidou, Natalie Boll-Avetisyan, Frans Adriaans, Arjen Zondervan, Paolo Turrini, Radek Šimík, Arno Bastenhof and Jeroen Goudsmit, as well as the Tilburg Chicks. They all made a huge difference. As did those whom I don't know how to thank.

Moreover, very visible contributions have to be attributed to Andreas, who pointed me to Brosziewski's work, Min, Xiaoli and Mana, wo helped me with the data, and Marieke, who translated the samenvatting.

And although not directly connected to this thesis, I wish to thank Philipp Cimiano for offering the time and freedom I needed to finish up, and for providing an environment that made it more than worth to move on.

Finally, a very special thanks goes to my parents and my brother for being my parents and my brother.

1

Introduction

The grammatical knowledge we have allows us to effectively link spoken language with meaning, both when we perceive utterances and need to understand them, and when we want to convey a meaning and need to choose the sounds we have to articulate. The aim of theoretical linguistics is to model this grammatical knowledge. Such a model usually comprises several recursive procedures, among them one for combining words into phrases and sentences (*syntax*) and one for constructing meanings (*semantics*). Since syntactic units and their meanings are related in a very systematic way, syntax and semantics are taken to be tightly connected.

The thesis at hand is about two particular phenomena at the interface between syntactic structure and meaning: wh-displacement and operator scope. Together they embody as well as challenge the tight connection between syntax and semantics that is commonly assumed. On the one hand, a lot of languages syntactically displace operator expressions exactly to the position where they semantically take scope. This led to many theories assuming displacement and scope to be two sides of the same coin. But on the other hand, displacement and scope do not coincide in general and across all languages. In fact, in quite a lot of cases operator expressions are neither displaced nor does their syntactic position correspond to their semantic scope position. This is why I want to explore an alternative way of looking at displacement and scope. Instead of considering them to be tightly linked, I want to argue that they are not connected at all, and that the mismatches are actually the normal case. This goes hand in hand with an alternative view on the syntax/semantics interface,

in which syntactic and semantic procedures can operate independently of each other.

In particular, I propose that our grammatical knowledge consists of two subsystems. The first one is a core system for combining simple expressions into more complex expressions, with syntax and semantics working completely in parallel. The second one comprises extensions to the core system consisting of syntactic and semantic procedures that operate independently of each other. I propose that it is those extensions that are responsible for non-local dependencies such as displacement and scope construal. More specifically, I propose that displacement is derived by a syntactic procedure that receives no semantic interpretation, and that operator scope is established by a semantic procedure that has no syntactic counterpart.

I will proceed by first developing the core system and a basic link between syntax and semantics, and then extending the core system with independent procedures for displacement and scope construal. This partial decoupling of syntax and semantics will provide a straightforward way to explain mismatches between form (in particular syntactic surface positions) and meaning (in particular semantic scope positions).

Additionally, my approach will be computational, as the title of the thesis suggests. And it will be so in two respects. First, the adopted semantic procedure employs concepts from computer science for establishing operator scope, mainly evaluation contexts and delimited control. And second, although I follow most theoretical linguists in studying language as a formal rule system, I go further by also implementing this rule system. Formal linguistics in this thesis is thus not a pen and paper enterprise but uses the computational tools at hand. I do not only want to specify a recursive algorithm that can systematically generate phrases and sentences along with their meanings, but I want to also be able to execute this algorithm and actually compute form-meaning pairs. So instead of only devising an algorithm that is formal enough to be implemented in a machine, I want to provide such an implementation. This can then be used to go one step further than formal definitions: test them for empirical adequacy and predictive power. In particular, the implementation will enable us to easily test cases that get too complex for keeping track of all details by hand. And it furthermore proves useful to sharpen the theory and ensure it to be consistent and work the way it was intended to work.

The language of choice for the implementation is the functional programming language Haskell. Functional programming is suitable for the task of linguistic computation because it allows to program at a very abstract level and furthermore keeps the step from definition to implementation very small. In fact, the main part of the implementation follows the formal definitions almost to the letter. The implementation thus can indeed serve to check the correctness of the definitions. The linguistically minded reader can nevertheless safely skip the implementation and simply feel assured that the linguistic rules I employ indeed compute the grammatical phrases and sentences of the

fragment I will focus on.

The plot of the book is the following. Chapter 2 introduces the dependencies this thesis is about: wh-displacement and operator scope. It investigates their characteristic properties and concludes that they are not two sides of the same coin but rather constitute two distinct mechanisms. After carving out the core system in Chapter 3, Chapter 4 models a syntactic procedure that can tell the story of displacement, and Chapter 5 models a semantic procedure that can tell the story of operator scope. In Chapter 6, I give an implementation of the suggested algorithm and briefly explain how it can be used. Finally, in Chapter 7, I summarize and investigate the implications that the proposed view has on the general modeling of the syntax/semantics interface.

2

Contrasting displacement and scope

This chapter introduces the two phenomena under consideration: wh-displacement and operator scope. We will start by looking at their behavior and characteristic properties. Of special importance will be restrictions on wh-displacement and scope and in howfar these restrictions can be considered to be related. Then I will review the reasons why it is commonly assumed that both phenomena are two sides of the same coin and look at reasons to reject this parallelism. The chapter ends with an overview of the thesis.

To get a taste of the phenomena under investigation, consider the following wh-question.

(2.1) Whom did the gods know that every citizen of Uruk feared?

On the one hand, there is a structural dependency between the clause-initial position of the wh-expression and the gapped position where it presumably originates from. Throughout the book, I will designate the gap as __ and mark the dependency by indices on the involved elements, as in (2.2a). And on the other hand, there is an interpretative dependency between these two positions: the front position semantically corresponds to an operator that binds a variable in the argument position marked by the gap, as indicated in (2.2b).

(2.2) a. Whom$_1$ did the gods know that every citizen of Uruk feared __$_1$?

b. Which x is such that the gods knew that every citizen of Uruk feared x?

Both dependencies are *unbounded*: in principle arbitrarily many clause boundaries can intervene.

(2.3) a. Whom$_1$ did you say that Anu thought that the gods know that every citizen of Uruk feared __$_1$?

b. Which x is such that you said that Anu thought that the gods knew that every citizen of Uruk feared x?

I follow Gazdar [42] in conceiving unbounded dependencies as consisting of three parts: *top* is the position where the dependency is introduced (in our example the wh-expression whom and the operator 'which x'), *middle* is the substructure that the dependency spans, and *bottom* is the position where the dependency ends (in our examples the gap or the variable). The top is also called *head* of the dependency and the bottom is also called *foot* of the dependency.

The next two sections are dedicated to looking at the two unbounded dependencies under consideration, displacement and operator scope, in more detail. Although they display a parallel structure in example (2.2) above, they turn out to have different characteristics, as we will see in Section 2.2. I will then, in Section 2.5, propose to take the mismatches as indication that displacement and operator scope are in fact different dependencies that should be treated separately.

2.1 Displacement

Consider again our first example:

(2.4) Whom$_1$ did the gods know that every citizen of Uruk feared __$_1$?

What tells us that the wh-phrase is indeed displaced, i.e. that there is a structural dependency between the wh-phrase and the gap position? An obvious observation is that the question asks for the object of the verb fear, which in echo questions and declarative sentences appears in the position indicated by the gap, as shown in (2.5).

(2.5) a. The gods knew that every citizen of Uruk feared whom?

b. The gods knew that every citizen of Uruk feared Gilgamesh.

Two facts further suggest a relation between the object position of the embedded verb fear and the fronted wh-phrase in (2.4). First, the fronted wh-phrase shows agreement with the verb. Due to the lack of overt agreement morphology, English is not a good language to observe this, but we can see at least that in (2.4), the wh-expression whom seems to get its case from the embedded verb fear, just like in (2.5a). Now, virtually every syntactic framework assumes that case assignment is a local dependency between a verb and its arguments; especially, case cannot be assigned across clause boundaries. Therefore, we

need to establish a local relationship between the fronted wh-phrase and the embedded verb somehow.

Second, we know that pronouns can only be bound in certain structural configurations. The configuration that is commonly accepted to be most relevant is c-command. We will look at it later; here, as a starting point, we simply assume that linear precedence is important for binding. To illustrate this consider (2.6), where we mark the interpretative dependency of pronominal binding by indices. In (2.6a) the antecedent precedes the pronoun, binding is therefore possible, whereas in (2.6b) the antecedent does not precede the pronoun, binding is therefore not possible.

(2.6) a. [Every king]$_1$ tyrannized his$_1$ citizens.
b. * His$_1$ king tyrannized [every citizen]$_1$.

Now, when the pronoun his is contained in a displaced wh-phrase, as in (2.7), it is not preceded by its antecedent everyone (neither is it c-commanded, for that matter) and binding should therefore be impossible. Nevertheless, binding is possible.

(2.7) [Which god of his$_1$ ancestors]$_2$ did everyone$_1$ worship __$_2$?

Why is that? Important is that the pronoun could be bound if the wh-phrase resided in the gap position. So, again, we have to assume that the wh-phrase is related to the gap position somehow.

In order to keep things general for now, we simply conclude that there is a dependency of some sort between a fronted wh-phrase and the corresponding gap. Let us now look at what form this dependency takes in other languages. There are mainly three strategies for the formation of wh-questions across languages, differing in where wh-phrases occur. The first one is *wh-in-situ*, which is employed for example in Korean (an SOV language), Chinese, and Japanese. In these languages, wh-phrases always appear at the bottom of the dependency.

(2.8) *Korean* (Beck & Kim [8])
Suna-ka muôs-ûl ilk-ôss-ni?
Suna-NOM what-ACC read-PST-Q
'What did Suna read?'

(2.9) *Mandarin Chinese* (Watanabe [126])
Ni xiang-zhidao wo weishenme gei Akiu shenme?
you wonder I why give Akiu what
'What do you wonder why I give Akiu what?'

(2.10) *Japanese*[1]
Akira-no Hikaru-ga dare-ni nani-o ageta-to omotte-imasu-ka?
Akira-TOP Hikaru-NOM who-DAT what-ACC gave-COMP think-be-Q
'Whom does Akira think that Hikaru gave what?'

[1] All Japanese examples without a reference are based on judgments by Mana Kobuchi.

The second strategy is *simple wh-movement*, employed for example by English and Dutch. Exactly one wh-phrase is fronted and all others stay in situ.

(2.11) Whom$_1$ did Gilgamesh tyrannize __$_1$ how?

(2.12) *Dutch*

Wie$_1$ heeft de jager __$_1$ waar ontdekt?
whom has the hunter where discovered
'Whom did the hunter discover where?'

The third strategy is *multiple wh-movement*, found in many Slavic languages such as Bulgarian and Serbo-Croatian. All wh-phrases have to be fronted.

(2.13) *Bulgarian* (Billings & Rudin [11])

Koj$_1$ kakvo$_2$ [na kogo]$_3$ __$_1$ kaza __$_2$ __$_3$?
who.NOM what.ACC to who.DAT say.PST
'Who told what to whom?'

(2.14) *Serbo-Croatian* (Bošković [123])

Ko$_1$ koga$_2$ __$_1$ voli __$_2$?
who who.ACC loves
'Who loves whom?'

Additional to where a wh-phrase occur, there is a strategy for indicating the scope of the corresponding wh-operator, referred to as *wh-scope marking*. While the actual wh-expression stays in situ or is displaced only within one clause, its scope is explicitly indicated by a scope marker in a higher position. The scope marker usually takes the form of the language's wh-word corresponding to what. This is illustrated by the German example (2.15). Another language with scope marking, also over long distances, is Hindi, see (2.16). In some cases the scope marker even seems to be phonologically empty, as in the Malay example (2.17). This is also referred to as *partial wh-movement*.

(2.15) *German*

Was glaubst du [wen$_1$ Shamhat verführen soll __$_1$]?
what believe you who.ACC Shamhat seduce shall
'Whom do you believe that Shamhat shall seduce?'

(2.16) *Hindi* (Mahajan [72])

Raam-ne kyaa socaa [ki ravii-ne kyaa kahaa [ki kon sa aadmii
Ram-ERG what thought that Ravi-ERG what said that which man
aayaa thaa]]?
came be.PST
'Which man did Ram think that Ravi said came?'

(2.17) *Malay* (Cole & Hermon [25])

Kamu fikir [[ke mana]$_1$ Fatimah pergi __$_1$]?
you think to where Fatimah go
'Where do you think that Fatimah went?'

Scope marking overlaps with the three other strategies we saw. Korean and Japanese, for example, are wh-in-situ languages but require an obligatory question particle that marks the scope of the corresponding wh-operator. We will look at the role of scope marking in Chapter 5.

In general, we assume that if a language fronts a wh-phrase, this fronting is obligatory. There are some languages that are considered to have optional fronting (for example Bahasa Indonesia, Egyptian Arabic and Palauan), however, Cheng [16] provided evidence for assuming that this fronting is, in fact, an instance of clefting and that those languages are best classified wh-in-situ languages.

2.2 Restrictions on displacement

Displacement is a dependency between two syntactic positions. It is subject to the structural condition of c-command and the following two kinds of locality:

- *Rigid locality*
  An expression may not be extracted from an island.
- *Relativized locality*
  An expression can be displaced only when no element intervenes that also has the relevant properties.

Let us examine those conditions in turn, after a short note on terminology: In the following, I will use the term 'displacement', or 'extraction', when refering to the structural dependency between an expression and a gap, and about 'movement' when talking about the operation that establishes this dependency. The term movement will not bear any theoretical commitment, though.

2.2.1 C-command

The most fundamental structural condition on displacement dependencies is the following:

> A displaced expression must c-command the corresponding gap.

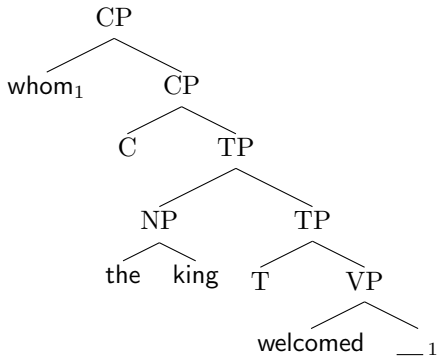
C-command is a representational notion. According to the standard definition due to Reinhart [88], it is defined as a relation between nodes in a tree.

> A node x *c-commands* another node y in a tree if x does not dominate y but every node that dominates x also dominates y.

In other words, the c-command domain of a node comprises its sisters and everything contained in them. As an illustration, consider the embedded sentence in (2.18a) together with the tree (2.18b), that represents its constituent structure as it would be commonly assumed. (The category labels are of no particular importance here.)

(2.18) a. The citizens of Uruk heard [$_{\text{CP}}$ whom$_1$ the king welcomed $\_\__1$].

b.



From a representational point of view, this tree is considered a syntactic object that is built by iterative rule applications, or that is defined by well-formedness conditions on trees. The displacement dependency between whom and the gap satisfies the c-command requirement posed above.

From a derivational perspective, this c-command requirement gets a trivial taste. Derivational perspective means that the focus is not on the syntactic object that is built but rather on how it is built. Looking at the tree above from a derivational point of view, we can read it as the history of how the CP was built: the and king were combined in order to build an NP, which was then combined with the result of combining T and the VP, then the resulting TP was combined with C, which was then combined with whom in order to build the CP.

The c-command relation between a displaced expression and the corresponding gap can be couched in derivational terms in the following way (cf. Epstein [35] and Epstein et al. [36]):

> An expression x *c-commands* another expression y if x was combined with y in the course of a derivation. Furthermore, x c-commands everything that was combined to form y, i.e. all of y's constituents.

So according to Epstein, syntactic relations are derivational constructs; they are established when expressions are combined.

If we now assume a movement-like operation that establishes the displacement dependency, i.e. if we assume that a displaced expression originates from the gap position, we can describe the derivational history of whom in our example roughly like this: First, whom is combined with the verb welcomed. This

establishes a syntactic relation between both (this allows for case assignment, for example). Later in the derivation, the original position of whom is replaced by a gap and whom itself is combined with the CP. It thereby c-commands everything contained in this CP, most importantly the position where it originated. Note that this is not specific to our example but holds in general for all displaced wh-phrases. Displaced wh-phrases therefore trivially c-command their corresponding gap.

2.2.2 Rigid locality

There are phrases that block displacement, so-called *islands*, first discussed by Ross [96]. Standardly, two kinds of islands are distinguished:

- *Strong* (or *absolute*) *islands* are phrases out of which no extraction whatsoever is allowed.
- *Weak* (or *selective*) *islands* are phrases out of which some elements may be extracted and others may not.

Let us first look at strong islands, i.e. phrases that do not allow any extraction at all. An example for strong islands are adjuncts. Extraction out of adjuncts seems to be impossible across all known languages and independent of the type of expression that is extracted. An example is given in (2.19) with the adjunct island indicated by brackets.

(2.19) * Whom$_1$ is she sure that Gilgamesh was happy [because he defeated __$_1$]?

Other cases of strong islands, although less universal, are subjects, whose behaviour with respect to extraction shows a clear asymmetry to that of objects: extraction from the subject in (2.20a) is out, whereas extraction of the same wh-phrase from the object in (2.20b) is fine.

(2.20) a. * Whom$_1$ did [a story about __$_1$] amuse you?

b. Whom$_1$ did you hear [a story about __$_1$]?

However, contrary to adjuncts, subjects do not cross-linguistically behave like islands, for there are a variety of languages that do allow extraction from subjects in certain configurations (e.g. Japanese, Hungarian, Turkish, Palauan, see Stepanov [110]). Such cases can actually also be found in English: while extraction from a subject is generally ungrammatical in active sentences like (2.21a), it is possible in their passive counterparts, see (2.21b) (cf. Chomsky [23]).

(2.21) a. * [Of which ship]$_1$ did [the captain __$_1$] defeat pirates?

b. [Of which ship]$_1$ was [the captain __$_1$] found dead?

Adjunct and subject islands are commonly captured by the *Condition on Extraction Domain* (CED) by Huang [55], which can be formulated as follows.

> *Condition on Extraction Domain*
> Extraction out of an XP is possible only if XP is a complement.

Since adjuncts and subjects are commonly considered to be non-complements, the CED predicts them to be islands.

Another example of strong islands are complex noun phrases like in (2.22).

(2.22) * Who$_1$ did Enkidu believe [the claim that Gilgamesh defeated __$_1$]?

Next, let us look at weak islands, i.e. phrases that allow only some expressions to extract. An example for weak islands are wh-phrases and topicalized phrases. They block extraction of the same kind, i.e. a wh-phrase blocks wh-extraction and a topicalized phrase blocks topicalization, as shown in (2.23) and (2.24).

(2.23) a. * Whom$_1$ did you know [where$_2$ the Mogelmons found __$_1$ __$_2$]?

b. * Why$_1$ did you wonder [whether the Mogelmons seeked John __$_1$,]?

(2.24) * [The mogelmons]$_1$, you knew [that John$_2$, __$_1$ seeked to kill __$_2$].

Other contexts that constitute weak islands are induced by negatives and scope-bearing elements. For a more extensive survey on weak island see Szabolcsi & den Dikken [114].

In general it seems that an f-domain blocks f-extraction, where f is some feature that triggers displacement. That is why weak island effects are nowadays often reduced to Rizzi's [95] *Relativized Minimality*, which prohibits movement of some type across an intervener of the same type, e.g. A-bar movement another A-bar moved expression, head movement across another displaced head, and so on. Assuming a more fine-grained distinction of movement types, this principle can account for weak island effects: wh-movement is not possible across another displaced wh-expression, topicalization is not possible across another topicalized expression, and so on. In Chapter 4 we will see how Relativized Minimality follows from our feature checking mechanism and how we can use it to derive the weak islands we just saw.

There is a problem with this approach to weak islands, though. It predicts that extraction should not be blocked by an intervener of a different kind. For example, it should well be possible to extract a wh-phrase from a topicalized phrase, and likewise it should be possible to topicalize an expression disregardless of wh-interveners. However, the facts are slightly different. It appears that topicalization islands are stronger than wh-islands – not only with respect to topicalization but also with respect to wh-movement and relativization. The German examples in (2.25), taken from Müller [81], show this. In (2.25a), a wh-phrase is extracted out of of topicalization island and the result is bad as expected. In (2.25b), on the other hand, an expression is topicalized across

a wh-island and the result is significantly better. The same picture holds for relativization: The ungrammatical (2.25c) shows relativization across a topicalization island and the slightly better (2.25d) shows relativization across a wh-island.

(2.25) *German* (Müller [81])

a. * Was$_1$ glaubst du [gestern$_2$ hat Fritz repariert $\_\__2$ $\_\__1$]?
what.ACC believe you yesterday has Fritz fixed

b. ?? Radios$_1$ weiß ich nicht [wie$_2$ man $\_\__2$ repariert $\_\__1$]?
radios know I not how one fixes

c. * die Radios, die$_1$ ich glaube [gestern$_2$ hat Fritz repariert $\_\__2$ $\_\__1$]
the radios which I believe yesterday has Fritz fixed

d. ?? die Radios, die$_1$ ich nicht weiß [wie$_2$ man $\_\__2$ repariert $\_\__1$]
the radios which I not know how one fixes

We will briefly come back to this contrast in Chapter 4.

An interesting fact about islands in general is that they do not constrain all unbounded dependencies: whereas displacement is subject to strong and weak island constraints, pronominal binding, for example, is not. Examples of this are (2.26) and (2.27). (2.26) is an instance of extraction from an adjunct. Since adjuncts are islands, the sentence is ungrammatical. In (2.27), on the other hand, the displacement dependency is replaced by a pronominal binding dependency. Dispite the island boundary, the sentence is perfectly fine.

(2.26) * Whom$_1$ did Enkidu smile [before Gilgamesh tyrannized $\_\__1$]?

(2.27) [Every citizen]$_1$ was happy [before Gilgamesh tyrannized him$_1$].

Island sensitivity is, in fact, a steady characteristic of extraction. And since other dependencies like pronominal binding generally lack it, it is often taken as a diagnostic for movement: If some operation is island-sensitive, it does involve movement; if it is not island-sensitive, it does not. In the course of the book, I will often draw on this diagnostic. It has to be used carefully, however, for two reasons. The first one is that sometimes where there is an island, there is also a way to get off it. One way to resort linguistic islands is by means of resumptive pronouns, as they occur, for example, in Irish, Hebrew and some varieties of Arabic. An example is the following sentence where the wh-phrase is related to a position inside an adjunct island. If this position were gapped, the sentence would be ungrammatical; if it is filled by a resumptive pronoun, however, it is fine.

(2.28) *Lebanese Arabic* (Aoun & Li [2])

Miin raaħit saamia minduun-ma tšuuf-o?
who left.3FS Samia without see.3FS-him
'Who did Samia leave without seeing?'

The second reason is that a lot of non-syntactic factors pervade island phenomena, especially weak islands. Among them are definiteness as in (2.29), and D-linking (referring to specific members in a pre-established set), as illustrated with 'how many' phrases in (2.30).

(2.29) a. * [Which woman]$_1$ did you discover [the poem about $\_\__1$]?
b. * [Which woman]$_1$ did you discover [Goethe's poem about $\_\__1$]?
c. [Which woman]$_1$ did you discover [a poem about $\_\__1$]?

(2.30) a. * [How many books]$_1$ are you wondering [whether to write $\_\__1$ soon]?
b. [How many books]$_1$ on the list are they wondering [whether to publish $\_\__1$ soon]?

Besides those semantic and pragmatic factors, possibly also processing issues play a role (c.f. Kluender [64]). But since these factors lie outside the scope of this thesis, I will not pay much attention to the data they give rise to. It should just be kept in mind that this kind of data exists and that it is not easily covered within a purely structural dimension.

2.2.3 Relativized locality

At the beginning of this section we saw that languages have different strategies for forming multiple questions: either all wh-phrases stay in situ (as in Chinese and Japanese), all wh-phrases are fronted (as in many Slavic languages), or exactly one wh-phrase is fronted (as in English and German). But this is not all variation there is; also with respect to the surface linear order of the fronted wh-phrases, languages behave differently. We find languages where it plays a role, which wh-phrase is fronted or in which order multiple wh-phrases occur. First take English, a language where exactly one wh-phrase is fronted. As the following example shows, it is not arbitrary which one this is.

(2.31) a. Who$_1$ [$\_\__1$ sought whom]?
b. * Whom$_1$ did [who seek $\_\__1$]?

These ordering effects are called *superiority effects*. Superiority expresses that it is the structurally higher one of two expressions that is targeted by an operation or enters a dependency, where A is structurally higher than B if A c-commands B. In the English examples above, the subject wh-phrase is structurally higher than the object wh-phrase, thus the subject wh-phrase is the one that is displaced.

The intuition behind recent accounts for superiority is that a structural relation must be satisfied in the smallest possible environment in which it can be satisfied. This is commonly captured by the *Minimal Link Condition* (MLC) [20], a version of Relativized Minimality. With respect to displacement dependencies, the MLC requires the structure these dependencies span to be as small as possible. That is, the MLC prevents extraction when there is an intervener

between gap position and front position, where intervention can be understood either in terms of a closer landing position for the extracted expression, or in terms of another expression that is closer to the landing position and could also be extracted. The latter case is instantiated in example (2.31b) above. There, extraction of whom is not the shortest extraction possible because who could also be extracted (since it is also a wh-phrase) and is structurally higher, i.e. closer to the front position.

Now take Slavic languages, where all wh-phrases are fronted. Do they show superiority effects? The answer varies. Some of them do not show superiority effects but rather admit a relatively free word order, as does Czech.

(2.32) *Czech* (Rudin [97])

a. Kdo kdy koho pozval, nevím.
who when whom invited not-know.1.SG
'Who saw whom when, I don't know.'

b. Koho kdy kdo pozval, nevím.

c. Kdy kdo koho pozval, nevím.

Others, like Bulgarian, on the other hand do exhibit ordering effects.

(2.33) *Bulgarian* (Rudin [97])

a. Koj kogo vižda?
who whom sees
'Who sees whom?'

b. * Kogo koj vižda?

However, note that the ordering is different from what superiority would require. Assuming that the position where the subject wh-phrase originates is structurally higher than the position where the object wh-phrase originates, the subject wh-phrase would be required to move first, and only after that the object wh-phrase could move. If movement expands structure, as widely assumed, this would give the ordering in (2.33b).

There are several ways to explain the different orderings we find. One is to conclude that in some languages, Bulgarian among them, it is not the structurally higher element that is extracted but the structurally lower one. This is dubbed *antisuperiority effect* because the ordering is the opposite of what superiority would predict. Another possibility is to claim that languages like Bulgarian do in fact obey superiority, i.e. it is the subject wh-phrase that is extracted first. However the object wh-phrase then does not move to a higher position but instead is 'tucked in' below the subject wh-phrase (cf. Richards [93]). In Chapter 4, we will see how to derive superiority and antisuperiority effects from the syntactic mechanism employed in this thesis.

So far we considered only two wh-phrases when looking at superiority effects, but once we turn to multiple questions with more than two wh-phrases,

things start to get even more interesting. For example, the English example (2.34a) is ungrammatical. According to the Minimal Link Condition that is because the fronted wh-phrase was not the structurally highest one. However this example is suddenly rendered grammatical upon adding another wh-phrase like in (2.34b).

(2.34) a. * Whom did who seek?

b. Whom did who seek where?

A possible way to think about these examples is along the lines of Kayne's connectedness approach [59]. Kayne's insight is that a dependency obeying a certain condition can eliminate the effects of that condition along the path of the dependency. For (2.34b) this would mean that whom does obey superiority with respect to where, and this furthermore voids superiority along the movement path, i.e. with respect to who.

Superiority violations like in (2.34b) can also be observed in languages where all wh-phrases are fronted. The picture for Bulgarian shows that although the wh-phrase that is fronted must be the structurally highest one, the ordering of lower wh-phrases does not play a role at all.

(2.35) *Bulgarian* (Bošković [120])

a. Koj kogo kak e tselunal?
who whom how is kissed
'Who kissed whom how?'

b. Koj kak kogo e tselunal?

Also here, Kayne's general line of thinking can be applied. Richards [94] does so in explaining the above facts with a principle he calls *Minimal Compliance*. It states that, within certain domains, a grammatical constraint has to be respected by one dependency of a particular kind only. Once that is the case, the constraint does not need to be respected anymore by other dependencies of the same kind in the same domain. This principle can be used to explain the Bulgarian examples if we assume that all wh-phrases are involved in one dependency, e.g. a feature checking relation with a particular other expression. The first instance of this checking relation targets the structurally highest wh-phrase and as a result it is fronted. Once this is done, all other wh-phrases are tucked in below (recall Richard's idea of tucking in from above). Now, since superiority was already satisfied, they are not obliged to obey it anymore. Thus the ordering of the lower wh-phrases is free.

Like in the previous section, there should be a final caveat about the empirical underpinnings of the facts mentioned. There are many non-structural factors that influence speaker's judgements with respect to the sentences we considered, among them animacy, D-linking, phonological differences and the distinction between main clauses and subclauses (see e.g. Featherstone [38] and Meyer [76]). But again, these factors lie outside the scope of this thesis and will not play a role in our further explorations.

2.3 Operator scope

Let us now turn to the interpretative side of wh-question constructions. Despite the variety of surface realizations, the wh-questions above do not differ in meaning. Their interpretation amounts to an operator-variable structure of the form 'Which x [. . . x . . .]'. We assume that both the operator 'Which' and the variable x are part of the meaning of the wh-expression (independent of whether it is displaced or not). The wh-expression thus makes two contributions to the sentence meaning: On the one hand, it supplies a variable to fill an argument slot of the verb, and on the other hand, it introduces an operator that binds that variable and takes scope over the whole sentence.

Let us also look at another kind of expressions that denote scope-taking operators: quantificational noun phrases such as every citizen, no god and someone. They are usually not displaced at all, but still the same operator-variable structure underlies their interpretation. The only difference with respect to wh-questions is that the operator involved is not 'Which' but 'For all', 'There is' and the like. Usually, quantificational noun phrases occur in the argument position they bind – in (2.36a) in subject position, in (2.37a) in object position, and in (2.38a) inside another noun phrase. Exceptions are floating quantifiers like all or both, whose position is not fixed but variable, as shown in (2.39).

(2.36) a. Every human is condemned to mortality.
b. For all humans x it holds that x is condemned to mortality.

(2.37) a. The gods awarded someone with an eternal life.
b. For some x it is the case that the gods awarded x with an eternal life.

(2.38) a. [The servant of the ruler of some city] despises tyranny.
b. There is a city x such that the servant of the ruler of x despises tyranny.

(2.39) a. We both should have defeated Huwawa.
b. We should both have defeated Huwawa.
c. We should have both defeated Huwawa.

Now what does it mean for an operator to take scope? The scope of an operator can be specified as follows (cf. Szabolcsi [113]):

> The *scope* of an operator is the domain within which it has the ability to affect the interpretation of other expressions.

Expressions that can be affected comprise pronouns, other quantifiers, and negative polarity items, among others. This thesis will concentrate solely on effects on other quantifiers in form of the relative scope they take. For example, in (2.40), the ancient gods can co-vary with the cultists such that for every cultist there is a different god he worships.

(2.40) Every cultist worships an ancient god.

The scope taking abilities of an operator expression are determined both by its syntactic position and its particular semantics. A collection that explores the semantic properties and their role in scope taking is Szabolcsi [112]. Although those semantic properties play a crucial role for the behavior of operators, we will not consider them at all. Instead, we will concentrate on the structural dimension involved. This is because our main focus is the role that the syntactic position of the operator expression plays in its scope taking. This way, we will end with a structural and feasible yet necessarily non-exhaustive treatment of operator scope.

This said, let us turn to restrictions on operator scope.

2.4 Restrictions on operator scope

Although operator expressions appear to give rise to the same operator-variable structure when interpreted, they do not show uniform scope behavior.

The most important observation with respect to the scope of quantifiers is that, in all the cases we have seen, the scope of the quantifier ranges over the whole clause it occurs in, independent of where exactly it occurs. Furthermore, the scope of a quantifier is restricted to that clause. The following sentence, for example, can only have the reading in (2.41a), where the quantifier everyone has scope over the embedded clause, but cannot have the reading in (2.41b), where it takes scope over the matrix clause.

(2.41) Someone thinks [that everyone can reach eternal life].

a. There is an x such that x thinks that for all y it holds that y can reach eternal life.

b. For all y it holds that there is an x such that x thinks that y can reach eternal life.

Note that the reading in (2.41b) is less specific than the one in (2.41a), i.e. admits more situations in which it is true. For a speaker to include these possibilities, it is thus not sufficient to use (2.41).

In case a clause contains more than one quantifier, their respective scope is not necessarily fixed. A widely acknowledged fact about English are scope ambiguities like in (2.42), which has both the linear scope reading in (2.42a) and the inverse scope reading in (2.42b).

(2.42) Most heroes survive all devastating battles.

a. Most heroes x are such that for all devastating battles y it holds that x survives y.

b. All devastating battles y are such that for most heroes x it holds that x survives y.

Again, b. does not entail a., therefore the reading in b. is not simply a subcase of the one in a., thus cannot be obtained by the linear order of quantifiers in (2.42) but only by their reversed order.

However, this does not hold in general. For example, the following sentence is not ambiguous, despite its containing two quantifiers. Instead it has only a linear scope reading.

(2.43) Most gods admire no human.

So it seems that not every quantifier can outscope other quantifiers. The distinction that is often considered relevant here is one between *strong* and *weak* quantifiers. It was first formulated by Milsark [77] with respect to indefinites and definites and later more broadly conceived and formalized by Barwise & Cooper [6]. Weak quantifiers are intersective (i.e. their truth depends only on the intersection of the two sets they relate) or, stated in different terms, symmetric (i.e. their restriction and scope can be exchanged without change in truth-conditions). Examples are some, no and less than two. Strong quantifiers, on the other hand, are non-intersective, or asymmetric. Examples are every, most, and not all. The distinction can be seen clearly with there-sentences: they are fine with weak quantifiers but not with strong ones.

(2.44) a. There are some archaeologists searching for new tablets.
b. There are no archaeologists searching for new tablets.
c. There are less than two archaeologists searching for new tablets.

(2.45) a. * There is every archaeologist searching for new tablets.
b. * There are most archaeologists searching for new tablets.
c. * There are not all archaeologists searching for new tablets.

Some quantifiers have both a weak and a strong reading. Examples are many and few. With respect to the there-test, they pair with weak or strong quantifiers, depending on the reading. That is, the sentences in (2.46) do not allow a strong reading, i.e. the reading that many/few of the archaeologists were searching for new tablets, but they do allow a weak reading, i.e. the reading that many/few of the people searching for new tablets were archaeologists.

(2.46) a. There are many archaeologists searching for new tablets.
b. There are few archaeologists searching for new tablets.

When modelling different scope behavior in Chapter 5, I will take up the intuition that only strong quantifiers can outscope other quantifiers, but weak quantifiers cannot (cf. e.g. Ruys [98]).

Among weak quantifiers there are some that deserve closer attention: certain indefinites show exceptional behavior in that their scope is not clause-bound like that of other quantifiers. Instead they can take almost unrestricted wide scope. For example, while the scope of every is restricted to the subclause

of (2.47) that it occurs in and the sentence therefore only has a linear scope reading, the indefinite in (2.48) can take scope over the intermediate and matrix sentence as well. This is shown in (2.48b) and (2.48c), both of which are available readings although they do not entail the linear reading (2.48a).

(2.47) a. Some archaeologist were happy [if every tablet could be deciphered].

b. We invited someone [who deciphered every fragment you found].

(2.48) We believe [it is unlikely [that some tablet cannot be deciphered]].

a. We believe that it is unlikely that there is a tablet x for which holds that x cannot be deciphered.

b. We believe that there is a tablet x for which holds that it is unlikely that x cannot be deciphered.

c. There is a tablet x for which holds that we believe that it is unlikely that x cannot be deciphered.

The scope freedom of indefinites does not only hold for embedded clauses but also shows with other scope islands. For example, adjuncts and coordinate constructions restrict the scope of many quantifiers (cf. (2.49a) and (2.50a), which have no inverse scope reading) but are not able to restrict the scope of indefinites (cf. (2.49b) and (2.50b), which allow an inverse scope reading).

(2.49) a. Many Dolions were killed [because no-one realized the mistake].

b. Many Dolions were saved [because someone realized the mistake].

(2.50) a. Every mythology expert thinks that [Jason and every argonaut] sought the golden fleece.

b. Every mythology expert thinks that [Jason and some argonauts] sought the golden fleece.

This concludes the scopal behavior of quantifiers. To summarize, quantifiers can take scope only over the clause they occur in, except for certain indefinites which are exceptionally free in taking scope. Within their scope, quantifiers can outscope other quantifiers if they are strong, but cannot if they are weak. In Chapter 5 we will look at how to model these different scope behaviors. We will do so in a purely structural way, so we will not have anything to say about how weakness or strength are connected to semantic properties (such as being intersective or not).

The exceptional behavior of some indefinites led researchers to conclude that there are two types of indefinites: those that par with quantifiers and therefore are subject to the same scope restrictions, and those that par with referential expressions, thus are not quantifiers and therefore not subject to scope restrictions (cf. Fodor & Sag [40], among others). I want to stay neutral with respect to this discussion. I will therefore concentrate on non-indefinite

quantifiers when modeling scope behavior in Chapter 5. However, I will add a section on how to model exceptional wide scope with this mechanism as well. This will moreover prove useful for wh-operators in some in situ and scope marking languages.

Let us now turn to operators associated with wh-phrases. With respect to their scope, we can roughly state the following three observations. First, if a wh-expression is displaced, the corresponding wh-operator usually takes scope over the clause it was displaced to. This can be seen in the following two English examples. In (2.51a), the wh-phrase is displaced inside the embedded clause, over which it takes scope. In (2.51b), on the other hand, it is displaced to the matrix clause and indeed takes scope over the whole sentence.

(2.51) a. Gilgamesh wonders [whom$_1$ the gods favored $\_\__1$ more than him].

b. Whom$_1$ did Gilgamesh think [that the gods favored $\_\__1$ more than him]?

Second, in the presence of a scope marker, the scope marker determines the clause over which the wh-operator takes scope. This is illustrated for Japanese in the following examples. In case of (2.52a), the question particle ka marks the embedded clause, in case of (2.52b), it marks the matrix clause. The scope of the wh-operator behaves accordingly.

(2.52) *Japanese* (Bošković [122], Cresti [29])

a. Peter-wa [anata-ga dare-o mita-ka] tazuneta.
Peter-TOP you-NOM who-ACC saw-Q asked
'Peter asked whom you saw.'

b. Kimi-wa [dare-ga kai-ta hon-o yomi-masi-ta]-ka?
you-TOP who-NOM wrote book-ACC read Q
'Which person x is such that you read a book that x wrote?'

Third, for in situ wh-expressions without an obligatory scope marker there are two possibilities. Either they take scope inside the clause in which they occur, as illustrated in the following Hindi example (where jaan ('know') can take both interrogative and propositional complements).

(2.53) *Hindi* (Bhatt [10])

Wajahat jaan-taa hai [ki Rima kis-ko pasand kar-tii hai]
Wajahat know-M.SG be.PRS.SG that Rima who-ACC like do-F be.PRS.SG
'Wahajat knows who Rima likes.'
* 'Who does Wahajat know Rima likes?'

Or they take scope in an arbitrary interrogative clause, as in the following Chinese example.

(2.54) *Mandarin Chinese*[2]

Zhangsan zhidao [shei du-le shu]
Zhangsan knows who read-ASP books
'Who does Zhangsan know read books?'
'Zhangsan knows who read books.'

2.5 Two sides of the same coin?

In the previous section, we saw that the syntactic position of a quantifier or wh-phrase and the scope position of the operator it denotes sometimes coincide and sometimes diverge. Let us recall the concurrences and mismatches and then consider the implications they have for the syntax/semantics interface.

2.5.1 Concurrences

The concurrences are evident: Many languages displace one or all wh-expressions to the position where they take scope. An instance is English with the following two examples. In (2.55a), the wh-operator is displaced to the matrix clause and indeed takes scope over the whole sentence, while in (2.55b), the wh-operator stays within the embedded clause and takes scope only there.

(2.55) a. Who$_1$ do you think [Enki loves $\_\__1$]?

b. You know [who$_1$ Enki loves $\_\__1$].

Languages that do not displace wh-expressions often make use of particles that occupy the position where the in situ wh-expression is intended to take scope. An example are the following two sentences of Japanese. In (2.56a), the question particle ka modifies the matrix clause, the wh-operator thus takes scope over the whole sentence. In (2.56b), on the other hand, the question particle ka modifies the embedded clause, the wh-operator thus takes scope only over the embedded clause.

(2.56) *Japanese*

a. Anata-wa Enki-ga dare-o aisiteiru-to omotte-imasu-ka?
you-TOP Enki-NOM who-ACC love-COMP think-be-Q
'Who do you think that Enki loves?'

b. Anata-wa Enki-ga dare-o aisiteiru-ka sitte-imasu.
you-TOP Enki-NOM who-ACC love-Q know-be
'You know who Enki loves.'

[2] All Chinese examples without a reference were checked with a native speaker (mostly Min Que).

These concurrences led to theories assuming a tight connection between displacement and scope. This is quite natural given that most formal linguists share Montague's assumption that there is a strict correspondence between syntax and semantics. More specifically, the syntactic and semantic principles of combination are designed to be homomorphic: Every syntactic rule is paired with a semantic rule. More specifically, it was proposed that the syntactic operation of displacement is mapped to the semantic operation of establishing scope. It became quite common to assume that displacement creates operator-variable structures and that therefore there is a one-to-one correspondence between the syntactic c-command domain of an expression and the semantic scope of the operator it denotes (see e.g. Heim & Kratzer [49]).

2.5.2 Mismatches

There are quite a few mismatches as well. Let us recall the three major ones. The first one is that languages can establish operator scope without displacement. Most quantificational noun phrases are a case in point, since they do not show any signs of having been displaced. Also, in some languages, in situ wh-phrases do not show characteristics of displacement. For example in Chinese and Quechua, questions with in situ wh-phrases can violate island constraints.

(2.57) *Mandarin Chinese*

Ni xiang-zhidao [wo weishenme gei Akiu shenme]?
you wonder I why give Akiu what
'Which reason x is such that you wonder what I give to Akiu because of x?'

(2.58) *Ancash Quechua* (Cole & Hermon [24])

Qam kuya-nki [ima-ta suwaq nuna-ta]?
you love-2PL what-ACC steal man-ACC
'Which x is such that you love the man who stole x?'

The second mismatch is that even in cases where wh-expressions are displaced, the overt position of the wh-phrase does not always coincide with the scope position of the corresponding operator. For examples, the in situ wh-phrase in the Japanese example (2.59) seems to have been displaced covertly for it gives rise to an island violation.

(2.59) *Japanese*

* Anata-wa [Taro-ga dare-o hometa-ka doo-ka] sitte-imasu-ka?
you-TOP Taro-NOM who-ACC praised-Q how-Q know-be-Q
'Which x is such that you know whether Taro praised x?'

Another example is the Malay question (2.60) we already saw. The wh-phrase is fronted inside the embedded clause but takes scope over the matrix clause.

(2.60) *Malay* (Cole & Hermon [25])

Kamu fikir [[ke mana]$_1$ Fatimah pergi $\_\_{}_1$]?
you think to where Fatimah go
'Where do you think that Fatimah went?'

Similar instances are wh-phrases that occur deep inside a pied piped phrase like in (2.61). Also there the wh-phrase occurs in a position lower than the one from where it takes scope.

(2.61) [The king of which city]$_1$ did Ishtar admire $\_\_{}_1$?

And the third mismatch is that sentences with more than one scope-taking expression display scope ambiguities. That is, despite the syntactic order among operator expressions, their semantic scope is not ordered. We saw examples of this in the last section. For instance, the relative scope of the two quantifiers in Most heroes survive all devastating battles was not fixed: either most heroes scopes over all devastating battles or vice versa.

To summarize, by far not all wh-expressions are displaced to their scope position or related to a scope marker. And other operator expressions, such as quantificational noun phrases, are neither displaced nor does their syntactic position correspond to the position where they take scope. That is, in many cases, the scope of an expression cannot be read off of its syntactic position.

2.5.3 Reconciling concurrences and mismatches

The mismatches between displacement and scope required theories assuming them to be tightly connected to undergo considerable adjustments. They needed to change either the syntactic operations in order to fit with semantics (e.g. by positing a covert displacement rule with slightly different properties than overt displacement), or the semantic operations in order to match the syntactic structures (e.g. by positing additional strategies of scope taking that do not rely on displacement). Let us briefly look at the most prominent adjustments that were proposed.

Adjusting the syntactic operations is based on the idea that the positions an expression can be interpreted in are those positions through which it moved in the course of a derivation. For example, quantificational noun phrases that occur in a position lower than where they take scope are considered to actually move to their scope position in the course of the derivation. To this end, May introduced a displacement rule called *Quantifier Raising* that moves quantifiers to their scope position on a level that is input to interpretation but invisible to phonology, hence is not spelled out (see May [74], [75]). This abstract syntactic level of representation is called *Logical Form* (or short: LF). Soon the common view on LF was one according to which all operators occupy a position that uniquely determines their absolute and relative semantic scope. That is, for every semantic reading, a different syntactic structure was postulated. This preserved a strict one-to-one correspondence between syntax and semantics.

Let us look at an example: The sentence in (2.62a) would have a logical form like in (2.62b).

(2.62) a. Whom did everyone fear?

b. [whom$_1$ [everyone$_2$ [__ $_2$ fear __ $_1$]]]

And the ambiguous sentence (2.63) would give rise to the two logical forms in (2.63a) and (2.63b), depending on the order in which the two quantificational noun phrases are raised.

(2.63) Most heroes survive all devastating battles.

a. [[most heroes]$_1$ [[all devastating battles]$_2$ [__ $_1$ survive __ $_2$]]]

b. [[all devastating battles]$_2$ [[most heroes]$_1$ [__ $_1$ survive __ $_2$]]]

The rationale behind LF was that the principles of grammar do not only determine possible syntactic structures but also possible logical forms that represent those syntactic aspects that are relevant for interpretation. Also, LF seemed to come for free because logical forms were assumed to be common syntactic structures and Quantifier Raising seemed to be the regular movement operation one already had. At least at first sight. At second sight, the displacement we know and Quantifier Raising show quite different patterns, as we saw when looking at constraints on displacement and scope: While the scope of quantifiers is not affected by islands but is clause-bound, displacement fails to reach beyond islands but can cross clause boundaries quite easily. Other technical objections against Quantifier Raising were that it is an adjunction rule, while no other core grammatical principle involved adjunction, and that it does not target a specific position, opposed to other displacement rules. Moreover, it did not behave like other displacement rules in that it was not feature-driven but applied only in order to assign semantic scope. Tanya Reinhart therefore later proposed that Quantifier Raising is only applied if there is no other way to arrive at a certain semantic interpretation (see e.g. Reinhart [90]).

Another kind of adjustment leaves the syntactic operations and structures as they are and instead changes the semantic operations. These adjustments are mainly based on the observation that situ wh-expressions show no sign of covert displacement and also in other respects differ from their displaced sibblings. This suggests that displacement is not essential to the interpretation of in situ wh-phrases and that languages have a different strategy to deal with them. There are several strategies that were proposed to deal with scope assignment of in situ wh-phrases. One of them was given by Baker [3]. He assumed that an in situ wh-phrase is coindexed with a Q-morpheme that resides in complementizer position, where it takes scope, representative for the wh-phrase. Engdahl [34], on the other hand, proposed to use the storage mechanism developed by Cooper ([26],[27]) and refined by Keller [61] to interpret in situ wh-phrases. This amounts to employing a stack that stores quantifier

interpretations which can be drawn from it whenever the semantic construction reaches the scope position. Other recent work that puts Cooper stores to use is Kobele's thesis [65]. Another mechanism that became popular for assigning scope goes by the name of *Unselective Binding*. It was developed by Lewis [71] and Heim [48] as a non-quantificational treatment of indefinites and was later also used for the interpretation of in situ wh-phrases. The idea is that indefinites and in situ wh-phrases are interpreted as open expressions that gain quantificational force only by having their free variable bound by a c-commanding operator that happens to be around. Reinhart [89], in a similar vein, assumed in situ wh-phrases to be indefinites that are bound by existential closure, but proposed to treat them not in terms of unselective binding but rather in terms of choice functions. In short, many approaches settled for assuming movement for displaced wh-phrases and an alternative scope assignment strategy for in situ wh-phrases.

To summarize, there are two ways to account for mismatches between displacement and scope while saving a strict correspondence between syntax and semantics: adjust either the syntactic or the semantic operations. But there is another possibility, of course. We can decide to give up the strict correspondence between syntax and semantics. A weak way to do this is to give up the one-to-one correspondence between syntactic structures and semantic readings. Instead we can assume that one syntactic structure is associated with one underspecified semantic representation (leaving the scope of quantifiers unspecified, for example), which then yields several semantic readings once it is specified (with the scope of quantifiers fixed). Examples for underspecification approaches are the algorithm by Hobbs & Shieber [53], *Quasi Logical Form* [1], UDRT [92], *Hole Semantics* [12], and *Minimal Recursion Semantics* [28].

All the above approaches have in common that they take the parallels between displacement and scope to be the normal case and then look for a way to account for the mismatches. Considering the quantity and quality of the mismatches, I want to explore the opposite view, viz. that the mismatches are the normal case. I want to propose that displacement and operator scope are two separate mechanisms, not necessarily working in parallel. It is then straightforward that in many cases they do not coincide. As a consequence, we get mismatches for free but have to account for the cases where displacement and scope, in fact, do coincide.

So, I will follow the general line of thinking of underspecification approaches in giving up a strict correspondence between syntax and semantics. But I will do so in a different way. I will assume that there is a core system of grammar for which there is indeed a strict correspondence between syntax and semantics. On top of that, however, I assume grammar to employ other procedures – purely syntactic ones without an effect on interpretation as well as purely semantic ones with no syntactic counterpart. And it is these procedures that I propose to handle displacement and operator scope.

2.6 A brief tour through the thesis

The general goal of this thesis is an algorithm for systematically constructing and linking forms and meanings. This algorithm is supposed to cover wh-displacement and operator scope, and it should be explicit enough to be implementable in a machine.

I will develop this algorithm in three steps. The first step, Chapter 3, is to carve out a core system for combining simple expressions into more complex ones. I will take expressions to be form-meaning pairs, where forms will be represented as typed strings together with syntactic features, and meanings will be typed terms of a lambda calculus. There will be a mapping between types of forms and types of meanings, which ensures that syntax and semantics are in sync.

The other two steps will be to extend the core system with a syntactic procedure for displacement and a semantic procedure for operator scope.

The syntactic procedure for displacement is the topic of Chapter 4. It will operate only on forms and will have no effect on meanings. Displacement will be driven by features. Expressions that carry features will not be combined immediately but instead will be kept and used only when they are in a local configuration with another expression that carries a matching feature. This approach to displacement resembles much work in generative grammar theories. It differs, however, in not building elaborate syntactic structures such as trees. The expressions that are stored because they still need to check features will be the only structure we will have and it will mirror only a rudimentary part of familiar constituent structure. Chapter 4 can therefore also be read as an exploration of how we can derive restrictions on displacement, such as island constraints, with as little structure as possible.

Chapter 5 will then introduce a semantic procedure for establishing scope. Contrary to the syntactic procedure, which operates only on forms, the semantic procedure will operate only on meanings. The main component will be a rewriting rule on semantic terms, which establishes scope by means of control transfer. The effect of this rewriting rule will be very similar to Montague's familiar *Quantifying In* rule (cf. [78]). The difference to most theories on scope, however, is that it will suffice to assume one single scope taking mechanism for all operators, be it operators denoted by quantificational noun phrases, by displaced wh-phrases, or by in situ wh-phrases. And, most importantly, displacement will not play a crucial role for the interpretation of any of these.

Chapter 6 then provides an implementation of both the core system and its two extensions for displacement and operator scope.

Finally, Chapter 7 summarizes the thesis, puts it into perspective and explores its implications. At the end of the book, we will have developed and implemented an algorithm for systematically constructing forms along with their meanings.

3

The base grammar

The general point of view this thesis takes is that the expressions of a language are generated by rules that operate on a finite collection of atomic expressions in order to build other, more complex expressions. This chapter is about explicating what expressions are and how they are combined. We will start by specifying expressions to be form-meaning pairs, with forms being the phonological representation and meaning being the semantic representation of the expression. Then we will define a simple operation for combining two form-meaning pairs into another form-meaning pair. It will be based on common assumptions of generative grammar and cover simple cases not yet involving displacement and operator scope. In the end of the chapter, we will have carved out a mechanism to mediate between form and meaning with a smooth interface that ensures that they are build in accordance with each other.

Let us start with the building blocks of our grammar: expressions. As already mentioned, expressions are defined as simple form-meaning pairs (and not as trees, for example). We will extend this definition in the next chapter.

$$\textbf{Expression} ::= (\textbf{Form}, \textbf{Meaning})$$

Throughout the thesis I will use Backus-Naur-Form (BNF) when stating definitions. I write non-terminal symbols in **bold face** and terminal symbols in normal font or *italics*. For example, the BNF rule $\mathbf{A} ::= b \mid \mathbf{B} \mid c\,\mathbf{D}$ expresses that a non-terminal **A** can be rewritten as either a terminal b or a

non-terminal **B** or a terminal c followed by a non-terminal **D**. The definition of expressions above, thus expresses that an expression is a pair of whatever **Form** and **Meaning** rewrite to (what this is, we will see in a minute). The set of all expressions is thus defined as the set of all pairs that we can build from any form and any meaning.

Forms are usually assumed to be sequences of sound making up a linguistic utterance, possibly together with some syntactic information. We will represent sequences of sounds as strings for ease of exposition. This is way too rough from a phonological perspective, but it will suffice for our purposes. So for us, form comprises a string together with necessary syntactic information such as a category. Meaning, on the other hand, uniquely determines the denotation of the expression. For us, it will be an expression of a lambda calculus. The next two sections will specify form and meaning further.

By defining expressions as form-meaning pairs, we take a stand with respect to how form and meaning are linked. We follow the conception that the meaning of an expression is constructed in parallel to the construction of the form. According to this view, expressions are two-dimensional objects that encode syntactic as well as semantic information. Other frameworks instantiating this view are, for example, Head-Driven Phrase Structure Grammar [87].

The opposed view consists in assuming that the meaning of an expression is assembled only after the construction of its form is finished. Transformational grammar theories such as Government and Binding theory exemplify this view. There, the syntactic structure is constructed prior to semantic interpretation. So a form is paired with a meaning only after all syntactic operations have applied. And in most cases, syntax is where all combinatorial action happens; phonology and semantics only interpret the finished syntactic structure. Sound and meaning are therefore derived from syntactic structure. An operation like Quantifier Raising fits very well with this picture: Syntax starts by building the surface structure of an expression, which is then interpreted by phonology. After that, the structure building process continues and produces a logical form that can then be input to semantics.

This is different from Montague's approach to grammar, where the steps for building a surface string are interspersed with the steps for building a logical form. This integration brings about that every expression that can be constructed receives an interpretation. That is, the syntactic and semantic well-formedness of an expression is defined simultaneously at each level of constituency. This is what we have in mind when devising an algorithm that computes the syntactic and semantic representation of an expression in tandem. This procedural view, in fact, also underlies recent work in transformational grammar. Many approaches in the Minimalist Program assume a step-wise procedure for the generation of utterances along with their meanings. The introduction of phases (c.f. Chomsky [21],[22]), for example, already chunks the syntactic structure that is built before being input to interpretation, and Epstein & Seely [37] later proposed to even let every step of the syntactic com-

putation be input to the semantic computation. The incremental view on the generation of expressions fits very naturally with the computational perspective we are taking. Therefore it should not be surprising to find our explorations in this general line of thinking.

Now let us move to explicating forms and meanings. In doing so, we will disregard all dependencies that are not relevant for the purpose of the thesis, that is case assignment, agreement, and the like.

3.1 Form

Our starting point is rather naive. Forms simply correspond to strings and syntactic operations combine those strings into other strings. The main conceptual goal will be to minimize the amount of structure generated by the grammar. Operations therefore do not build any hierarchical constituent structure unless necessary. (And it will be necessary only next chapter, when we consider displacement.) This approach is in the spirit of all syntax theories that take syntactic operations to be local and not have access to anything in the past or future of a derivation.

For combining strings we use string concatenation, i.e. a function that maps two input strings to the juxtaposition of both. We refer to it as $+\!\!+$. For example, mighty $+\!\!+$ Gilgamesh yields the string mighty Gilgamesh.

Next, we need to restrict string combination because we do not want to allow the concatenation of every string with every other string. For instance, we do not want to allow the concatenation of the string mighty with the string it rains. In order to restrict the possibilities to those which are actually grammatical, we classify strings according to their ability to combine with other strings, like categorial grammars do. The sets of strings that share combinatorial behavior are named by types. Types thus encode with which kind of other strings a string can combine, and which kind of string this combination yields.

Types are defined as being either *basic types*, that specify the syntactic category of an expression (noun phrases, verb phrases, and so on), or as being *functional types*, that specify with strings of which type (and in which order) a certain string can combine.

Definition 1. *The set of syntactic types is given by:*

Cat	::=	NP	*(noun phrases)*
	|	N	*(common nouns)*
	|	VP	*(verb phrases)*
	|	CP	*(sentences)*
	|	**Cat** $\rightarrow$ **Cat**	*(functional types)*

I will use the terms 'syntactic type', 'category' and 'syntactic category' interchangeably.

Now forms can be defined to be typed strings, with :: standing for 'is of type'. I will print types in grey font for better readability. (Also this definition will be extended in the next chapter.)

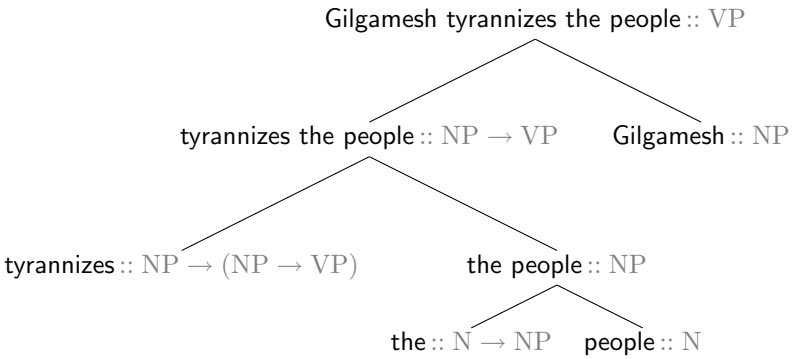
Form ::= **String** :: **Cat**

Here are some examples of forms:

Gilgamesh :: NP
people :: N
the :: N → NP
tyrannizes :: NP → (NP → VP)

Let me first mention that there is no principled reason here for using NP instead of DP as category for noun phrases. We could as well specify the type of Gilgamesh as DP, the type of the determiner as N → DP (or even NP → DP), and the type of the verb as DP → (DP → VP).

Now we want to combine these example strings according to their categories, e.g. like specified in the following derivation tree:



In order to do so, we need an operation that concatenates strings according to their types. We will call this function **merge**. It basically combines a string a of type $c_1 \to c_2$ with a string b of type c_1 and yields a new string $a + b$ of type c_2. We say that *a subcategorizes for b*, or that b is subcategorized by a. So we can, for example, combine the string the :: N → NP with the string people :: N in order to form the new string the + people :: NP. Furthermore, I will follow the notions of generative grammar and call a subcategorizing string *head* and a string that is subcategorized a *complement*.

Now there is one more thing we have to take care of: linearization. Defining **merge** like we just stated it would always concatenate the complement to the right of the head. However, this is not the case. Consider the derivation tree above again. The string tyrannizes the people :: NP → VP combines with the string Gilgamesh :: NP and although the latter is subcategorized, it has to be concatenated to the left.

There are several ways to incorporate linearization. One is to follow Categorial Grammars and add directionality to the functional types. We would then have two functional types, which we can write as $c_1 \backslash c_2$ and c_2 / c_1, or closer to our previous notation as $c_1 \rightarrow c_2$ and $c_2 \leftarrow c_1$. Strings of both categories subcategorize for a string of category c_1 and when combined with it yield a new string of category c_2, but in the former case the subcategorized string is concatenated to the left and in the latter case it is concatenated to the right.

Generative grammar encodes this difference in another way, namely by distinguishing two kinds of subcategorized expressions, complements and specifiers. Complements are expressions that are merged first and specifiers are expressions that are merged later. In a language like English, complements are always concatenated to the right of the subcategorizing expression and specifiers are always concatenated to the left. (See Kayne [60] for the proposal that this linearization order is in fact universal and underlies all languages.) Since we have no means to distinguish first merge from second merge, we will stick to encoding the direction in which an expression is concatenated in its type. We do so by introducing a diacritic $^<$ that marks categories of strings which are concatenated to the left. Strings of categories without this diacritic will be concatenated to the right. For example, the category of tyrannizes changes to NP → (NP$^<$ → VP). It now selects for an NP that is concatenated to the right and an NP that is concatenated to the left. This gives the right word order, as we already assumed it in the example tree above. It is important to note that the diacritic $^<$ does not change the category, so a string that subcategorizes for an NP$^<$ can still be combined with an NP without the diacritic.

3.2 Meaning

Now we want to pair the typed strings with meanings. So what are meanings? In the Montagovian tradition, a semantics for natural language is specified by translating natural language expressions into expressions of a logical language that is then subject to a modeltheoretic interpretation. Throughout this book, we will stay in this tradition. As logical language we will use a typed lambda calculus with constants (see e.g. Barendregt & Barendsen [4] for an introduction). The modeltheoretic interpretation is assumed to be standard and will not be explicated. The reason is that the mechanism for quantifier scope devised in Chapter 5 will solely rely on rewriting rules. We are therefore only interested in the structure of our semantic expression, not in the relation they bear to the real world (or a model thereof).

To start with, we define meanings as expressions of a lambda calculus decorated with a type **Type**. Such a type is either a *basic type* (e or t) or a *functional type*. We will slightly extend the type inventory in Chapter 5, but for now this is all we need.

Type	::=	e	*(entities)*
	|	t	*(truth-values)*
	|	**Type** $\rightarrow$ **Type**	*(functions)*

Now the language for meanings consists of the following expressions:

- constants c, which will comprise nullary to n-ary predicate constants such as the individual constant *enkidu*, the one-place predicate *king*, and the two-place predicate *fight*, as well as logical constants and operators such as $\wedge$ and $\exists$
- variables x for every type
- abstractions $\lambda x.E$ (with E an expression)
- applications $(E_1\ E_2)$ (with E_1, E_2 expressions of the language)

Meanings are defined as typed expression **Meaning**, where I again use :: to stand for 'is of type' and again print types in grey font, for better readability. I assume that the language contains variables x for every type, and that abstractions and applications are typed in the standard way. The definition of meanings uses τ, τ_1, τ_2 as variables ranging over types. It will be extended in Section 5.2 of Chapter 5 below.

Meaning	::=	$c :: \tau$	(constants)
	|	$x :: \tau$	(variables)
	|	$(\lambda x :: \tau_1.\textbf{Meaning} :: \tau_2) :: \tau_1 \rightarrow \tau_2$	(abstraction)
	|	$(\textbf{Meaning} :: \tau_1 \rightarrow \tau_2\ \textbf{Meaning} :: \tau_1) :: \tau_2$	(application)

Up to now, lambda abstraction is the only variable binding operation we have, and in fact it is the only one we need for now (later we will add ξ as another variable binder, however). It relates a variable with a place in an expression that is named by that variable. In $\lambda x.E$ with E some expression, E is the *scope* of the lamba operator. We say that all occurrences of x in E are *bound*. For example, in the expression $\lambda x.(\textit{king}\ x)$, the variable x is bound by

λ. If there is no enclosing lambda abstraction that binds a variable x, then x is *free*.

The operational semantics of the calculus is given by the usual beta-reduction (β) and eta-reduction (η).

$$(\lambda x.E_1 \;\; E_2) \rhd E_1\{x \mapsto E_2\} \qquad (\beta)$$

$$\lambda x.(E\ x) \rhd E \qquad (\eta)$$
$$\textit{where } x \textit{ is not free in } E$$

Beta-reduction is substitution of every free occurrence of x in E_1 by E_2, notated as $E_1\{x \mapsto E_2\}$. We assume this substitution to be capture avoiding, i.e. ensure that the variables of E_1 and E_2 have different names. Eta-reduction tells us that, for example, $\lambda x.(\textit{king}\ x)$ and *king* express the same predicate.

Since we do not want the pairing of forms and meanings to be arbitrary, we will restrict it. To this end, we assume that forms are paired with meanings in such a way that the following mapping $^\circ :: \mathbf{Cat} \to \mathbf{Type}$ between syntactic and semantic types is satisfied (it will be refined in Section 5.2 below).

Definition 2. *We assume a mapping $^\circ$ between syntactic and semantic types, such that:*

$$\begin{aligned}
\mathrm{NP}^\circ &= e \\
\mathrm{N}^\circ &= e \to t \\
\mathrm{VP}^\circ &= t \\
\mathrm{CP}^\circ &= t \\
(c^<)^\circ &= c^\circ \\
(c_1 \to c_2)^\circ &= c_1^\circ \to c_2^\circ
\end{aligned}$$

That is, atomic syntactic types are mapped to a stipulated semantic type, presumably in accordance with the modeltheoretic interpretation one has in mind. Complex syntactic types are mapped straightforwardly to complex semantic types so that syntactic implication corresponds to semantic implication. Moreover, linearization has no effect on the mapping, i.e. the syntactic difference of being concatenated to the left or right is lost on the semantic side.

Here are some simple example denotations of lexical items that satisfy the mapping $^\circ$. In the next section we will see how they can be combined into a sentence.

Form		Meaning	
Gilgamesh	:: NP	*gilgamesh*	:: e
goddess	:: N	*goddess*	:: $e \to t$
weep	:: $\text{NP}^< \to \text{VP}$	*weep*	:: $e \to t$
slay	:: $\text{NP} \to (\text{NP}^< \to \text{VP})$	*slay*	:: $e \to (e \to t)$

3.3 Combining form-meaning pairs

Now we want to combine form-meaning pairs into new form-meaning pairs. The combination should be a combination of the forms paired with a combination of the meanings. To keep things simple, I assume that the combination of forms corresponds to string concatenation and the combination of meanings corresponds to functional application. The operation responsible for this, **merge**, is defined as follows:

Definition 3. *Let s_1, s_2 range over typed strings and E_1, E_2 range over typed semantic expressions,*

$$\textbf{merge}\ (s_1, E_1)\ (s_2, E_2) = (s_1 \oplus s_2, (E_1\ E_2))$$

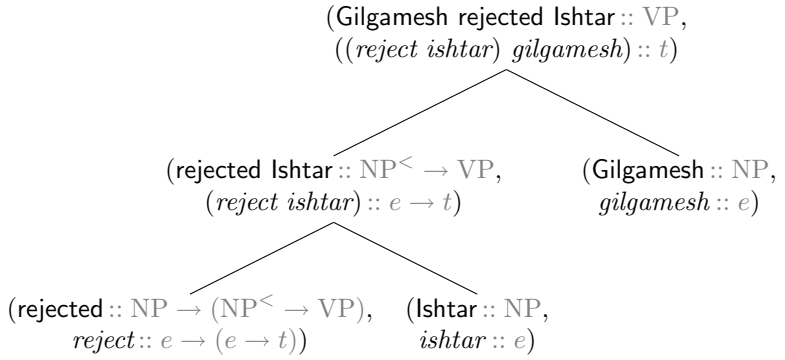
Where $\oplus$ is an operation that concatenates two strings of matching categories, with the order dependening on the linearization diacritic. Letting c, c' range over syntactic categories, $\oplus$ is defined as follows:

$$s_1 :: c \to c' \oplus s_2 :: c\ \ = s_1 +\!\!+ s_2 :: c'$$
$$s_1 :: c \to c' \oplus s_2 :: c^< = s_2 +\!\!+ s_1 :: c'$$

As an example, consider the VP Gilgamesh rejected Ishtar. The lexical items involved are the following:

- (Gilgamesh :: NP, *gilgamesh* :: e)
- (Ishtar :: NP, *ishtar* :: e)
- (rejected :: $\text{NP} \to (\text{NP}^< \to \text{VP})$, *reject* :: $e \to (e \to t)$)

First we merge the verb rejected with the object NP Ishtar, and then we merge the resulting expression with the subject NP Gilgamesh. The derivation tree looks like this:



Note that **merge** only succeeds if both string concatenation and functional application succeed, thus if both the syntactic and the semantic types match. In fact, presupposing the mapping $^\circ$ from the previous section to hold for lexical items ensures that every syntactically well-typed expression we can build is paired with a denotation that is semantically well-typed. Or to state it more precisely: For every expression $(s :: c, E :: \tau)$ consisting of a syntactic form s of category c and a semantic expression E of type τ, it holds that if s is well-typed, then so is E. Morover, it holds that $c^\circ = \tau$.

For lexical items this is a requirement we have to impose on the lexicon. The rest then follows straightforwardly from the definition of **merge**.

That is, with respect to the core system that we considered in this chapter, syntax and semantics work in tandem and never part company.

3.4 Summary and limitations

Up to this point we have a procedure for generating expressions that is very much like context-free phrase structure grammars. We took expressions to be form-meaning pairs, assuming a mapping $^\circ$ between syntactic and semantic types that ensures a close correspondence between the paired forms and meanings. We then defined an operation **merge** for constructing more complex form-meaning pairs by combining forms and combining meanings in parallel. Forms were combined by string concatenation and meanings were combined by functional application.

With the mechanism we have, we can generate expressions that do not involve non-local dependencies. Figure **??** provides a lexicon that gives an idea of a simple fragment we can handle. Here the syntactic categories are extended with adverbial phrases AdvP for which $^\circ$ is defined as $\text{AdvP}^\circ = e \to t$. We can use the example lexicon to generate expressions like the following:

- (Gilgamesh defeated Huwawa :: CP, $((\textit{defeat huwawa})\ \textit{gilgamesh}) :: t$)
- (Ishtar is almighty :: CP, $(\textit{almighty ishtar})$)
- (Enki fought without fear :: CP, $((\textit{fight enki}) \wedge ((\textit{without fear})\ \textit{enki})) :: t$)

Figure 3.1: Example lexicon

(Gilgamesh :: NP, $gilgamesh :: e$)

(Enkidu :: NP, $enkidu :: e$)

(Ishtar :: NP, $ishtar :: e$)

(Huwawa :: NP, $huwawa :: e$)

(Enki :: NP, $enki :: e$)

(fear :: N, $fear :: e \rightarrow t$)

(wept :: $\text{NP}^{<} \rightarrow \text{VP}$, $weep :: e \rightarrow t$)

(fought :: $\text{NP}^{<} \rightarrow \text{VP}$, $fight :: e \rightarrow t$)

(defeated :: $\text{NP} \rightarrow (\text{NP}^{<} \rightarrow \text{VP})$, $defeat :: e \rightarrow (e \rightarrow t)$)

(rejected :: $\text{NP} \rightarrow (\text{NP}^{<} \rightarrow \text{VP})$, $reject :: e \rightarrow (e \rightarrow t)$)

(thought :: $\text{CP} \rightarrow (\text{NP}^{<} \rightarrow \text{VP})$, $think :: t \rightarrow (e \rightarrow t)$)

($\epsilon :: \text{VP} \rightarrow \text{CP}$, $\lambda p.p :: t \rightarrow t$)

(that :: $\text{VP} \rightarrow \text{CP}$, $\lambda p.p :: t \rightarrow t$)

(is :: $\text{AdvP} \rightarrow (\text{NP}^{<} \rightarrow \text{VP})$, $\lambda P \lambda x.(P\ x) :: (e \rightarrow t) \rightarrow (e \rightarrow t)$)

(almighty :: AdvP, $almighty :: e \rightarrow t$)

(without :: $\text{NP} \rightarrow ((\text{NP} \rightarrow \text{VP})^{<} \rightarrow (\text{NP} \rightarrow \text{VP}))$,
$\lambda x \lambda P \lambda y.((P\ y) \wedge ((without\ x)\ y)) :: e \rightarrow ((e \rightarrow t) \rightarrow (e \rightarrow t)))$

- (Ishtar thought that Enkidu wept :: CP, ((*think* (*weep enkidu*)) *ishtar*) :: t)

The fragment could not cover quantificational noun phrases or questions. Since the operation **merge** works completely local and knows no means for using one expression at two different points in the derivation, it cannot handle displacement and scope construal. The task of the next two chapters is to extend the available operations so we can account for discontinuous dependencies. The most important feature of these extensions will be that syntax and semantics go separate ways. I propose that the dependency between a displaced expression and the corresponding gap is a purely syntactic one and that the dependency between an operator and the variable it binds is a purely semantic one. As a consequence, displacement will be handled by a syntactic procedure and scope construal will be a matter of a semantic procedure. Chapter 4 provides the syntactic procedure for displacement, and Chapter 5 provides the semantic procedure for scope construal.

4

A syntactic procedure for displacement

This chapter will extend the core system of the previous chapter with a procedure for displacement. The goal is to be able to generate questions like Who do you think that Enkidu knows that Gilgamesh rejected. As mentioned in the first chapter, who stands in a local relationship with the verb rejected. However in the surface string it does not occur adjacent to the verb but in front position. So we need a mechanism that can both relate who to the verb and derive its displaced position.

Syntactic theories differ in how they meet the challenge posed by displacement. The syntactic mechanism that I am going to outline in this chapter is based on a proposal by Brosziewski [14], which combines characteristics of different approaches. It specifies syntactic derivations as compositional processes, without referring to phrase structures, similar to categorial grammars [79]. But unlike in categorial grammars, the displacement operation is not type-driven but feature-driven, as assumed by the Minimalist Program [20]. It dislocates an expression by passing information via a sequence of local steps, similar to how Generalized Phrase Structure Grammar [43] envisaged displacement.

The main advantage of Brosziewski's approach is that it allows us to disregard all information that do not play a role. We will work with rudimentary syntactic structures which encode only those information that are relevant for establishing displacement dependencies. We can thus talk about the mechanism behind these dependencies without having to get specific about other

processes or issues purely related to the theoretical framework.

Brosziewski's proposal is a formalization of generative grammars that is, in fact, very close to the tree-less version of Stabler's Minimalist Grammars [106]. It keeps only the information that are needed later, in particular expressions that are displaced. However, while tree-less Minimalist Grammars keep displaced expressions in a flat list, Brosziewski recursively builds pairs of expressions. This gives rise to some structure which those Minimalist Grammars miss and which will play a crucial role in deriving remnant movement in Section 4.5.

Throughout the present chapter, I will develop an extension of Brosziewski's proposal. It modifies his original definitions in favor of an even more rigorous formalization suitable for implementation, while keeping the spirit of the original. The main reason for departing from his formulations is to make it fit the core system of the previous chapter. Most importantly, Brosziewski's version lacks semantic representations, which I will add.

In formulating displacement operations, I will disregard everything that is not relevant, i.e. I will abstract away from all local dependencies like case assigment and agreement. The displacement procedure will operate on rudimentary syntactic structures which encode only those information that are needed. It will discard all other information and thereby keep syntactic structure to a minimum.

The chapter will proceed as follows. After explicating the operations that displace an expression, we will look at how these operations can account for the different patterns we find in multiple wh-questions across languages, and how they derive some of their main properties. Then we will generalize the employed operations to also handle remnant movement. Finally, we will put it into perspective by comparing it to other syntactic frameworks.

4.1 Features

In order to trigger and control displacement we need somewhat richer information than we have up to now. I follow Generative Grammars in assuming that displacement is driven by features and therefore enrich the typed strings which constituted the form dimension with an unordered list of features **Feat**. The definition of forms then reads like this:

Definition 4.

$$\textbf{Form} ::= \textbf{String} :: \textsf{Cat}\ [\textbf{Feat}]$$

Features can be thought of as representing some property of an expression that needs to be satisfied in a local configuration with a certain other expression. With respect to features, I follow Brosziewski in sticking to very simple

assumptions. More specifically, I assume that features come in two varieties: as *goal features* f and as *probe features* $\bullet f$ with a prefix that indicates that it matches with a corresponding goal feature.

Definition 5. *The set of syntactic features is given by:*

$$\textbf{Feat} ::= \textbf{Value} \mid \bullet\,\textbf{Value}$$
$$\textbf{Value} ::= wh \mid top$$

We will mainly look at features wh for question formation and only quickly mention topicalization features *top* in passing. But in principle features in **Value** can also comprise other features, like case features, agreement features, and so on.

Adding features to typed strings does not have any effect on the type of the string. This is because the features do not change with which other strings a string can combine. Instead they will drive the operations that preserve enough information to establish displacement dependencies. Before turning to these operations, let us look at some examples of forms:

(4.1) a. $\mathsf{Enkidu} :: \mathrm{NP}\,[\,]$
b. $\mathsf{who} :: \mathrm{NP}\,[wh]$
c. $\epsilon :: \mathrm{VP} \rightarrow \mathrm{CP}\,[\bullet wh]$

The form in (4.1a) constitutes an already familiar noun phrase. It is of category NP and its feature list is empty. The wh-expression in (4.1b) is also of category NP and moreover it has a syntactic feature wh that needs to be checked and that will be responsible for the displacement of the expression. The expression that will check the wh-feature is a complementizer as given in (4.1c). It subcateorizes for a VP and additionally contains a syntactic feature $\bullet wh$ which can be checked by a corresponding feature wh. Its phonological content is empty, which is represented by the empty string ϵ.

In the following, I will write features as superscripts. That is, I will write $\mathsf{who} :: \mathrm{NP}\,[wh]$ as $\mathsf{who}^{wh} :: \mathrm{NP}$, and so on.

In the previous chapter, expressions were form-meaning pairs which were ignorant about internal structure and derivational past. With features we now introduced information that needs to be remembered, for a feature needs to be accessible until it can be checked. We therefore need to allow some information and structure to be kept when building expressions. To this end, we distinguish between simple and complex expressions. Simple expressions will be form-meaning pairs, just like so far. Complex expressions, on the other hand, will be pairs of a form and an expression. (I use angled brackets for these pairs and round brackets for form-meaning pairs so that both can be distinguished better at first sight.)

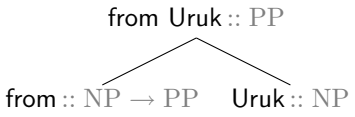
Definition 6. *Expressions are defined as follows:*

Expression	::=	(**Form**, **Meaning**)	*(simple expressions)*
	|	⟨**Form**, **Expression**⟩	*(complex expressions)*

The forms in complex expressions constitute subexpressions that still have features to check. If an expression still has features to check, it is called *active*, if it does not, it is called *inactive*. Inactive expressions can be forgotten because they do not play a role in the further derivation, but active expressions need to stay accessible in order to establish syntactic dependencies.

In the previous chapter, derivations did not build phrase structures. Recall that when simple expressions were combined, the result was another simple expression. No structure was built and information about the expressions that were combined as well as their structural configurations were forgotten. For example, when combining the two expressions from :: NP → PP and Uruk :: N (ignoring meanings for the moment), the resulting simple expression from Uruk :: PP contains all information relevant for the further derivation. Since there is not need to look into the structure (4.2), the root is all information that is kept.

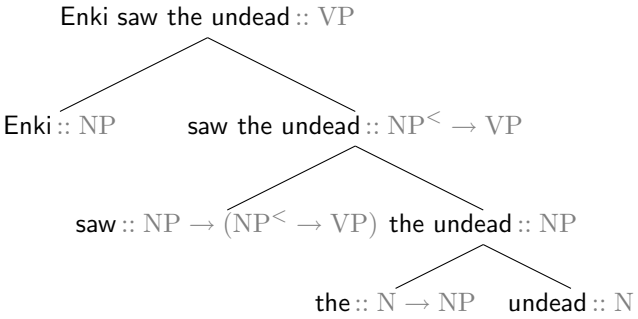
(4.2)



That is, although simple expressions can be the result of a long derivation, their internal structure is not accessible to syntactic operations. In this respect they behave like lexical items.

The same holds if a phrase like (4.3) is built. (Again, we leave out meanings for the moment.) The expressions from which it was built and their structural configurations will not play a role in the further derivation; they can therefore be discarded. Everything that syntactic operations need to care about is the root of the tree corresponding to the simple expression Enki saw the undead :: VP.

(4.3)



Disregarding all information about the past of a derivation in this way comes very close to removing the representational residue that causes Brody's conceptual problem with derivational approaches (see Brody [13]).

The situation is different, however, if expressions are involved that have to check features at a later stage of the derivation, like the wh-pronoun in Who did Enki see. The wh-expression ($\mathsf{who}^{wh} :: \mathrm{NP}, who$) (with who a placeholder for the meaning) carries a feature wh that requires the expression to enter a relation with an expression that carries a corresponding feature $\bullet wh$. When combining the verb with the wh-pronoun, we thus cannot build another simple expression forgetting about the subexpressions it was built from; who needs to be accessible until it can check its feature.

The next section will explicate operations to keep syntactic information that cannot be forgotten. The way to do this is to keep it accessible as the first element of a complex expression. For example, combining the verb see and the wh-expression who^{wh} will yield the following pair:

$$\langle \mathsf{who}^{wh}, (\mathsf{see} :: \mathrm{NP}^{<} \to \mathrm{VP}, \lambda y.((see\ who)\ y) :: e \to t) \rangle$$

It encodes that an expression of category $\mathrm{NP}^{<} \to \mathrm{VP}$ was built and that this had involved a wh-expression which still needs to check its wh-feature. The form of the wh-expression is kept as separate information, which is ignored when combining the complex expression above with other expressions, unless it can be resolved. For example, the derivation would proceed and build:

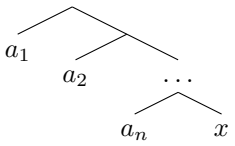
$$\langle \mathsf{who}^{wh}, (\mathsf{did\ Enkidu\ see} :: \mathrm{VP}, ((see\ who)\ enkidu) :: t) \rangle$$

When eventually a C-head carrying a feature $\bullet wh$ enters the derivation, who can check its feature wh. Then both expressions of the pair can be combined into a simple expression again, since there are no more features that need to be checked. The result is:

$$(\mathsf{who\ did\ Enkidu\ see} :: \mathrm{CP}, ((see\ who)\ enkidu) :: t)$$

More generally, a complex expression $\langle a_1, \langle a_2 \dots \langle a_n, x \rangle \rangle \rangle$ can be seen as having built a syntactic expression x together with a list (or stack) of forms

$a_1, a_2, \ldots, a_n$, that were extracted from x. They are carried along they can check their features. If we read the expression as a tree, it corresponds to:



Important is that the structure in complex expressions is the only structure syntactic operations will have access to. It is also import that in a complex expression $\langle a_1, \langle a_2 \ldots \langle a_n, x \rangle \rangle \rangle$ only x is associated with a meaning and thus determines the semantic behavior of the whole expression. All a_i are forms without meanings. This reflects that displacement is a purely syntactic issue with no semantic counterpart.

4.2 Displacement operations

Let us start with some notational conventions. In the following I will always use a, b as variables for forms, s for simple expressions and x, y, z as variables for arbitrary syntactic expressions (simple or complex). Furthermore, I will use f as a variable for single features, and F as a variable for feature lists. I will write a form a that has a set F of features as a^F. If F is empty, I will write it as $\emptyset$ or drop it and only write a if the features do not play a role at all (as is the case, for example, in the definition of **nucleus** below). If only one feature f plays a role, I will write a^f. This expresses that a has the feature f in its feature list but leaves open whether it also has other features or not. Sometimes I will denote the feature list of an arbitrary expression x as (**fs** x). The function **fs** can be defined as follows.

$$\begin{aligned} \textbf{fs}\ (a^F, E) &= F \\ \textbf{fs}\ \langle a, x \rangle &= \textbf{fs}\ x \end{aligned}$$

That is, the features of a complex expression are the features of its second element. Analogously, the type of a complex expression is set to be the type of its second element. Thus an expression $\langle a, x \rangle$ has exactly those syntactic properties that x has, except for the fact that it is complex. With respect to syntactic and semantic operations it will therefore largely behave like x. This reflects that a is preserved information that does not play a role until its features can be checked. Since x, on the other hand, is an expression that does play a role in the derivation, I introduce the notion of the *nucleus* of a complex expression. It is the second element of the deepest embedded pair, i.e. that simple expression that determines the properties of the whole complex

expression.

$$\textbf{nucleus}\ (a, E) = (a, E)$$
$$\textbf{nucleus}\ \langle a, x\rangle = \textbf{nucleus}\ x$$

For example, in a complex expression $\langle a_1, \langle a_2 \ldots \langle a_n, (b, E)\rangle\rangle\rangle$, the simple expression (b, E) is the nucleus. Furthermore, I will refer to the forms $a_1, \ldots, a_n$ as being *at the edge* of the complex expression. Defining the set of forms at the edge is straightforward:

$$\textbf{edge}\ (a, E) = \emptyset$$
$$\textbf{edge}\ \langle a, x\rangle = \{a\} \cup (\textbf{edge}\ x)$$

The edge of $\langle a_1, \langle a_2 \ldots \langle a_n, (b, E)\rangle\rangle\rangle$, for example, is $\{a_1, a_2, \ldots, a_n\}$.

The core of syntactic operations will be the function **merge** for combining two expressions into a third one. We already gave a definition for simple expressions on page 46 of the previous chapter. Now we have to extend this definition in order to also apply to complex expressions. The general idea is as sketched in the previous section: we want to discard information about structure, derivational history, and so on, unless it is really necessary to keep this information. Here is the definition of **merge**. It uses the function **split**, which will be defined below. We will come to it in a minute.

Definition 7.

$$\textbf{merge}\ x\ y = \begin{cases} \textbf{merge}\ x\ (\textbf{split}\ y) & \textit{if}\ \textbf{fs}\ y \neq \emptyset \\ \textit{see}\ \text{(M1–M3)} & \textit{otherwise} \end{cases}$$

(M1)	**merge**	(a^F, E_1)	(b, E_2)	$= ((a \oplus b)^F, (E_1\ E_2))$
(M2)	**merge**	$\langle a, x\rangle$	s	$= \langle a,\ \textbf{merge}\ x\ y\rangle$
(M3)	**merge**	x	$\langle a, y\rangle$	$= \langle a,\ \textbf{merge}\ x\ y\rangle$

Where $\oplus$ is defined as concatenating two strings of matching categories, with the order depending on the linearization diacritic:

$$a :: c_1 \rightarrow c_2 \oplus b :: c_1 = a +\!\!+ b :: c_2$$
$$a :: c_1^{<} \rightarrow c_2 \oplus b :: c_1 = b +\!\!+ a :: c_2$$

The former Definition 3.3 of **merge** on page 46 is contained in this new definition as the case (M1). (M1) combines two simple expressions into another simple expression by concatenating their strings, keeping the features of the head (i.e. the subcategorizing expression), and applying the denotation of the head to the denotation of the complement (i.e. the subcategorized expression).

The order of the concatenation is determined by the presence or absence of the linearization diacritic, as before.

As an example, consider merging the transitive verb meet with the noun phrase Gilgamesh. According to (M1), **merge** proceeds as follows:

$$\begin{aligned}\textbf{merge}\ &(\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet} :: e \to (e \to t))\\ &(\mathsf{Gilgamesh} :: \mathrm{NP}, \mathit{gilgamesh} :: e)\\ =\ &(\mathsf{meet\ Gilgamesh} :: \mathrm{NP}^{<} \to \mathrm{VP}, (\mathit{meet\ gilgamesh}) :: e \to t)\end{aligned}$$

(M1) is the base case of the recursive definition of **merge**: The other two cases, (M2) and (M3), will eventually boil down to it. (M2) and (M3) specify those cases in which it is necessary to keep information because there are features that still need to be checked. As an example, suppose merging meet not with the inactive NP Gilgamesh but with the active NP who^{wh}. It still has features to check, so it cannot simply be concatenated with the verb because that way, information about its features would be forgotten. Instead, we need to make sure that it cannot only satisfy the subcategorization requirements of the verb but is also able to check its wh-feature at some later point in the derivation. In order to let it make these two contributions, we invoke a mechanism that splits an expression's form into two forms, one of which is kept at the edge. It is called **split** and is defined as follows – where, again, a is a variable ranging over forms, F is a feature set, c ranges over categories and E over denotations, and ϵ denotes the empty string.

Definition 8.

$$\begin{aligned}\textbf{split}\ (a^F, E) :: c = \langle a^F, (\epsilon :: c, E)\rangle :: c\\ \textit{or}\ \langle \epsilon^F, (a :: c, E)\rangle :: c\end{aligned}$$

That is, splitting a simple expression (a^F, E) amounts to creating a complex expression. The nucleus inherits the type of a and its meaning component. The features of a, on the other hand, are associated with the form at the edge and will be carried along until they can be checked. The phonological content of a is either also associated with the form at the edge and carried along, or it is associated with the nucleus and thus stays in base position. The string that ends up at the edge is assumed to be of the general type String. The reason is that once the dependency is resolved at the top, the edge will be concatenated with the rest of the expression (see the definition of **remerge** below). Since string concatenation is a function of the general type String → (String → String), the category of the strings does not matter.

This approach of splitting an expression is close to the copy theory of movement introduced by Chomsky [19]. There, a displacement dependency amounts to a chain of copies of an expression. Usually exactly one of the copies is pronounced (typically the structurally highest one) and exactly one of them is interpreted (typically the structurally lowest one). Variations of this pattern account for variations in languages. Spell out of the lowest copy, for example, would instantiate covert movement in wh-in-situ languages (see e.g. Reintges et al. [91]). Now, what **split** does is like copying, with the only difference that it does not duplicate the parts of the expression but distributes it among the copies.

Note that **split** is defined only for simple expressions. We will generalize it to complex expressions in Section 4.5.

Now let us pick up our example of merging meet with who. As lexical for who, we assume the following:

$$(\mathsf{who}^{wh} :: \mathrm{NP}, \mathit{who} :: e)$$

The denotation $\mathit{who} :: e$ is only a place holder, because the semantic dimension of wh-expressions will be taken care of by a separate mechanism, which is subject of the next chapter.

We can now specify how the derivation proceeds, namely by splitting who (for conciseness, I leave out semantic type information):

$$\begin{aligned}
&\textbf{merge}\ (\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet})\ (\mathsf{who}^{wh} :: \mathrm{NP}, \mathit{who})\\
&= \textbf{merge}\ (\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet})\ \ (\textbf{split}\ (\mathsf{who}^{wh} :: \mathrm{NP}, \mathit{who}))\\
&= \textbf{merge}\ (\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet})\ \ \langle \mathsf{who}^{wh} :: \mathrm{String}, (\epsilon :: \mathrm{NP}, \mathit{who})\rangle
\end{aligned}$$

Before we can proceed, we need to know how to merge complex expressions. This is what (M2) and (M3) tell us. The idea is very straightforward. A complex expression behaves like its nucleus, so if a complex expression is merged, this is because its nucleus has certain properties; the expressions at the edge are only carried along. Thus the merge operation should affect the nucleus and ignore the expressions at the edge. This is exactly what (M2) and (M3) do: they pass s (or x, respectively) to the nucleus. So, merging complex expressions amounts to merging their nuclei, while the edges are carried along further.

Our derivation would proceed as follows (note that it turns out important that the category of who was associated with the nucleus):

$$\begin{aligned}
&\textbf{merge}\ (\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet})\ \langle \mathsf{who}^{wh} :: \mathrm{String}, (\epsilon :: \mathrm{NP}, \mathit{who})\rangle\\
&= \langle \mathsf{who}^{wh} :: \mathrm{String}, \textbf{merge}\ (\mathsf{meet} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \mathit{meet})\ (\epsilon :: \mathrm{NP}, \mathit{who})\rangle\\
&= \langle \mathsf{who}^{wh} :: \mathrm{String}, (\mathsf{meet} :: \mathrm{NP}^{<} \to \mathrm{VP}, (\mathit{meet}\ \mathit{who}))\rangle
\end{aligned}$$

That is, who^{wh} is kept at the edge, while ϵ serves to satisfy the verb's subcategorization requirements.

Until now we can handle the bottom and middle of a dependency. But we still miss an operation that resolves the dependency at the top. I will call it **remerge**. It applies as soon as we have a configuration $\langle a, x\rangle$, where a has a feature $\bullet f$ (or f) and the nucleus of x has the corresponding feature f (or $\bullet f$, respectively). I will give the definition only for one case; the other one is completely parallel. In particular, **remerge** does two things. First, it checks the matching features. The mechanism of feature checking will be subject of the following section; here it can be understood simply as deletion. And second, it concatenates a with the nucleus of x, unless a has more features to check (then it has to be kept at the edge). For multiple wh-displacement we will have to say more about what happens with other expressions at the edge of x, but this will also be subject of the next section. So the definition of **remerge** is still preliminary.

Definition 9 (preliminary).

$$\textbf{remerge}\ \langle a^f, x^{\bullet f}\rangle = \begin{cases} a + x & \textit{if } a \textit{ has no more features} \\ \langle a, x\rangle & \textit{otherwise} \end{cases}$$

Where $+$ is string concatenation with the form of the nucleus:

$$a + (b, E) = (a +\!\!+ b, E)$$
$$a + \langle b, x\rangle = \langle b, a + x\rangle$$

There is an important thing to note about **remerge**. The way the definition is stated, only simple expressions can be remerged. This will turn out to be important later, in Section 4.5, to obtain Freezing effects. Also note that the definition is not completely general, since it does not cover cases like $\langle a, \langle b^f, \langle c, d^{\bullet f}\rangle\rangle\rangle$. To capture also those, we can simply specify **remerge** to always apply to the outermost pair first and, if no features can be checked, to percolate through the pair until it reaches the nucleus:

$$\textbf{remerge}\ \langle b, x\rangle = \langle b, \textbf{remerge}\ x\rangle$$
$$\textbf{remerge}\ (a, E) = (a, E)$$

Furthermore, **remerge** is assumed to apply as soon as possible, following the idea that operations in general have to be performed as soon as possible. This has been stated in the form of the *Earliness Principle* by Pesetsky [86], it was later adopted by Chomsky [22] in his condition *Maximize Matching Effects*, and was also expressed in O'Grady's *Efficiency Requirement* [84]. In transformational terms, movement cannot skip a potential landing site.

Let us come back to our example derivation. Assume that in the meanwhile it proceeded by merging a subject NP and applying do-support (which we skip over here). The resulting VP is the following:

$$\langle \mathsf{who}^{wh}, (\mathsf{did\ Enkidu\ meet} :: \mathrm{VP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \rangle$$

Now it reached the point where a complementizer carrying a probe feature $\bullet wh$ can be merged. The lexical entry for such a complementizer is the same as in the example lexicon of last chapter (see Figure **??** on page 48) but with an additional feature list containing the probe feature $\bullet wh$:

$$(\epsilon^{\bullet wh} :: \mathrm{VP} \to \mathrm{CP}, \lambda p.p :: t \to t)$$

Merging it with the VP, we arrive at a configuration that will trigger **remerge**:

$$\begin{aligned}
&\textbf{merge}\ (\epsilon^{\bullet wh} :: \mathrm{VP} \to \mathrm{CP}, \lambda p.p) \\
&\qquad \langle \mathsf{who}^{wh}, (\mathsf{did\ Enkidu\ meet} :: \mathrm{VP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \rangle \\
&= \langle \mathsf{who}^{wh}, \textbf{merge}\ (\epsilon^{\bullet wh} :: \mathrm{VP} \to \mathrm{CP}, \lambda p.p) \\
&\qquad\qquad (\mathsf{did\ Enkidu\ meet} :: \mathrm{VP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \rangle \\
&= \langle \mathsf{who}^{wh}, (\mathsf{did\ Enkidu\ meet}^{\bullet wh} :: \mathrm{CP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \rangle
\end{aligned}$$

Now **remerge** applies, that is, the features wh and $\bullet wh$ are deleted and who is concatenated with the nucleus. The category and the denotation of the whole expression stays unaffected.

$$\begin{aligned}
&\textbf{remerge}\ \langle \mathsf{who}^{wh}, (\mathsf{did\ Enkidu\ meet}^{\bullet wh} :: \mathrm{CP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \rangle \\
&= \mathsf{who} + (\mathsf{did\ Enkidu\ meet} :: \mathrm{CP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu})) \\
&= (\mathsf{who\ did\ Enkidu\ meet} :: \mathrm{CP}, ((\mathit{meet}\ \mathit{who})\ \mathit{enkidu}))
\end{aligned}$$

We arrived at a form-meaning pair without any more features to check and with the wh-phrase displaced in the final string. We say that a derivation yielding such a simple expression *converges*, for it builds an expression without unfinished business. The grammatical sentences of a language thus are all simple expressions of category CP that our grammar generates.

To summarize this section, whenever we merge two expressions x and y, the properties of x remain, while the properties of y are forgotten. If y still has features to check, it is split in order to keep the relevant information accessible. This approach to displacement is one where expressions move in order to satisfy their own needs, blind of where this will happen and whether it is possible at all. Thus the mechanism that triggers displacement is divorced from any notion of landing site. In Chomskian terms, the approach is Greed-based and not Attract-based. A challenge all those approaches have to face is the variety observed in multiple wh-questions. It will be our next topic.

4.3 Multiple wh-questions and feature checking

Now that we have the core of the syntactic mechanism at hand, let us look at how we can capture the properties of wh-movement outlined in the first chapter. In doing so, we will explicate the feature checking mechanism and introduce two language-specific parameters that give rise to the variation we find across languages. Recall that with respect to multiple wh-questions languages follow one of the following three strategies (cf. Section 2.1):

- all wh-expressions stay in situ (e.g. Japanese)
- all wh-expressions are fronted (e.g. Bulgarian)
- exactly one wh-expression is fronted, the others stay in situ (e.g. English)

There are different ways to account for this variation. For example, Chomsky [20] introduced a distinction between strong and weak features. Strong features trigger movement, while weak features do not. In our implementation this would correspond to specifying that strong features choose to associate the phonological content of an expression with the edge when splitting it (i.e. to carry it along), while weak features choose to associate the phonological content with the nucleus (i.e. to leave it in base position). Although this would produce the difference between Japanese-like and Bulgarian-like languages, we could not give a straightforward treatment of English-like languages. In questions with more than one wh-expression, one of them (in most cases the structurally highest one) would have to carry a strong wh-feature, while all others would need to carry weak wh-features. But since we have no look-ahead capabilities, there is no way to decide which kind of feature a wh-expression should carry when it is merged.

For this reason, I will not follow Chomsky's proposal. Instead, I leave the feature system as it is and assume that the feature setup of wh-expressions is the same in all languages: all wh-expressions – whether Japanese, Bulgarian or English – carry a feature wh, that needs to be checked.

It follows that all these wh-expressions are split and carried along in the course of a derivation. So in all languages, the derivation of questions with two wh-expressions will yield the following configuration at the top of the dependency (assuming for now that the probe feature $\bullet wh$ is introduced at the CP level in all languages, and that E stands for the meaning of the already built expression):

(4.4) $\langle a^{wh}, \langle b^{wh}, (c^{\bullet wh} :: \mathrm{CP}, E)\rangle\rangle$

This configuration triggers **remerge**, so the features wh and $\bullet wh$ are checked and the forms at the edge are concatenated with the nucleus. There are two aspects here that need to be specified. First, which of the two wh-features at the edge are checked? The answer we give below will be: all of them. And second, which of the forms at the edge is concatenated with the nucleus? The

answer to that will differ across languages; I will also come back to it a bit later.

Let us first look at what we need to arrive at. We want to model the effect of all, no or exactly one wh-expression being fronted. An important difference between languages will therefore be whether the phonological content is associated with the edge of the split expression and ends up in top position, or with the nucleus and stays in bottom position. In wh-in situ languages like Japanese we want all phonological content to stay in bottom position, so we want to arrive at (4.5b). (Again, I do not specify the meaning of wh-expressions. Since it does not play a role here, I typeset it in grey font.)

(4.5) a. Dare-ga ringo-o tabeta no?
who-NOM apple-ACC ate Q

b. $\langle \epsilon^{wh}, \langle \epsilon^{wh}, (\text{darega ringoo tabeta no}^{\bullet wh}, ((\textit{eat apple})\ \textit{who}))\rangle\rangle$

In languages that front all wh-expressions, like Bulgarian, all phonological content should be associated with the top position, so the general configuration (4.4) should be instantiated like in (4.6b).

(4.6) a. Koj_1 kogo_2 $[\_\__1$ vižda $\_\__2]$?
who whom sees

b. $\langle \text{koj}^{wh}, \langle \text{kogo}^{wh}, (\text{vižda}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle\rangle$

In languages that front exactly one wh-expression, like English, one of the wh-expressions should associate the phonological content with the top position and the others should associate it with the bottom position, as in (4.7b).

(4.7) a. Who_1 $[\_\__1$ saw whom]?

b. $\langle \text{who}^{wh}, \langle \epsilon^{wh}, (\text{saw whom}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle\rangle$

Let us first consider languages like English, that front exactly one wh-expression. They pose a serious challenge for approaches like ours because some wh-expression needs to associate its phonological content with the top position and the others need to associate it with the bottom position. But since the syntactic mechanism has no look-ahead capabilities, there is no way to know which one is split in which way. All we can do is assume that it is optional whether the phonological content is carried along or not. So for English, **split** is defined as stated in the previous section:

$$\textbf{split}\ (a^F, E) :: c = \langle a^F, (\epsilon :: c, E)\rangle :: c$$
$$\text{or}\ \langle \epsilon^F, (a :: c, E)\rangle :: c$$

Consider again the simple example in (4.7a) above. Just like in all languages, both wh-expressions have to be split when they are merged, because both carry a feature *wh* that needs to be checked later. When they are split, it is now optional whether their phonological content is carried along at the edge or stays at the nucleus. So there are four possible expressions that can arise when building (4.7a). They are given in (4.8).

(4.8) a. $\langle \mathsf{who}^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{saw}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle$
b. $\langle \epsilon^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{who\ saw}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle$
c. $\langle \mathsf{who}^{wh}, \langle \epsilon^{wh}, (\mathsf{saw\ whom}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle$
d. $\langle \epsilon^{wh}, \langle \epsilon^{wh}, (\mathsf{who\ saw\ whom}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle$

The respective order of who^{wh} and whom^{wh} is due to how (M2) and (M3) distribute pairs over each other. To see this, suppose saw was already merged with whom^{wh}, resulting in $\langle \mathsf{whom}^{wh}, \mathsf{saw}\rangle$. Now this expression is merged with who^{wh}. Since the wh-expression has to be split, this amounts to merging two complex expressions:

$$\textbf{merge}\ \langle \mathsf{whom}^{wh}, (\mathsf{saw}, (\textit{see whom}))\rangle\ \langle \mathsf{who}^{wh}, (\epsilon, \textit{who})\rangle$$

According to the definition of **merge**, (M3) applies first. This is because (M2) requires the second argument to be a simple expression.[1] So we get:

$$\begin{aligned}
&\langle \mathsf{who}^{wh}, \textbf{merge}\ \langle \mathsf{whom}^{wh}, (\mathsf{saw}, (\textit{see whom}))\rangle\ (\epsilon, \textit{who})\rangle \\
&= \langle \mathsf{who}^{wh}, \langle \mathsf{whom}^{wh}, \textbf{merge}\ (\mathsf{saw}, (\textit{see whom}))\ (\epsilon, \textit{who})\rangle\rangle \\
&= \langle \mathsf{who}^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{saw}, ((\textit{see whom})\ \textit{who}))\rangle\rangle
\end{aligned}$$

Now let us look at the four possibilities in (4.8) in turn, to see how we end up with the right result. Let us start by observing that (4.8c) and (4.8d) will yield the string who saw whom, so these derivations should succeed, while (4.8a) and (4.8b) would yield who whom saw and whom who saw, respectively, so these derivations should not converge.

Let us first look at (4.8a), so let us assume that the phonological content of all wh-expressions is carried to the top:

$$\langle \mathsf{who}^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{saw}^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle$$

The wh-forms who^{wh} and whom^{wh} at the edge are of type String and the nucleus $\mathsf{saw}^{\bullet wh}$ is of type CP. Now **remerge** applies and according to its definition on page 67, it deletes the wh-feature of the outermost wh-form and concatenates it with the rest of the expression. (Recall that we assumed **remerge** to first apply to the whole pair and only when that fails to apply to the embedded pair.)

Now let us refine the feature checking process. It will rely on the following two assumptions:

[1] This is what introduces the order between (M2) and (M3). The opposite order would result if the first argument, x, in (M3) would be required to be simple. If neither the second argument in (M2) nor the first argument in (M3) were required to be simple, both (M2) and (M3) would apply when merging two complex expressions. There seems to be no inherent reason to choose one over the other, but I regard it as most natural and warranted to decide for the order preserving option.

(FC1) Feature checking checks all occurrences of a certain feature at the edge, as well as the corresponding feature on the nucleus.

(FC2) When a feature is checked, it is deleted.

The first assumption, (FC1), expresses that feature checking applies blindly, i.e. does not only apply to the features that trigger the checking but also to all other features of the same kind. The second assumption, (FC2), says what to do with checked features. Deleting them is an obvious choice here since we do not want to keep information that is not needed anymore. And features are not needed anymore because they fulfilled their role once they are checked, regardless of whether they receive an interpretation at the interfaces or not.

Let us turn back to the first case (4.8a) of our English example. Applying **remerge** deletes all *wh*-features (according to (FC1) and (FC2)) and concatenates the outermost wh-expression (who) with the nucleus of the expression (saw):

$$\begin{aligned}
&\textbf{remerge}\ \langle \mathsf{who}^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{saw}\ ^{\bullet wh}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle \\
&= \mathsf{who} + \langle \mathsf{whom}, (\mathsf{saw}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle \\
&= \langle \mathsf{whom}, (\mathsf{who} +\!\!+ \mathsf{saw}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle \\
&= \langle \mathsf{whom}, (\mathsf{who\ saw}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle
\end{aligned}$$

The result is a complex expression, so the derivation did not converge yet. In fact, it is stuck. The wh-expression whom is still at the edge but it does not have any features to check. It will therefore never be able to take part in a feature checking configuration, so it will never get a chance to be concatenated with the nucleus. This means whom will always stay at the edge and the derivation is never going to converge, no matter what other operations apply.

The situation with (4.8b) is exactly parallel. Also there, the wh-feature of whom is checked but whom cannot be concatenated. It will stay at the edge and prevent the derivation from converging.

Now, what about (4.8c) and (4.8d)? Just like in the other two cases, applying **remerge** to (4.8c) deletes the wh-features and concatenates who with the nucleus, resulting in:

$$\begin{aligned}
&\textbf{remerge}\ \langle \mathsf{who}^{wh}, \langle \epsilon^{wh}, (\mathsf{saw\ whom}\ ^{\bullet wh}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle \\
&= \mathsf{who} + \langle \epsilon, (\mathsf{saw\ whom}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle \\
&= \langle \epsilon, (\mathsf{who\ saw\ whom}, ((\mathit{see}\ \mathit{whom})\ \mathit{who})) \rangle
\end{aligned}$$

And analogously for (4.8d), which will result in exactly the same after applying **remerge**:

$$\begin{aligned}&\textbf{remerge}\ \langle \epsilon^{wh}, \langle \epsilon^{wh}, (\textsf{who saw whom}^{\bullet wh}, ((see\ whom)\ who))\rangle\\ &= \epsilon + \langle \epsilon, (\textsf{who saw whom}, ((see\ whom)\ who))\rangle\\ &= \langle \epsilon, (\textsf{who saw whom}, ((see\ whom)\ who))\rangle\end{aligned}$$

Again, the derivation did not converge because it yields a complex expression with a form left at the edge. But let us look closer. This time the form at the edge is $\epsilon^{\varnothing}$, i.e. the empty string with an empty feature list. Now, an empty string with an empty feature list will never have any effect on the derivation. So occurrences of them can be deleted without making any difference. In this sense, we can consider the edge to be non-empty actually. Formally this amounts to adding the following assumption:

$$\langle \epsilon^{\varnothing}, y\rangle = y$$

That is, both (4.8c) and (4.8d) end up with an empty edge which can be deleted. After deleting it, we arrive at the simple expression who saw whom. The derivation thus converges. (Note that this option is not available for (4.8a) and (4.8b), because the expression at the edge is not empty and we could not possibly delete it.)

So both succeeding derivations (4.8c) and (4.8d) have the same result. However they arrive there in a sightly different way. Is there a difference between both? The answer is yes. Their results do indeed differ in the case of questions with only one wh-expression. In this case, the derivation corresponding to (4.8c), where the phonological content of the highest wh-phrase is fronted, results in the following (again, ignoring do-support):

$$\langle \textsf{whom}^{wh}, (\textsf{did Enkidu see}^{\bullet wh}, ((see\ whom)\ enkidu))\rangle$$

After applying **remerge**, we get Whom did Enkidu see. Whereas the derivation according to (4.8d), where all phonological content stays in situ, results in the following expression:

$$\langle \epsilon^{wh}, (\textsf{Enkidu saw whom}^{\bullet wh}, ((see\ whom)\ enkidu))\rangle$$

After applying **remerge**, this yields the echo question Enkidu saw whom. That is, for the syntactic mechanism outlined here, echo question are constructed like any other questions (at least syntactically). This will make an interesting prediction a bit later.

Next, consider the case of multiple fronting languages like Bulgarian. Recall the simple example from above, repeated here as (4.9).

(4.9) a. Koj_1 kogo_2 [$\_\__1$ $\mathsf{vižda}$ $\_\__2$]?
who whom sees

b. $\langle \mathsf{koj}^{wh}, \langle \mathsf{kogo}^{wh}, (\mathsf{vižda}\ ^{\bullet wh}, ((see\ whom)\ who)) \rangle \rangle$

In order to arrive at (4.9a), where all wh-expressions are fronted, I assume that for those languages **split** is defined by always associating the phonological content with the edge of the split expression. This amounts to discarding the optionality in the definition of **split**, i.e. to specify it as follows:

$$\textbf{split}\ (a^F, E) :: c = \langle a^F, (\epsilon :: c, E) \rangle :: c$$

This way, the phonological content of every displaced expression will be carried along to the top of the dependency. Thus in our example, the configuration at the top will be the desired (4.9b):

$$\langle \mathsf{koj}^{wh}, \langle \mathsf{kogo}^{wh}, (\mathsf{vižda}\ ^{\bullet wh}, ((see\ whom)\ who)) \rangle \rangle$$

Applying **remerge** would proceed like in English:

(4.10) $\textbf{remerge}\ \langle \mathsf{koj}^{wh}, \langle \mathsf{kogo}^{wh}, (\mathsf{vižda}\ ^{\bullet wh}, ((see\ whom)\ who)) \rangle \rangle$
$= \mathsf{koj} + \langle \mathsf{kogo}, (\ \mathsf{vižda}, ((\ see\ \ whom)\ \ who)) \rangle$
$= \langle \mathsf{kogo}, (\mathsf{koj} +\!\!+\ \mathsf{vižda}, ((\ see\ \ whom)\ \ who)) \rangle$
$= \langle \mathsf{kogo}, (\ \mathsf{koj\ vižda}, ((\ see\ \ whom)\ \ who)) \rangle$

And like in English, we would be stuck with a non-empty form at the edge, that prevents the derivation from converging. This is clearly not what we want. We rather want to allow also kogo to be concatenated with the nucleus once its wh-feature is checked. That is, the difference between Bulgarian and English seems to be the following: In English only the outermost form that triggers **remerge** can be concatenated, while in Bulgarian this is possible also for all other forms that check the relevant feature. This can be thought of as the difference between allowing multiple specifiers and allowing only uniquely filled specifiers. If only one specifier is possible, only one wh-expression can be fronted. However, if multiple specifiers are possible, all wh-expressions involved in a feature checking relation can be fronted. That Bulgarian among other languages allows multiple specifiers was proposed, e.g., by Richards [94] (based on a proposal by Koizumi [66]). That English allows only one specifier is dicussed, e.g., in Bošković [121].

For languages that allow multiple specifiers, the definition of **remerge** has to be extended slightly to also concatenate other forms at the edge that carry a matching feature.

Definition 10 (final).

$$\textbf{remerge}\ \langle a^f, x^{\bullet f} \rangle = \begin{cases} a + \overline{x} & \textit{if } a \textit{ has no more features} \\ \langle a, \overline{x} \rangle & \textit{otherwise} \end{cases}$$

Where + is string concatenation with the form of the nucleus:

$$a + (b, E) = (a +\!\!+ b, E)$$
$$a + \langle b, x\rangle = \langle b, a + x\rangle$$

And where $\overline{x}$ is defined as follows:

$$\overline{\langle b^f, y\rangle} = \begin{cases} b + \overline{y} & \textit{if multiple specifiers are allowed} \\ & \textit{and } b \textit{ has no more features to check} \\ \langle b, \overline{y}\rangle & \textit{otherwise} \end{cases}$$
$$\overline{\langle b, y\rangle} = \langle b, \overline{y}\rangle$$
$$\overline{(b^{\bullet f}, E)} = (b, E)$$

That is, for languages that allow only unique specifiers, $\overline{x}$ is like x except that all occurrences of the feature f at the edge are deleted. For languages that allow multiple specifiers, on the other hand, all forms at the edge of x that carry the feature f are also concatenated with the form of the nucleus (unless they have more features to check).

The Bulgarian derivation from above would thus not proceed like in (4.10) but as follows:

(4.11) **remerge** $\langle \mathsf{koj}^{wh}, \langle \mathsf{kogo}^{wh}, (\mathsf{vižda}\ ^{\bullet wh}, ((\textit{see whom})\ \textit{who}))\rangle\rangle$
$= \mathsf{koj} + \mathsf{kogo} + (\mathsf{vižda}, ((\textit{see whom})\ \textit{who}))$
$= (\mathsf{koj\ kogo\ vižda}, ((\textit{see whom})\ \textit{who}))$

It is interesting to note that the order of concatenation of bs in $\overline{x}$ in the definition of **remerge** is such that it reflects the order of the expressions in base position. Multiple wh-displacement is thus an order-preserving operation. In order to derive Minimal Compliance effects as we saw them on page 26 in Chapter 2, one could assume a re-ordering of the wh-expressions by means of a scrambling operation (Bulgarian indeed employs scrambling).

This treatment of multiple wh-questions makes non-trivial predictions. For example, recall that in English we had two ways to derive a single wh-question. One was building an expression like (4.12a), i.e. fronting the phonological content, and the other one was building an expression like (4.12b), i.e. leaving the phonological content in situ. The former is a usual wh-question while the latter constitutes an echo-question.

(4.12) a. $\langle \mathsf{whom}^{wh}, (\mathsf{did\ Enkidu\ see}\ ^{\bullet wh}, ((\textit{see whom})\ \textit{enkidu}))\rangle$
b. $\langle \epsilon^{wh}, (\mathsf{Enkidu\ saw\ whom}\ ^{\bullet wh}, ((\textit{see whom})\ \textit{enkidu}))\rangle$

Since in Bulgarian-like languages, **split** always associates the phonological content with the nucleus of the split expression, no wh-expression can be spelled

out in situ. The syntactic mechanism thus predicts that there are no echo-questions with in situ wh-expressions in Bulgarian-like languages. And this is indeed borne out. As Bošković [123] observes, the following questions are ungrammatical even when read as echo questions.

(4.13) *Serbo-Croatian* (Bošković [123])

?*	Ivan	kupuje	šta?
	Ivan	buys	what

(4.14) *Bulgarian* (ibid.)

?*	Ivan	e	kupil	kakvo?
	Ivan	is	bought	what

(4.15) *Russian* (ibid.)

?*	Ivan	kupil	čto?
	Ivan	bought	what

Let us finally also turn to languages like Japanese, where all wh-expressions stay in situ. The way to achieve this pattern is parallel to that of Bulgarian: by discarding the optionality of **split**. But it is done so in the opposite way. Instead of always associating the phonological content of a split expression with its edge, it is always associated with its nucleus. This corresponds to specifying **split** as follows:

$$\textbf{split}\ (a^F, E) :: c = \langle \epsilon^F, (a :: c, E)\rangle :: c$$

This way, the phonological content of all moving expressions stays in situ. The above example, here repeated as (4.16a), thereby yields the desired configuration (4.16b).

(4.16) a.

Dare-ga	ringo-o	tabeta	no?
who-NOM	apple-ACC	ate	Q

b. $\langle \epsilon^{wh}, \langle \epsilon^{wh}, (\text{darega ringoo tabeta no}^{\bullet wh}, ((\textit{eat apple})\ \textit{who}))\rangle\rangle$

Then **remerge** applies and all wh-features are deleted. Also, the outermost form at the edge is concatenated with the nucleus of the expression. The result is:

$$\langle \epsilon, (\text{darega ringoo tabetano}, ((\textit{eat apple})\ \textit{who}))\rangle$$

Now we can either assume that the empty expression at the edge is deleted, like in English, or that it is concatenated with the nucleus, like in Bulgarian. Both possibilities yield the simple string dare-ga ringo-o tabeta no of type CP.

This concludes the wh-patterns we wanted to cover. Let us briefly summarize. This section introduced two language-specific parameters. The first one is relevant at the bottom of the dependency. It specifies whether the phonological content of a split expression is associated with its edge or its nucleus.

This determines whether a wh-expression occurs fronted or in situ. The second one is relevant at the top of the dependency. It specifies whether only the outermost form that checks a feature can be concatenated or whether this is possible for all forms that check the relevant feature. This was interpreted in transformational terms as whether a language allows multiple or only uniquely filled specifiers.

English sets these parameters such that **split** optionally associates the phonological content either with the edge or with the nucleus. Furthermore, English allows only uniquely filled specifiers. Bulgarian **split** differs in always associating the phonological content with the edge. Furthermore, Bulgarian allows multiple specifiers. Japanese **split**, on the other hand, always associates the phonological content with the nucleus. Whether Japanese employs multiple specifiers or not does not matter for the construction of wh-questions. Setting the parameters in this way derives the three different wh-patterns we observe with these three languages: all wh-expressions occur fronted in Bulgarian, all wh-expressions occur in situ in Japanese, and exactly one wh-expression occurs fronted in English while all others occur in situ.

4.4 Intervention effects

Chapter 2 gave an overview of locality restrictions on displacement. Now we want to look at how to derive them from the operations we employed. We will first look at wh-islands and the absence of wh-island effects, and then turn to superiority effects.

4.4.1 Wh-islands

The main generalization behind wh-islands was that wh-expressions cannot be extracted from a wh-domain. This is captured by the syntactic mechanism carved out in the previous two sections in the same way it prevented the wrong pattern for multiple wh-questions in English. I will demonstrate this with the following example.

(4.17) * Whom$_1$ did Enkidu wonder [what Ishtar granted __$_1$]?

For simplicity's sake, I assume granted to be of the ditransitive syntactic type NP $\rightarrow$ (NP $\rightarrow$ (NP$^{<}$ $\rightarrow$ VP)). But in fact it is also be possible to assume a Larsonian VP shell structure (cf. Chomsky [17] and Larson [70]) or any other structure. What is important here is that the result of constructing the embedded CP what Ishtar granted is the following expression with two wh-forms at the edge:

$$\langle \mathsf{what}^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{Ishtar\ granted}^{\bullet wh}, (((\mathit{grant}\ \mathit{whom})\ \mathit{what})\ \mathit{ishtar})) \rangle$$

This configuration triggers **remerge** and, analogously to the derivations of the previous section, yields the following result:

$$\langle \textsf{whom}, (\textsf{what Ishtar granted}, (((\textit{grant whom})\ \textit{what})\ \textit{ishtar}))\rangle$$

As in the non-converging derivations of English multiple wh-questions, we end up with a non-empty element at the edge (whom, in this case) that has no more features to check and prevents the derivation from converging.

Note that this does not only hold for displacement triggered by a wh-feature. It is exactly the same for every kind of extraction. So it holds in general that there is no f-displacement out of an f-domain, with f being an arbitrary feature.

If on the other hand there are two expressions that are displaced due to two different features, these two extractions do not conflict. As an illustration, consider the following sentence, where the NP the Bull of Heaven is topicalized out of a wh-domain created by displacement of where.

(4.18) [The Bull of Heaven]$_1$, Gilgamesh wondered [where Ishtar got $\_\__1$].

To see how it works, consider the embedded CP where Ishtar got. It corresponds to the following expression with both extracting forms, where and the Bull of Heaven, at the edge (I will skip over the semantic dimension):

$$\langle \textsf{the Bull of Heaven}^{top}, \langle \textsf{where}^{wh}, (\textsf{Ishtar got}^{\bullet wh}, \ldots)\rangle\rangle$$

This is a configuration that triggers **remerge**, however not with the outermost expression at the edge, because it has a feature that is different from the *wh*-feature of the nucleus, but with where. The result is the following:

$$\begin{aligned}
&\textbf{remerge}\ \langle \textsf{the Bull of Heaven}^{top}, \langle \textsf{where}^{wh}, (\textsf{Ishtar got}^{\bullet wh}, \ldots)\rangle\rangle \\
&= \langle \textsf{the Bull of Heaven}^{top}, \textbf{remerge}\ \langle \textsf{where}^{wh}, (\textsf{Ishtar got}^{\bullet wh}, \ldots)\rangle\rangle \\
&= \langle \textsf{the Bull of Heaven}^{top}, (\textsf{where Ishtar got}, \ldots)\rangle
\end{aligned}$$

The form the Bull of Heaven is unaffected by this remerging process. It stays at the edge until it can check its topicalization feature.

Recall from Chapter 2 that there are phenomena that can obviate islands. One of them was D-linking. As we saw on page 24, a wh-expression can indeed be extracted from a wh-domain if it is D-linked, that is, somehow anchored in the context. Here are examples from Bulgarian and Swedish showing the same pattern: D-linked wh-phrases can escape wh-islands, non-D-linked ones cannot.

(4.19) *Bulgarian* (Bošković [125])

a. * Kakvo$_i$ se čudiš [koj znae koj prodava $\_\__i$]?
'What do you wonder who knows who sells?'

b. Koja$_i$ ot tezi knigi se čudiš [koj znae koj prodava $\_\_{}_i$]?
'Which of these books do you wonder who knows who sells?'

(4.20) *Swedish* (Maling [73] and Engdahl [34], cited from Bošković [124])

a. * Vad$_i$ frågade Jan [vem som skrev $\_\_{}_i$]?
what asked Jan who that wrote
'What did Jan ask who wrote?'

b. [Vilken film]$_i$ var det du gärna ville veta
which film was it you gladly wanted know.INF
[vem som hade regisserat $\_\_{}_i$]?
who that had directed
'Which film did you want to know who had directed?'

There is a way to account for these facts within our account, namely by assuming that D-linked wh-expressions do not carry a feature wh but a different feature for D-linkedness. Here is why. A wh-island configuration for us occurs with an expression $\langle a^{wh}, \langle b^{wh}, x^{\bullet wh}\rangle\rangle$. In such a configuration, both a and b check their wh-feature, and either both of them can be concatenated with x or the derivation does not converge. But it is not possible for one of them to move further and check its wh-feature somewhere else. If a D-linked wh-expression now has a different feature, say *DLink*, the configuration would be $\langle a^{wh}, \langle b^{DLink}, x^{\bullet wh}\rangle\rangle$. Then b is not part of the feature checking for wh anymore and can thus be extracted further without a problem (analogous to topicalized expressions that can extract from wh-domains in the same way).

In general, the only possibility for an expression to escape an island is to have a feature setup different from the island creating expression. This account of islands is very similar to the one in Stroik [111]. A problem that we face is that all features are treated equal, so all islands should be equally strong. Either an expression has the same feature as the island and is trapped, or it has different features and can escape. However, this is not exactly what we observe in natural languages. On page 23 in Chapter 2, we saw that topicalization islands seem to be stronger than wh-islands. Being able to account for contrasts like this would require a more fine-grained feature system à la Starke [108].

Looking at non-D-linked wh-expressions, we predict island sensitivity for all instances of wh-extraction we saw so far. This is because overt and covert displacement are all handled by the same operations. They differ only in where the phonological content ends up. A consequence of this uniformity is that all three wh-patterns we saw are expected to underlie the same restrictions, in particular to all be island sensitive. We already saw that this holds for languages like English and Bulgarian, where at least one wh-expression occurs fronted. And it is indeed also true for Japanese. Japanese shows sensitivity to wh-islands although all wh-expressions occur in situ, like in the following example.

(4.21) *Japanese*

* Kimi-wa [Taro-ga dare-o hometa kadooka] sitte-imasu ka?
you-TOP [Taro-NOM whom-ACC admired whether] know-POLITE Q
'Which person x is such that you know whether Taro admired x?'

This supports the claim that the same operations are involved like in languages that overtly front wh-expressions.

However, it does not hold in general, for Japanese in situ wh-expressions may obviate strong islands. An instance is the following example.

(4.22) *Japanese* (Tsai [117])

John-wa [[dare-o aisiteiru] onna-o] nagutta no
John-TOP who-ACC loves woman-ACC hit Q
'Who is the person x such that John hit the woman who loves x?'

This kind of island insensitive in situ wh-expression can also be observed in Chinese:

(4.23) *Mandarin Chinese*

Ni xiang-zhidao [wo weishenme gei Akiu shenme]
you wonder I why give Akiu what
'Which reason x is such that you wonder what I give to Akiu because of x?'

And yet another example is Ancash Quechua, a language which employs both fronted and in situ wh-phrases (cf. Cole & Hermon [24]). If a wh-expression occurs fronted, it is subject to wh-islands, see (4.24a). If it occurs in situ, on the other hand, it is not island sensitive, see (4.24b).

(4.24) *Ancash Quechua* (Cole & Hermon [24])

a. * Ima-ta-taq qam kuya-nki suwaq nuna-ta
what-ACC-Q you love-2PL steal man-ACC
'Which x is such that you love the man who stole x?'

b. Qam kuya-nki ima-ta suwaq nuna-ta?
you love-2PL what-ACC steal man-ACC
'You love the man who stole what?'

In general, languages know both island sensitive and island insensitive in situ wh-expressions. This suggests that there are in fact two in situ strategies: one which corresponds to the usual displacement operations, as we have seen it in this section, and is thus subject to island constraints, and one that does not correspond to displacement and is thus not subject to island constraints.

The questions that this usually raises is: How do wh-expressions that are not displaced take scope? The answer here is very easy: the same way all other operators take scope. As already advertised in the introduction, scope

will be established by means of a semantic mechanism that is independent of displacement. So once we know how to construct the scope of quantificational noun phrases, in situ wh-phrases pose no additional problem. We will turn to island insensitive wh-phrases in Chinese and Japanese in Section 5.4.3 of the next chapter.

4.4.2 Superiority

Recall the contrast in (4.25).

(4.25) a. Who_1 [$\_\__1$ saw whom]?

b. * Whom_1 [did who see $\_\__1$]?

We already came across this effect in Chapter 2. We ascribed it to the fact that displacement operations target the structurally higher wh-phrase. Now let us look at it from a slightly different point of view, that of intervention. In (4.25b), extraction of the object wh-expression whom crosses another wh-expression. This expression is said to *intervene*. In (4.25a), on the other hand, extraction of the wh-expression who does not cross an intervening wh-expression.

This intervention effect has first been formulated by Chomsky [18] as the Superiority Condition, stating that in a structure $x \ldots [\ldots z \ldots y \ldots]$, an operation cannot involve x and y if it could also apply to x and z and y is superior to z, that is if z intervenes. In his original formulation, Chomsky defined superiority in a very general way: an expression x is *superior* to an expression y if every expression of a major category (nouns, verbs, adjectives, and their projections) that dominates x also dominates y but not conversely. Nowadays, intervention is understood in terms of asymmetric c-command. For our considerations here, it actually does not matter which one we adopt.

Let us recall how the syntactic mechanism of this chapter accounted for the contrast between (4.25a) and (4.25b). The derivations of these sentences arrived at the **remerge** configurations given in (4.26a) and (4.26b), respectively.

(4.26) a. $\langle \mathsf{who}^{wh}, \langle \epsilon^{wh}, (\mathsf{saw\ whom}^{\bullet wh}, ((see\ whom)\ who)) \rangle \rangle$

b. $\langle \epsilon^{wh}, \langle \mathsf{whom}^{wh}, (\mathsf{did\ who\ see}^{\bullet wh}, ((see\ whom)\ who)) \rangle \rangle$

The ordering of the forms at the edge resulted from the way (M2) and (M3) distribute complex expression over each other, as we saw in Section 4.3 above. Applying **remerge** to (4.26a) resulted in (4.27a), whereas applying **remerge** to (4.26b) resulted in (4.27b).

(4.27) a. $\langle \epsilon, \mathsf{who\ saw\ whom} \rangle$

b. $\langle \mathsf{whom}, \mathsf{did\ who\ see} \rangle$

The crucial difference is that in (4.27a) the form that is left at the edge is empty. We assumed that it can be deleted and thereby arrived at the simple expression who saw whom. The form left at the edge in (4.27b), on the other

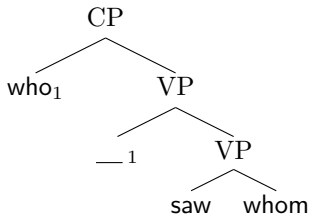
hand, is not empty, thus cannot be deleted. Instead it prevents the derivation from converging.

Note that the displacement operations in both cases proceed in exactly the same way and on their own are licit. Things go right or wrong only depending on whether the phonological content of the split expressions was carried along or left in situ. And in fact, the pattern for English in (4.26) can be extended to more than two wh-expressions. The relevant generalization is that only the phonological content of the outermost form at the edge of a pair can be non-empty. Every phonological content of deeper embedded forms will prevent the derivation from converging. That is, only derivations of expressions of the general form $\langle a^f, \langle \epsilon^f, \langle \epsilon^f \dots \langle \epsilon^f, x^{\bullet f} \rangle \rangle \rangle \rangle$ will converge after applying **remerge**. As soon as some phonological content intervenes between a and x, the derivation will not converge anymore.

In order to see to which extent this notion of intervention corresponds to intervention defined in terms of dominance or c-command, we need to look at whether the ordering of forms at the edge actually reflects such structural tree relations.

First, the order among forms at the edge indeed reflects the order of their base positions. Consider the example (4.25a) from above. A minimal tree according to standard assumptions could look like in (4.28a). The expression we would build before applying **remerge** has the structure given in (4.28b).

(4.28) a.



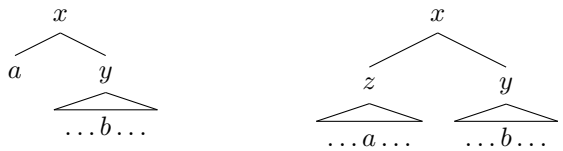
b. $\langle \textsf{who}, \langle \epsilon, (\textsf{saw whom}, ((\textit{saw whom})\ \textit{who})) \rangle \rangle$

At the edge of the expression in (4.28b), who is 'higher' than ϵ (corresponding to whom). This matches the hierarchy of the base position of who (indicated by the gap) and the position of whom in the tree.

In general, the order at the edge amounts to the order in which the expressions entered the derivation. In a pair $\langle a_1, \langle a_2 \dots \langle a_n, x \rangle \rangle \rangle$, a_i was merged after a_{i+1}. That is, the expression merged first will end up deepest in the pair (or: closest to the nucleus), and the outermost forms are those that were merged most recently. This is due to how (M2) and (M3) distribute complex expressions: The forms of a newly merged expression are stacked on top of already present forms.

Now can we also retrieve more complex structural relations from our pairs? The answer is no. With respect to c-command, for example, the following holds. Although it is the case that a structure in which a c-commands b corresponds to a pair where a is nested deeper in the pair than b, this correspondence does

not hold in the other direction. Consider the following structures, where a and b are assumed to have features that force them to move:



Although a c-commands b in the left tree whereas it does not in the right one, the derivation of both will build an expression of the form $\langle a, \langle b, \ldots \rangle \rangle$.

As a consequence, intervention effects in our approach cannot be bound to c-command but only to derivational order. Moreover, we predict that intervention without c-command is possible. And this is indeed the case, as observed by Heck & Müller [47]. It is illustrated by the following contrast.

(4.29) a. ?* Whom$_1$ did [the man [that defeated what]] admire $\_\__1$?
b. Who$_1$ $\_\__1$ admired [the man [that defeated what]]?
c. Whom$_1$ did [the man [that defeated Huwawa]] admire $\_\__1$?

In (4.29a), the dependency between whom and the corresponding gap spans a complex noun phrase that contains another wh-expression what. The sentence is ungrammatical, thus what seems to intervene, although it does not c-command the gap. If it does not intervene, like in (4.29b), the sentence is fine. Also if the intervening wh-expression what is replaced by a non-wh-expression, as in (4.29c), no intervention effect occurs. There are parallel cases for intervening wh-expressions in adverbial clauses and also for long-distance wh-movement, cf. Müller [80].

In our account, c-command does not play a role; only derivational order matters. The facts in (4.29) therefore follow in a way completely parallel to the other superiority cases. Let us briefly demonstrate this. First note that (4.29c) does not cause any problems because it contains only one wh-expression. The interesting cases are (4.29a) and (4.29b). Since all wh-expressions in English carry a wh-feature that needs to be checked, we can assume that the complex noun phrase the man that defeated what corresponds to the expression $\langle \epsilon^{wh}, (\text{the man that defeated what}, \ldots) \rangle$. The sentence in (4.29a) then corresponds to the expression (4.30a) and the sentence in (4.29b) corresponds to the expression (4.30b).

(4.30) a. $\langle \epsilon^{wh}, \langle \text{whom}^{wh}, (\text{did the man that defeated what admire}, \ldots) \rangle \rangle$
b. $\langle \text{who}^{wh}, \langle \epsilon^{wh}, (\text{admired the man that defeated what}, \ldots) \rangle \rangle$

The order at the edge reflects the derivational order: In (4.29a), the verb is first merged with whom and only afterwards with the complex noun phrase, therefore ϵ is stacked higher than whom. In (4.29b), on the other hand, the verb is first merged with the complex noun phrase and only afterwards with who, the order at the edge is therefore the opposite one. We can already see that

we get an intervention effect in the first but not in the second case. Applying **remerge** results in (4.31a) and (4.31b), respectively.

(4.31) a. $\langle \textsf{whom}, (\textsf{did the man that defeated what admire}, \ldots) \rangle$

b. $\langle \epsilon, (\textsf{who admired the man that defeated what}, \ldots) \rangle$

The expression in (4.31b) has an empty expression at the edge, which can be deleted, whereas the expression in (4.31a) has a non-empty expression at the edge, which prevents the derivation from converging.

To conclude, the syntactic mechanism devised in this chapter can capture intervention effects. It does so by relying on derivational order rather than structural notions. Not surprisingly, this also causes problems; we will look at them in Chapter 7. Now I want to turn to another case where the hierarchy induced by derivational order can be used to straightforwardly account for restrictions on displacement.

4.5 Extension: Remnant movement and Freezing

So far, **split** is defined only for simple expressions. In this section I want to generalize it to complex expressions. In order to see what this is useful for and why one has to be careful when doing so, we start by looking at possible and impossible displacement configurations, more specifically at remnant movement and Freezing configurations. For these explorations always keep in mind that complex expressions arise from splitting an expression that is displaced, and that a phrase x from which an element a was extracted is a pair of the form $\langle a, x \rangle$ (possibly with more forms at the edge).

That **split** is only defined for simple expressions implies that displacement is restricted to constituents without subextraction. For example, unproblematic are the displacement structures in (4.32), where only simple expressions are extracted.

(4.32) a. $a_2 \ldots a_1 \ldots [_x \ldots \_\__2 \ldots \_\__1 \ldots]$

b. $[_x \ldots a_1 \ldots \_\__1 \ldots]_2 \ldots \_\__2 \ldots$

In (4.32a), the constituent x corresponds to an expression of form $\langle a_1, \langle a_2, x \rangle \rangle$, something we are quite familiar with already. The case in (4.32b) amounts to extracting and remerging a while generating x. Then x itself is extracted from some larger constituent. By that time, a has been remerged already and x thus corresponds to a simple expression again.

Configurations like in (4.33), on the other hand, cannot be derived. In both cases, a is extracted from x while x is extracted itself. That is, when x is split, it corresponds to an expression of form $\langle a, x \rangle$, so **split** would have to apply to a complex expression.

(4.33) a. $[_x \ldots \_\__1 \ldots]_2 \ldots [\ldots a_1 \ldots [\ldots \_\__2 \ldots]]$

b. $a_2 \ldots [[_x \ldots \_\__2 \ldots]_1 \ldots \_\__1 \ldots]$

(4.33a) is a case of remnant movement, and (4.33b) is a Freezing configuration. What differs is the derivational order of the two displacements involved.

- *Remnant movement*
  First a is extracted from x, then (the rest of) x is displaced.

- *Freezing configuration*
  First x is displaced, then a is extracted from x.

Freezing configurations, i.e. extraction from a displaced phrase, is generally taken to be impossible in languages, whereas remnant movement is assumed to be possible. To illustrate this, consider the following examples from German. (4.34a) instantiates remnant movement: the NP das Buch scrambles out of the VP, then the remnant VP is topicalized. (4.34b) on the other hand shows a Freezing effect: first the NP ein Buch worüber is scrambled, and subsequently the PP worüber is extracted from that NP.

(4.34) a. $[_{\text{VP}}$ __$_1$ Gelesen$]_2$ hat $[_{\text{NP}}$ das Buch$]_1$ keiner __$_2$
read has the book.ACC no-one.NOM

b. $^*[_{\text{PP}}$Worüber$]_2$ hat $[_{\text{NP}}$ ein Buch __$_2\,]_1$ keiner __$_1$ gelesen?
about what has a book.ACC no-one.NOM read

Let us look at how these sentences would be generated with the syntactic mechanism of this chapter. First consider the Freezing configuration (4.34b). The NP corresponds to the following complex expression (ignoring the semantic dimension):

$$\langle \textsf{worüber}^{wh}, (\textsf{ein Buch} :: \text{NP}, \ldots) \rangle$$

When this NP is merged with the verb, it would have to be split in order to be displaced itself. But since it is a complex expression, **split** cannot apply. This seems desirable because (4.34b) is indeed excluded. But for the same reason also remnant movement is impossible. Consider (4.34a). The VP corresponds to the following expression (where I use Σ as the feature triggering scrambling):

$$\langle \textsf{das Buch}^{\Sigma}, (\textsf{gelesen} :: \text{VP}, \ldots) \rangle$$

This is a complex expression that cannot be split, although it would need to be split in order to be fronted.

Since we do not want to exclude the latter case, the definition of **split** has to be extended to complex expressions. An obvious way to do so would be to split the nucleus while keeping the edge:

$$\textbf{split}\ \langle a, x \rangle = \langle a, \textbf{split}\ x \rangle$$

This indeed allows for remnant movement. The derivation for (4.34a), that got stuck when having to split the expression $\langle \mathsf{das\ Buch}^{\Sigma}, (\mathsf{gelesen} :: \mathrm{VP}, \ldots)\rangle$, could now proceed. The result of applying **split** is:

$$\langle \mathsf{das\ Buch}^{\Sigma}, \langle \mathsf{gelesen}^{top}, (\epsilon :: \mathrm{VP}, \ldots)\rangle\rangle$$

The derivation proceeds without further complications, with the two forms at the edge being remerged when they can check their features.

But in the same way, the derivation of (4.34b) converges and Freezing effects are no longer obtained. The expression $\langle \mathsf{worüber}^{wh}, (\mathsf{ein\ Buch} :: \mathrm{NP}, \ldots)\rangle$ can now be split, resulting in:

$$\langle \mathsf{worüber}^{wh}, \langle \mathsf{ein\ Buch}^{\Sigma}, (\mathsf{gelesen}, \ldots)\rangle$$

Again, the derivation proceeds without further complications and the two forms at the edge are remerged when their features can be checked.

So, the derivations involving Freezing configurations proceed in exactly the same way derivations involving remnant movement do. The reason is the following: We decided to split a complex expression $\langle a, x\rangle$ such that the result is $\langle a, \langle x, \epsilon\rangle\rangle$. Above we observed that the difference between Freezing and remnant movement is the order in which a and x are extracted, most importantly in which order a and x are remerged. However, this is a difference we cannot capture. Keeping both a and x as separate elements at the edge does not give us a way to tell in which order the two will be remerged, and especially it does not give us a way to allow one order and disallow the other.

What to do about it? We need to distinguish the case of a being remerged before x (remnant movement) from the case of x being remerged first (Freezing). This is actually possible without much ado. The trick is to not split $\langle a, x\rangle$ such that a and x are kept as separate forms at the edge but instead such that $\langle a, x\rangle$ is kept as what it is: one constituent, a complex expression. That is, **split** should result in the expression $\langle\langle a, x\rangle, \epsilon\rangle$. Once we adapt **remerge** in order to reach a in this configuration, we are done. What does the job of allowing remerge of a before x (remnant movement) but not vice versa (Freezing) is the fact that **remerge** is defined only for simple forms. Here is why. If we first remerge a, this is unproblematic because it is a simple form. Then x remains at the edge; if it is a simple form as well, it can also be remerged without a problem. If we, however, tried to first remerge x, it still amounts to the complex form $\langle a, x\rangle$ for which **remerge** is not defined. This possibility is therefore blocked.

In a nutshell, we exploit the derivational difference between remnant movement and Freezing configurations by extracting $\langle a, x\rangle$ as one constituent and therefore causing a difference in the order of remerging a and x.

Let us look at how this works for our examples above. Recall that the complex expressions we needed to split were (4.35a) in the case of Freezing and (4.35b) for remnant movement.

(4.35) a. $\langle \mathsf{worüber}^{wh}, (\mathsf{ein\ Buch} :: \mathrm{NP}, \ldots)\rangle$
b. $\langle \mathsf{das\ Buch}^{\Sigma}, (\mathsf{gelesen} :: \mathrm{VP}, \ldots)\rangle$

Applying **split** in the way just described now yields (4.36a) and (4.36b), respectively.

(4.36) a. $\langle\langle \mathsf{worüber}^{wh}, \mathsf{ein\ Buch}^{\Sigma}\rangle, (\epsilon :: \mathrm{NP}, \ldots)\rangle$
b. $\langle\langle \mathsf{das\ Buch}^{\Sigma}, \mathsf{gelesen}^{top}\rangle, (\epsilon :: \mathrm{VP}, \ldots)\rangle$

In the first case, the scrambling feature can be checked first, thus the whole complex edge at the edge would need to be remerged. (This is something about the understanding of complex expressions and forms in this thesis: the nucleus is the core. It can have an edge or not. However, there can be no edge without a nucleus. So remerging the nucleus always involves the nucleus and its edge.) This remerging fails because remerge is not defined for complex forms.

Also in the second case, (4.36b), the scrambling feature can be checked first. This leads to das Buch being remerged. Since it is a simple form, this is unproblematic. The result is the expression $\langle \mathsf{gelesen}^{top}, (\mathsf{das\ Buch\ keiner} :: \mathrm{VP}, \ldots)\rangle$ (the form of the nucleus depends on the exact stage of the derivation, which does not play a role here). At some later point in the derivation, the topicalization feature can be checked. Since gelesen is also a simple form, it can be remerged as well without any problem. The remnant movement derivation hence is perfectly fine.

There is a nice consequence of this derivational approach. It concerns a restriction on remnant movement that Müller [82] formulated as the principle of *Unambiguous Domination*.

(4.37) *Unambiguous Domination*
In a structure $\ldots[_x \ldots y \ldots] \ldots$, x and y may not undergo the same kind of movement.

Within Müller's representational approach, it is necessary to define a local domain in which this condition applies, because otherwise sentences like (4.38) would wrongly be predicted to be out, for the embedded CP and the more inclusive NP undergo the same kind of movement.

(4.38) $[_{\mathrm{NP}}$ Wessen Frage $[_{\mathrm{CP}}$ was_1 du magst $\_\__1]]_2$ hat $\_\__2$ dich geärgert?
whose question what you like has you annoyed

For us, this problem does not arise. The displaced wh-expression was_1 is split, percolated and remerged upon constructing the embedded CP. And since it is remerged already when the CP is finished, the CP corresponds to a simple expression. Thus when the NP is constructed and extracted, it will be a simple expression itself. At no point of the derivation does a problematic configuration arise. In fact, the structure in (4.38) is as depicted in (4.32b) at the beginning of this section – a structure that we could already handle without being able to split complex expressions.

Now I want to extend our formal definitions so they can capture what we just sketched informally. First of all we need to allow pairs of forms at the edge, in other words, allow forms to be recursive.

$$\begin{aligned}\textbf{Form} ::= \textbf{String} &:: \textsf{Cat}\ [\textbf{Feat}] \\ \mid \langle &\textbf{Form}, \textbf{Form}\rangle\end{aligned}$$

The definition of **merge** stays like it is. But we have to extend the definition of **split**. For the simplest case it should look like this:

$$\textbf{split}\ \langle a, (b, E)\rangle = \langle\langle a, b\rangle, (\epsilon, E)\rangle$$

And we want to allow this to be recursive and also work if the edge is already a pair:

$$\textbf{split}\ \langle\langle a, b\rangle, (c, E)\rangle = \langle\langle\langle a, b\rangle, c\rangle, (\epsilon, E)\rangle$$

The general definition comprising these cases and also taking optionality into account is the following:

$$\begin{aligned}\textbf{split}\ \langle x, \langle \ldots, (a^F, E)\rangle\rangle &= \langle\langle x, a^F\rangle, \langle \ldots, (\epsilon, E)\rangle\rangle \\ \textit{or}\ &\langle\langle x, \epsilon^F\rangle, \langle \ldots, (a, E)\rangle\rangle\end{aligned}$$

Since we can now have nested forms at the edge, we need to extend what counts as being at the edge. We say that a form x is at the edge of a complex expression $\langle y, z\rangle$ if one of the following three conditions holds:

(i) x is equal to y

(ii) x is at the edge of y

(iii) x is at the edge of z

Clauses (i) and (iii) capture the cases we already encountered. For example (i) reaches what in $\langle \mathsf{what}, z\rangle$ and (iii) reaches who in $\langle \mathsf{what}, \langle \mathsf{who}, z\rangle\rangle$. Up to now (ii) would have been equivalent to (i), because y was always simple. Now (ii) captures the new case of reaching what in $\langle\langle \mathsf{what}, x\rangle, z\rangle$. The definition for the function **edge** thus now goes as follows:

$$\textbf{edge}\ (a, E) = \emptyset$$
$$\textbf{edge}\ \langle x, y\rangle = \{x\} \cup (\textbf{edge}\ x) \cup (\textbf{edge}\ y)$$

We also need to generalize the definition of **remerge**, in order to consider all new cases of edges, i.e. to be able to not only remerge the first element of a pair but also the edge of this element. It has to capture, for example, cases of the following forms:

- $\langle\langle a^f, x\rangle, y^{\bullet f}\rangle$
- $\langle\langle\langle a^f, x\rangle, z\rangle, y^{\bullet f}\rangle$
- $\langle\langle x, \langle a^f, z\rangle\rangle, y^{\bullet f}\rangle$

Let $x[a]$ denote an expression x in which a occurs, and let $x[\,]$ stand for x where a is removed, and $x[b]$ for x where a is replaced by b. Then we can give a general definition of **remerge** along the following lines:

$$\textbf{remerge}\ x^{\bullet f}[a^f] = \begin{cases} \overline{x}[\,] + a & \textit{if } a \textit{ has no more features} \\ \overline{x}[a] & \textit{otherwise} \end{cases}$$

*For the outermost $a \in$ (**edge** x), and where $\overline{x}$ is as before (see Definition 10 on page 67).*

What we keep, most importantly, is the restriction that only simple expressions can be remerged.

This concludes how to generalize the syntactic mechanism in order to incorporate remnant movement while still capturing Freezing effects. This generalization shows one of the possibilities the syntactic mechanism offers due to assuming a recursive structure with respect to displaced elements.

Since we will not need remnant movement in the further course of the thesis, we will consider this section a digression and for reasons of simplicity stick to the less general mechanism of the previous sections.

4.6 Summary

In the previous chapter we started from expressions being form-meaning pairs, where meaning is represented by terms of a lambda calculus and form is represented by typed strings. The present chapter was about carving out a mechanism operating on the form dimension that can account for displacement dependencies. To this end, typed strings were extended with an unordered list

Figure 4.1: Summary of the form dimension.

$$
\begin{aligned}
\textbf{Cat} &::= \text{NP} \mid \text{N} \mid \text{VP} \mid \text{CP} \mid \textbf{Cat}^{<} \mid \textbf{Cat} \rightarrow \textbf{Cat} \mid \text{String} \\
\textbf{Feat} &::= \textbf{Value} \mid \bullet\,\textbf{Value} \\
\textbf{Value} &::= \mathit{wh} \mid \mathit{top} \mid \Sigma \\
\\
\textbf{Form} &::= \textbf{String} :: \textbf{Cat}\ [\textbf{Feat}] \\
\textbf{Expression} &::= (\textbf{Form}, \textbf{Meaning}) \ \mid\ \langle \textbf{Form}, \textbf{Expression} \rangle
\end{aligned}
$$

of features. These features express properties that need to be satisfied upon presence of a matching feature. If a feature cannot be satisfied immediately, it needs to stay accessible while the derivation proceeds. This was taken care of by operations that split the form of an expression and carry it along until its features can be checked. A summary of the definition of expressions is given in Figure 4.1, and a summary of the syntactic operations **merge**, **split** and **remerge** is given in Figure 4.2.

We saw how the operations employed are able to derive the patterns of wh-displacement we find across languages (multiple fronting, single fronting, and wh-in-situ) without much further stipulation. We also saw how a range of locality conditions such as wh-islands and superiority already follow from the mechanism.

The main characteristics of the approach are the following:

- *Strict locality*
  Syntactic operations depend only on the properties of the expressions to which they apply.

- *Accessibility of active expressions*
  Accessible for syntactic operations are exactly those expressions that still have unchecked features. Once they entered the derivation, their form side is kept at the edge of a complex expression and remains accessible until all their features are checked.

- *Forgetfulness*
  No other information (the past of the derivation, structural configurations, and so on) is preserved.

- *Order preservation*
  The order in which forms appear at the edge corresponds to the order in which they entered the derivation. The order in which they are remerged in the case of multiple displacement reflects the order in which they would appear in base position.

Figure 4.2: Summary of the syntactic operations.

$$\textbf{merge}\ x\ y = \begin{cases} \textbf{merge}\ x\ (\textbf{split}\ y) & \text{if } \textbf{fs}\ y \neq \emptyset \\ \text{see (M1–M3)} & \text{otherwise} \end{cases}$$

$$\begin{array}{llll} \text{(M1)} & \textbf{merge} & (a^F, E_1) & (b, E_2) & = ((a \oplus b)^F, (E_1\ E_2)) \\ \text{(M2)} & \textbf{merge} & \langle a, x\rangle & s & = \langle a,\ \textbf{merge}\ x\ y\rangle \\ \text{(M3)} & \textbf{merge} & x & \langle a, y\rangle & = \langle a,\ \textbf{merge}\ x\ y\rangle \end{array}$$

Where $\oplus$ is defined as concatenating two strings of matching categories, with the order depending on the linearization diacritic:

$$\begin{aligned} a :: c_1 \rightarrow c_2 \oplus b :: c_1 &= a +\!\!+ b :: c_2 \\ a :: c_1^{<} \rightarrow c_2 \oplus b :: c &= b +\!\!+ a :: c_2 \end{aligned}$$

$$\textbf{split}\ (a^F, E) :: c = \begin{cases} \langle a^F, (\epsilon :: c, E)\rangle :: c & \text{in wh-fronting languages} \\ \langle \epsilon^F, (a :: c, E)\rangle :: c & \text{in wh-in-situ languages} \end{cases}$$

In mixed languages, this choice is optional:

$$\textbf{split}\ (a^F, E) :: c = \langle a^F, (\epsilon :: c, E)\rangle :: c \ \text{ or } \ \langle \epsilon^F, (a :: c, E)\rangle :: c$$

$$\begin{array}{lll} \textbf{remerge} & \langle a^f, x^{\bullet f}\rangle & = \begin{cases} a + x & \text{if } a \text{ has no more features} \\ \langle a, x\rangle & \text{otherwise} \end{cases} \\ \textbf{remerge} & \langle a, x\rangle & = \langle a, \textbf{remerge}\ x\rangle \\ \textbf{remerge} & (a, E) & = (a, E) \end{array}$$

Where $+$ is string concatenation with the form of the nucleus:

$$\begin{aligned} a + (b, E) &= (a +\!\!+ b, E) \\ a + \langle b, x\rangle &= \langle b, a + x\rangle \end{aligned}$$

And where $\overline{x}$ is defined as follows:

$$\overline{\langle b^f, y\rangle} = \begin{cases} b + \overline{y} & \text{if multiple specifiers are allowed} \\ & \text{and } b \text{ has no more features to check} \\ \langle b, \overline{y}\rangle & \text{otherwise} \end{cases}$$

$$\begin{aligned} \overline{\langle b, y\rangle} &= \langle b, \overline{y}\rangle \\ \overline{(b^{\bullet f}, E)} &= (b, E) \end{aligned}$$

In the following section I will point out in how far the present set-up differs from its predecessor (Brosziewski [14]) and how it compares to other approaches to the local modeling of non-local dependencies.

4.7 Comparison with other approaches

The radically local approach to displacement developed in this chapter is based on Brosziewski's *Derivational Theory* [14] and thus draws heavily on his work. Below, I will point out the differences in design, focus, and coverage. Since the ideas behind it are quite general, it shares characteristics with a range of other approaches, such as feature-enriched categorial grammars, Minimalist Grammars, phase theory, and Tree Adjoining Grammar. This section will give a brief comparison.

4.7.1 Brosziewski's Derivational Theory

My refined and extended version of Brosziewski's displacement mechanism keeps the core concepts and ideas of the original, however differs in two major points. The first difference concerns the treatment of islands. Brosziewski does in fact anticipate the possibility of the treatment I give in Section 4.4.1 above, but he rejects it as implausible and argues for an approach that shifts the responsibility 'onto semantics and a theory of selection' [14, p. 65], unfortunately without giving more than a sketch of it. I hope to have shown that an approach in terms of syntactic features can indeed be adopted and is not that implausible after all.

The second difference concerns the conception of **remerge**. In Brosziewski's version, **remerge** amounts to an application of **merge**. In general, for him **merge** comprises all possible combinations of expressions, be it induced by subcategorization, by matching features, or by neither (as in the case of adjunction). I chose to not keep this conception but rather introduce a clear distinction between **merge**, triggered by subcategorization, and **split** and **remerge**, triggered by syntactic features. This is a design issue more than an empirical one, but it has important consequences. My approach cleanly separates the operation for establishing local dependencies from the operations for establishing non-local dependencies. This plays a crucial role with respect to the general design of the syntax/semantics interface, since I take the former, but not the latter, to be paired with meaning assembly.

Additional to the basic mechanism for phrasal displacement, Brosziewski considers head movement. I discarded this direction because it is not relevant here. Nevertheless, it might be interesting to note that his approach to head movement could easily be adopted in my version as well.

Moreover, there is a number of issues treated in the present chapter that constitute a genuine extension of Brosziewski's work. One is the generalization of the **split** operation to complex expressions, which allowed us to capture

remnant movement. Another one is the detailed treatment of the patterns occurring with multiple wh-displacement and the according explication of the feature checking mechanism. And yet another issue, presumably the most important one, concerns the meaning dimension. Brosziewski does not consider semantic issues beyond some basic mention; his work is syntactic in nature. The present thesis extends his work with a semantic dimension and thereby develops it into a full-fledged syntax/semantics interface.

In the remainder of the section, let us consider how my extension of Brosziewski's approach to displacement compares with other approaches.

4.7.2 Movement-based approaches

One of the main points behind Brosziewski's proposal, that I adopted as an important motivation, is the idea of not remembering whole derivations but preserving only those information that are necessary to establish non-local dependencies. This general idea can also be found in some recent movement-based approaches to displacement like phase theory (see Chomsky [21],[22]). In phase theory, only small chunks of structure are built; they are sent to phonology and semantics for interpretation as soon as possible. An expression that still has features to check can stay accessible by means of movement to the phase edge (usually the specifier of a designated head), which is not sent to the interfaces immediately but is kept until the next phase is finished.

It is possible to read expressions as we encountered them in this chapter in phase theoretic terms: The a_i in a complex expression $\langle a_1, \langle a_2 \ldots \langle a_n, x\rangle\rangle\rangle$ can be seen as being at the edge of a phase, while everything contained in x was already sent to the interfaces.

This analogy, however, breaks down at second sight. The first important point with respect to which complex expressions differ from phases is that, although the extracted expressions a_i are percolated through every step of the derivation, keeping them accessible does not require additional movement steps triggered by edge features (cf. Chomsky [23]) or local optimization (cf. Heck & Müller [47]). It suffices that they are copied once, when they enter the derivation. After that, they simply stay at the edge of a complex expression, being pushed up higher and higher in the structure as more and more material is merged with the nucleus below them. The fact that no record of the forms at the edge are kept on the intervening nodes distinguishes my approach also from approaches relying on Slash feature percolation, for example GPSG [43], its successor HPSG [87], and proposals by Koster [67] and Neeleman & Koot [83].

With respect to structure expansion, my approach is actually very close to Tree Adjoining Grammar (see e.g. Joshi et al. [57], Kroch & Joshi [69], Kroch [68], and Frank [41]). In Tree Adjoining Grammar (TAG), all dependencies are established locally in elementary trees. These structures can then be expanded by inserting recursive structures between the extracted element and its corresponding gap – very much like in my approach additional material is merged

between the two parts of a split expression. However, TAG still employs a step of local movement in the initial elementary tree. My approach is much more radical in not employing any movement step at all, not even a local one.

The probably most important property of the forms that are kept at the edge is that they are kept in the order in which they were merged. This has an order preserving effect in the case of multiple wh-displacement: When the extracted expressions reach their target position, they are concatenated according to the order in which they appear at the edge, which reflects the order in which they would appear in base position. This offers a maximally simple account of order preservation, which is not possible without some stipulation in approaches relying on movement steps (including TAG). Moreover, it can be generalized to virtually all cases of multiple displacement that exhibit order preserving effects.

And finally, my approach sets itself apart in yet another way, the way of reducing the part of the structure that is accessible for syntactic operations. Instead of restricting accessibility to a certain domain (verb shells and clauses, or every phrase XP) like in phase theory, the model of derivations proposed in this chapter restricts the available expressions to those that carry yet unchecked features – independently of how deep in the structure (or how long ago) they were introduced.

This idea can also be found in the very recent approach of Stroik [111], where only those expressions are accessible for further operations that are incompatible with the currently active head (which largely coincides with having unchecked features). He employs a survival principle that copies those expressions into the numeration (or the workspace, if you want), from where they can then be imported back into the derivation. This exporting and re-importing applies at every step of the derivation, as long as the expression is compatible with the active head and can be incorporated into the structure. We can read our complex expressions $\langle a_1, \langle a_2 \ldots \langle a_n, x \rangle \rangle \rangle$ in Stroik's terms by assuming that the a_i correspond to those expressions that were exported due to still having unsatisfied properties. The difference is that we do not need a mechanism for copying expressions into the numeration and later re-importing them into the derivation; rather our complex expressions are the workspace themselves.

4.7.3 Feature-enriched categorial grammar and Minimalist Grammars

As mentioned above, one of the characteristics of Brosziewski's approach to displacement is the total lack of movement. This lack of movement also lies at the core of categorial grammars, a lexicalized grammar formalism based on directional type logic. The assembly of form is determined by the types that are assigned to lexical expressions and by general inference rules for these types. Instructions for the assembly of meaning, on the other hand, can simply be read off from syntactic derivations. The base grammar we developed in Chapter 3

is, in fact, very similar to the base logic of a categorial grammar, to the point of sharing the limitation of not being able to capture non-local dependencies. One possibility to overcome this limitation in categorial grammars is the introduction of structural reasoning controlled by unary modalities (corresponding to our features) that license the re-ordering of expressions and thereby determine which positions are accessible for semantic manipulation (e.g. for binding by an operator). These facilities allow to account for a range of cross-linguistic variation, which was comprehensively shown for wh-question formation by Vermaat [119].

The main point in which my account for displacement differs from the categorial approach (besides taking a generative and not a deductive perspective) is that categorial grammars inherently comprise a strict correspondence between syntax and semantics (due to the Curry-Howard correspondence, see e.g. Girard et al. [44]), whereas I opt for loosening that tie. Which approach will prove more successful in accounting for natural language phenomena at the syntax/semantics interface is a matter of future research. I will point to some directions in the last chapter.

Loosening the tie between syntax and semantics in the present chapter meant that displacement is a purely syntactic process that neither receives a semantic interpretation nor builds structures that could feed semantics. Note that extraction was not even encoded in the syntactic types: An expression $\langle a, x\rangle$ inherits its type from x. The fact that there is an element that still needs to check features is encoded only by keeping it at the edge. The edge thus plays a role very similar to a stack. In this respect, my approach converges with Stabler & Keenan's recent version of Minimalist Grammars (see Stabler & Keenan [107]). Minimalist Grammars are an algebraic formulation of the principles of Chomsky's Minimalist Program [20], developed by Stabler [106] and equiped with a semantic interpretation procedure by Kobele [65]. Stabler & Keenan's version interestingly dispenses with tree structures and instead resorts to lists of extracted expressions as the only information that is kept in the course of a derivation, very much like our edge of complex expressions. This makes it a very close relative of the approach developed in the present chapter, possibly they would even turn out to be largely equivalent. The difference, however, is that Stabler & Keenan employ flat lists of extracted expressions, while I introduced a recursive structure at the edge, allowing to extract expressions which itself contain extracted expressions. In Section 4.5 above, we saw that this additional structure can be exploited to capture remnant movement while at the same time obtaining Freezing effects.

4.8 Concluding remark: Why displacement?

The present chapter gives rise to an important question: If the procedure for displacement does not have a semantic effect and moreover does not build structures that are input to semantics, then what purpose does it serve?

One possible answer lies in the realm of information structure. Information structure encodes distinctions such as givenness and aboutness of information in a sentence. These notions are argued to be not directly encoded by grammar but marked by prosody and word order, for example. In case of word order this would mean that information structural notions are derived from configurations that, in turn, are created by displacement. Such a proposal can be found, e.g., in Slioussar [104]. On the basis of Russian data, she argues for an information structure model that encodes relative accessibility and salience based on syntactic configurations. Such a model could rely on a very general assumption like the following: An expression x is interpreted as more accessible or more salient than an expression y if x is higher in the syntactic hierarchy than y (e.g. moved over it). This picture fits very well with my approach to displacement because the displacement operations developed here change the relative order of expressions (whereas absolute information about structure and projection labels is lost).

Displacement therefore can have an interpretative effect without directly receiving a semantic interpretation.

5

A semantic procedure for scope construal

This chapter focuses on the meaning dimension of expressions. The goal is to equip the grammar developed so far with a procedure for establishing operator scope. The starting point is the assumption that such a procedure is not about interpreting displacement. Thus, the operations introduced in the previous chapter will not receive an interpretation. Instead, expressions are interpreted upon entering the derivation, that is when they are first merged. Scope construal will then take place in their meaning component alone. The means to do so will be delimited control. The main idea is to extend the lambda calculus employed in Chapter 3 with control operators that allow to establish non-local scope. Quantificational noun phrases and wh-expressions will be assigned a denotation that exploits this means.

This chapter follows the recent line of research that utilizes control flow mechanisms for natural language semantics, see, e.g., de Groote ([32],[33]), Barker [5], Shan [102], Barker & Shan [103], as well as Bernardi & Moortgat [9] and Kiselyov [63].

First, we will look at the standard way to treat quantificational noun phrases and the problems it poses for the syntax/semantics interface. Then we will become acquainted with the notions of evaluation order and delimited control and show how they can be used to establish non-local scope. This will include an account of different scope behaviors as well as scope ambiguities. The same mechanism can then be used for the scope of displaced and in situ wh-

expressions. In the end, we will look at predictions this makes with respect to scope islands and scopal interactions.

Since we take wh-expressions and quantificational noun phrases to denote operators, we start by adding operators to our calculus.

5.1 Operator scope

So far we only considered noun phrases that denote individuals, i.e. entities of type e. But for quantificational noun phrases like someone, every goddess and no human, we cannot use denotations of type e, for those expressions do not denote particular individuals. Instead, we will follow the Montagovian tradition of assuming them to denote *generalized quantifiers* of type $(e \to t) \to t$, i.e. functions that take a predicate as argument and state that this predicate is true for some human, for every goddess, or the like.

For representing the denotation of quantificational noun phrases, we use the well-known operators $\exists$ and $\forall$ from predicate logic. For wh-phrases like who and which king, on the other hand, we introduce a new operator that we write as $\mathfrak{W}$. These operators are added as second-order predicate constants to our language. We therefor define the following abbreviations:

- $\exists x.E$ is shorthand for $(\exists\ \lambda x.E)$.
- $\forall x.E$ is shorthand for $(\forall\ \lambda x.E)$.
- $\mathfrak{W}\, x.E$ is shorthand for $(\mathfrak{W}\ \lambda x.E)$.

All these operators are treated as variable binding operators. We say, for example, that in $\exists x.(\mathit{immortal}\ x)$, the variable x is bound by $\exists$. The role of these operators is to express quantificational force. For the familiar $\exists$ and $\forall$ this is existential and universal force, respectively. That is, an expression like $\exists x.(\mathit{immortal}\ x)$ is to be understood as stating that filling the position named by x with all possible instantiations will yield at least once a true statement. So $\exists$ applies to a first-order predicate and states that for some entity in the universe, this predicate is true. The expression $\forall x.(\mathit{immortal}\ x)$ states that filling the position named by x with all possible instantiations will always yield a true statement. So $\forall$ applies to a first-order predicate and states that it is true for all entities of the universe. Both operators differ with respect to the way in which the final truth-value depends on filling the argument position x, but the way they bind this variable is the same.

Now what about the interpretation of the operator $\mathfrak{W}$? While $\forall x.E$ and $\exists x.E$ are of type t, we want $\mathfrak{W}\, x.E$ to have a different type, for questions are not true or false. Rather, a formula of form $\mathfrak{W}\, x.E$ is intended to ask for all instantiations of x for which E is true. This complies with the most well-known approaches to the semantics of interrogatives. One of them, going back to Hamblin [46] and Karttunen [58], assumes that a question denotes the set of all possible (or true) answers. Another one goes back to Higginbotham

& May [52] and Groenendijk & Stokhof [45] and is based on the intuition that the meaning of a question is a partition of the logical space into those possibilities that can serve as an answer. Questions would then, for example, be equivalence classes of possible worlds. For our explorations, however, the actual denotation of a question does not matter. Since we are only interested in how the scope of a wh-operator is established, we will therefore not subscribe to a particular theory of question semantics but rather use an unanalyzed type q as the type of expressions of the form $\mathfrak{W}x.E$. You can imagine this type q to be an abbreviation for your favorite question type.

Now that we introduced the scope-taking operators, let us specify the notion of logical scope. It is actually analogous to the notion we had in Section 2.3. The scope of an operator is that part of an expression over which the operator can have a semantic effect. In $\forall x.E$ (and $\exists x.E$ and $\mathfrak{W}x.E$ analogously), the operator $\forall$ takes scope over E. For example, in the expression

$$(\mathit{angry\ enki}) \wedge \forall x.(\mathit{doomed}\ x),$$

the scope of the operator $\forall$ is $(\mathit{doomed}\ x)$. In this subexpression, it binds x.

Now let us put the operators to use and turn to meaning assignments for noun phrases that do not denote simple individuals. For example, we want to assign the meaning in (5.39b) to the sentence (5.39a).

(5.39) a. Gilgamesh rejected every goddess.

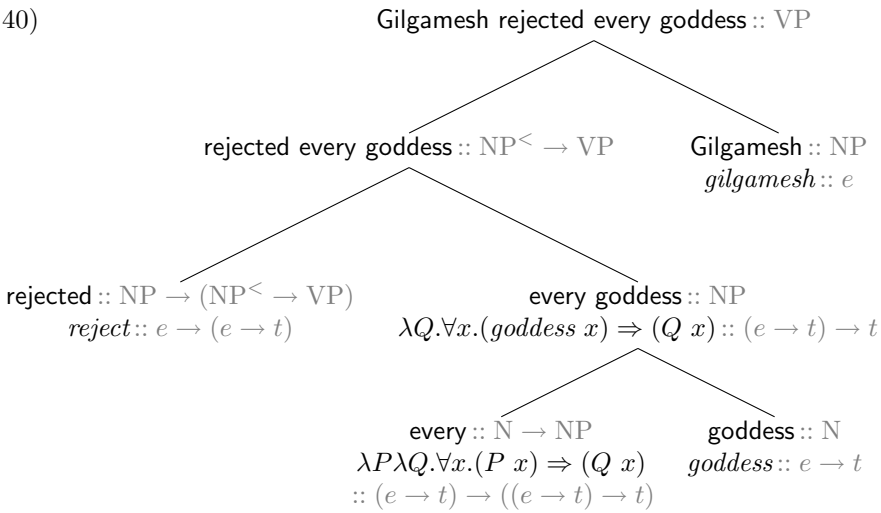
b. $\forall x.(\mathit{goddess}\ x) \Rightarrow ((\mathit{reject}\ x)\ \mathit{gilgamesh})$

Let us first look at the generalized quantifier denotations that determiners and quantificational noun phrases are usually associated with:

Form	**Meaning**
everyone :: NP	$\lambda Q.\forall x.(\mathit{person}\ x) \Rightarrow (Q\ x)$ :: $(e \to t) \to t$
someone :: NP	$\lambda Q.\exists x.(\mathit{person}\ x) \wedge (Q\ x)$ :: $(e \to t) \to t$
every :: N → NP	$\lambda P \lambda Q.\forall x.(P\ x) \Rightarrow (Q\ x)$ :: $(e \to t) \to ((e \to t) \to t)$
some :: N → NP	$\lambda P \lambda Q.\exists x.(P\ x) \wedge (Q\ x)$ :: $(e \to t) \to ((e \to t) \to t)$

The first problem we encounter is that these denotations do not satisfy the mapping $^\circ$ from syntactic to semantic types specified in Definition 2 on page 45. According to this mapping, a syntactic expression of type NP should be paired with a semantic expression of type e, and a syntactic expression of type N → NP should be paired with a semantic expression of type $(e \to t) \to t$. The mapping $^\circ$ ensured that if the syntactic combination of two expressions is well-typed, then their semantic combination is well-typed too. With the above denotations for determiners and quantificational noun phrases we give up $^\circ$ and instantly lose this well-typedness result. For example, building the derivation tree for (5.39a) succeeds syntactically but fails on the semantic side.

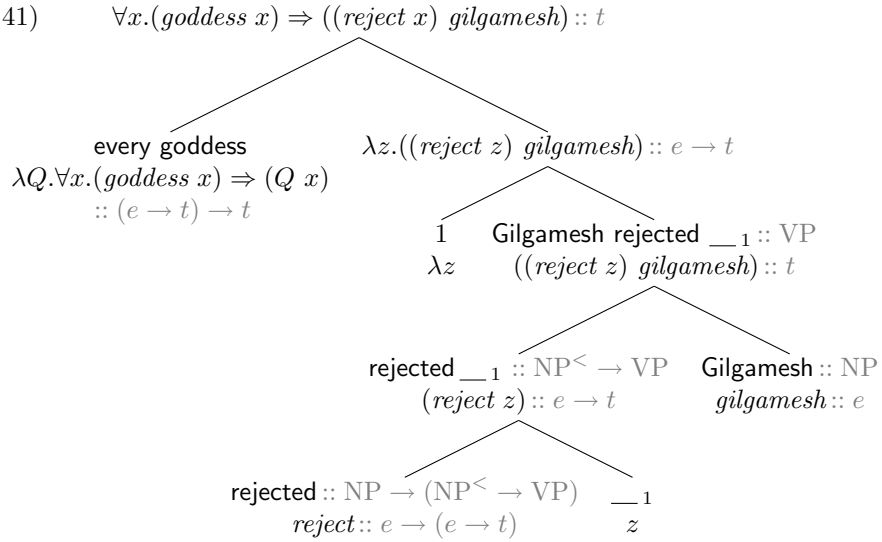
(5.40)



Building the semantic expression for rejected every goddess does not succeed because a type error occurs when merging the verb with its object. The latter is a quantificational noun phrase of type $(e \to t) \to t$ (the usual type of a generalized quantifier), while the verb is of type $e \to (e \to t)$, i.e. wants an argument of type e, not of type $(e \to t) \to t$. This clash is due to the two roles that quantificational noun phrases play in a derivation. First, they contribute to the argument structure of the verb they are merged with by filling one of its argument positions. That is, locally they behave like an individual of type e. And second, they take logical scope over the bigger constituent they occur in. That is, non-locally they behave like quantifiers of type $(e \to t) \to t$.

There are several ways to reconcile the local contribution of quantifictional noun phrases with their non-local scope assignment. We already mentioned some of them in Chapter 2. Another very straightforward one is the flexible types approach by Hendriks ([50],[51]), which offers type-shifting operations that can, for example, shift the type of the verb in order to make it fit its arguments. In our example Gilgamesh rejected every goddess, the verb rejected would be lifted to type $((e \to t) \to t) \to (e \to t)$ in order to take a generalized quantifiers as its first argument and an entity denoting noun phrase as its second argument. The Montagovian tradition usually goes a less flexible way refered to as 'generalization to the worst case': All noun phrases are uniformly assigned one type, and since this cannot be e, they are all assumed to be generalized quantifiers of type $(e \to t) \to t$. The most prominent approach in this vein involves LF movement of the quantifiers to their scope position (recall Section 2.5.3). For our example, this can be depicted as in (5.41). The quantificational noun phrase every goddess is extracted from its original position, leaving behind a gap that is interpreted as a variable, which is then abstracted over when the noun phrase is remerged at the top (the details do not need to concern us here). There it has the right type to take the verb phrase denotation as an argument.

(5.41)



The very same effect is achieved by Montague's rule of *Quantifying In* and the storage mechanisms we mentioned in Chapter 2. They all share the general idea that an operator expression takes scope by being applied to the sentence denotation where its original argument position is abstracted over. In the next section, we will look at this strategy from the perspective of evaluating expressions. This will pave the way for introducing the tools relevant in the rest of the chapter.

5.2 Delimited control

Recall the lambda calculus from Chapter 3 employed for the meaning dimension of expressions. The main rewriting rule of the operational semantics was beta-reduction. Its application was not restricted in any way, that is, when there is more than one possible beta-reduction that can be performed on a given expression, all of them are allowed. Consider the following example:

(5.42) $(\lambda p.((know\ p)\ enki)\ \ (\lambda P.\exists x.(P\ x)\ \ unicorn))$

There are two possibilities to reduce this expression. We can start by applying $\lambda P.\exists x.(P\ x)$ to *unicorn*, and then feed the result to $\lambda p.((know\ p)\ enki)$. We can also start by applying $\lambda p.((know\ p)\ enki)$ to the unreduced expression $(\lambda P.\exists x.(P\ x)\ unicorn)$. Which order we chose does not make a difference; both result in $((know\ \ \exists x.(unicorn\ x))\ enki)$.

We can be more explicit about which reduction is applied by talking about the contexts in which we apply a certain rule, so-called *evaluation contexts*. An evaluation context is a meta expression representing a family of expression with a special variable called *hole*, usually written as [ ]. The hole indicates the part

of an expression that may be subject to rule application. So the context can be seen as the focus of the rule that is going to be performed next. For example, in the expression (5.42) above, the reduction of $(\lambda P.\exists x.(P\ x)\ \mathit{unicorn})$ takes place in following context:

$$(\lambda p.((\mathit{know}\ p)\ \mathit{enki})\ [\ ])$$

Why is it useful to be able to express this? There are two things we can do with evaluation contexts once we can talk about them explicitly. First, we can restrict them as to allow rule applications only in some contexts but not in others. And second, we can manipulate them. We will need both in the next section, so let us look at them in turn.

Let us start with restricting rule applications to certain evaluation contexts. As an example, recall the rule for beta-reduction:

$$((\lambda x.E_1)\ E_2) \rhd E_1\{x \mapsto E_2\}$$

It describes the computation step of reducing an expression, independent of any context. That is, wherever the expression $((\lambda x.E_1)\ E_2)$ occurs in another expression, we can rewrite it as $E_1\{x \mapsto E_2\}$. Now consider we do not want to allow beta-reduction in all contexts. One way to restrict it is, for example, by specifying it as follows, where V stands for *values* of the language (constants, variables, and abstractions, but no applications):

$$((\lambda x.E)\ V) \rhd E\{x \mapsto V\}$$

This rule now only applies to applications where the argument is a value, i.e. an unreducible expression. So in our example,

$$(\lambda p.((\mathit{know}\ p)\ \mathit{enki})\ \ (\lambda P.\exists x.(P\ x)\ \ \mathit{unicorn})),$$

we can no longer substitute the unreduced expression $(\lambda P.\exists x.(P\ x)\ \ \mathit{unicorn})$ for p. This we can do only once we reduced it to the value $\exists x.(\mathit{unicorn}\ x)$. We therefore enforce an order of reductions. This particular one is called *call-by-value*, because functions can be applied only to values.

And there is another way to restrict evaluation contexts. Consider the following expression:

$$((\lambda P.P\ \mathit{wise})\ (\lambda x.x\ \mathit{enki}))$$

Although we specified call-by-value evaluation, we have two possibilities to proceed here: either first reduce $(\lambda P.P\ \ \mathit{wise})$, or first reduce $(\lambda x.x\ \ \mathit{enki})$. Let us say we want to fix the order of reductions such that they operate from left to right. We can do so by means of context rules, which have the general form:

$$\text{If } E \rhd E', \text{ then } C[E] \rhd C[E'].$$

This expresses that if E reduces to E', then we can rewrite E as E' in context C. For instance, specifying the context as application, we could have a context rule of the following form:

$$\text{If } E \rhd E', \text{ then } (V\ E) \rhd (V\ E').$$

It expresses that if an expression E reduces to E', then we may rewrite the former as the latter in the argument of an application if the applicand is already a value (i.e. a non-reducible expression). A more compact way of representing this rule is by means of the evaluation context $(V\ [\,])$. Thus, imagine we want to allow beta-reduction for arguments only if the applicand is non-reducible, then we would specify the rule for beta-reduction like this:

$$(V\ (\lambda x.E_1\ E_2)) \rhd (V\ E_1\{x \mapsto E_2\})$$

Since this is not very clear and since we would have to define a separate rule for all contexts we want to allow, we rather specify the admissible evaluation contexts C by means of a grammar like the following:

$$C\ ::=\ [\,]\ \mid\ (V\ C)\ \mid\ (C\ E)$$

It specifies that holes can occur as arguments of applications when the applicand is a value, and as applicands of applications regardless of the argument. Now we formulate the rule for beta-reduction with respect to such contexts C:

$$C[(\lambda x.E_1\ E_2)] \rhd C[E_1\{x \mapsto E_2\}]$$

This way we not only provide a computational rule that specifies how to rewrite expressions but we also fix the contexts in which this rule can be applied. This determines that $([\,]\ (\lambda x.x\ \textit{enki}))$ is a valid evaluation context for beta-reduction (because we can generate it with the grammar for C), therefore we can beta-reduce whatever expression occurs in the position indicated by the hole, whereas $((\lambda P.P\ \textit{wise})\ [\,]))$ is not a valid evaluation context for beta-reduction, because the hole occurs in the applicand of an application (i.e. we cannot generate the context with the grammar for C). So when reducing the expression $((\lambda P.P\ \textit{wise})\ (\lambda x.x\ \textit{enki}))$, we first have to reduce the left application, because the right one would take place in an illicit context. We succeeded in fixing a left-to-right order of evaluation.

Now let us turn to the possibility of manipulating evaluation contexts. To see why this is useful and how it can be done, recall the example in (5.40) above: Gilgamesh rejected every goddess. We encountered a type mismatch when trying to apply the verb denotation of type $e \to (e \to t)$ to the object noun phrase denotation of type $(e \to t) \to t$. The corresponding semantic expression is the following:

$$(\textit{reject}\quad \lambda P.\forall x.(\textit{goddess}\ x) \Rightarrow (P\ x))$$

The idea of Quantifying In amounts to transforming this expression into the following one:

$$(\lambda P.\forall x.(\textit{goddess}\ x) \Rightarrow (P\ x)\ \ \lambda z.(\textit{reject}\ z))$$

To see how to get there, let us first look at the original term from the perspective of the object noun phrase. The evaluation context of that noun phrase is

(*reject* [ ]). If we write it as a lambda expression, it corresponds to $\lambda z.(\textit{reject}\ z)$. The next step then is to apply the denotation of the object noun phrase to this reified evaluation context. So the gist of the Quantifying In strategy is to pull the noun phrase denotation out of its context and apply it to it. Note that this manipulates the evaluation context of the noun phrase because it no longer is the initial (*reject* [ ]) but a new one, namely ([ ] $\lambda z.(\textit{reject}\ z)$). In a way, we endow the noun phrase with control over its evaluation context, for it is now able to take scope over it.

In order to make this manipulation of the evaluation context explicit, two things need to be specified. First, we need to formulate the computational rule that allows an expression to take control over its context. And second, we need to delimit the context it can control. This is important for imagine an expression E (e.g. a generalized quantifier) occurs in the following expression:

$$((\textit{know}\ (\textit{die}\ E))\ \textit{gilgamesh})$$

There are several evaluation contexts that can be considered:

- (*die* [ ])
- (*know* (*die* [ ]))
- ((*know* (*die* [ ])) *gilgamesh*)

The next section will be dedicated to explicating the mechanism that enables expressions to take control over a delimited context, for short: the mechanism of *delimited control*, and to incorporate it into the meaning dimension of the grammar. The tool that theoretical computer science knows for accessing and manipulating evaluation contexts are *control operators*, such as `control` (cf. Feilleisen [39]) and `shift` (cf. Danvy & Filinski [30]). They are members of a family of delimited control operators and have, in fact, the same expressive power (see the interdefinability results of Shan [101] and Kiselyov [62]). They come with delimiters called `prompt` and `reset`, that delimit the context that is accessed.

In this chapter, I will employ `shift` as a meaning component of quantificational noun phrases and wh-phrases, in order to let them capture and modify (in particular take scope over) their evaluation contexts. This builds mainly on work by Barker and Shan; I will draw connections in Section 5.7 below. The reason to chose `shift` over `control` is its static scoping (as opposed to the dynamic scoping of `control`). We will see what this means in the next section.

Let us first have a look at how delimited control works in general. We will write `shift` as ξ and `reset` as $\langle\,\rangle$. More specifically, we will employ expressions of the form $\xi k.E$ and $\langle E\rangle$. The reduction rule for expressions $\xi k.E$ will specify an operation over the evaluation context: The context up to the nearest enclosing reset is captured, reified as a function and bound to k. To illustrate this, consider the following arithmetic expression:

$$6 + \langle 4 + \xi k.((k\ 2) \times (k\ 7))\rangle$$

The context that will be captured is the one enclosing the ξ-expression up to the reset, i.e. $4 + [\,]$. This is reified as a function, $\lambda x.4 + x$, and substituted for all occurrences of the variable k. The binder ξk is removed and the result is:

$$6 + ((\lambda x.4 + x)\ 2) \times ((\lambda x.4 + x)\ 7)$$

Applying beta-reduction, it evaluates to:

$$6 + (4 + 2) \times (4 + 7)$$

We can treat our type mismatch example from above in the same way. So consider again applying the denotation of the verb rejected to the generalized quantifier denotation of the object noun phrase every goddess, this time with shift and reset inserted:

$$\langle (\mathit{reject}\ \ \xi k.\forall x.(\mathit{goddess}\ x) \Rightarrow (k\ x)) \rangle$$

The context that shift captures is the evaluation context of the generalized quantifier, i.e. $(\mathit{reject}\ [\,])$. This is abstracted as a function and substituted for all occurrences of k. The result is $\langle \forall x.(\mathit{goddess}\ x) \Rightarrow (\lambda z.(\mathit{reject}\ z)\ x) \rangle$. Finally, we apply beta-reduction, remove the outermost reset, and arrive at the expression $\forall x.(\mathit{goddess}\ x) \Rightarrow (\mathit{reject}\ x)$. So, although the generalized quantifier started out as the argument of the predicate *reject*, it ended up taking scope over it. For that it is crucial that k occurred below $\forall$, i.e. inside its scope. This way the context is plugged into the scope of the operator.

5.3 Extending the meaning dimension

Now we make the mechanism of control transfer fully explicit. To this end, the lambda calculus for semantic expressions from Chapter 3 is extended with a control operator ξ (shift) and a corresponding delimiter $\langle\,\rangle$ (reset). In order to encode control transfers in the type of expressions, the type inventory is enriched with a type τ^{β}_{α}. It will be explained shortly. Furthermore, another basic type q for questions is added.

Definition 11. *The set of semantic types is given as follows:*

Type	::=	e	*(entities)*
	|	t	*(truth-values)*
	|	q	*(questions)*
	|	$\mathbf{Type} \rightarrow \mathbf{Type}$	*(functions)*
	|	$\mathbf{Type}^{\mathbf{Type}}_{\mathbf{Type}}$	*(impure expressions)*

We call e, t *atomic types*, and t, q *result types*. In the course of this chapter, we will use Greek letters $\tau, \alpha, \beta, \gamma, \delta$ as variables ranging over arbitrary types, and the variable r to range over result types.

In place of the typed expressions **Meaning** from Chapter 3, we now define typed expressions **E**, which additionally comprise logical constants like negation, conjunction and the second-order predicates $\exists, \mathfrak{W}$, as well as control operators shift and reset.

Definition 12. *Typed expressions* **E** *are defined as follows, where c is a variable ranging over the non-logical constants of the language.*

E	::=	$c :: \tau$	*(non-logical constants)*
	|	$x :: \tau$	*(variables)*
	|	$\exists :: (e \to t) \to t$	*(existential quantification)*
	|	$\mathfrak{W} :: (e \to t) \to q$	*(question operator)*
	|	$\neg :: t \to t$	*(negation)*
	|	$\wedge :: t \to (t \to t)$	*(conjunction)*
	|	$(\lambda x :: \tau'.\mathbf{E} :: \tau) :: \tau' \to \tau$	*(abstraction)*
	|	$(\mathbf{E} :: \tau' \to \tau\ \mathbf{E} :: \tau') :: \tau$	*(application)*
	|	$(\xi k :: \tau \to \alpha.\mathbf{E} :: \beta) :: \tau_\alpha^\beta$	*(shift)*
	|	$\langle \mathbf{E} :: \tau \rangle :: \tau$	*(reset)*

The non-logical constants c include predicate constants such as *gilgamesh* $:: e$, *king* $:: e \to t$, *brave* $:: e \to t$, *suffer* $:: e \to t$, *like* $:: e \to (e \to t)$, and so on. Logical constants comprise the operators $\exists$ and $\mathfrak{W}$, the connectives $\neg$ and $\wedge$, and the control operator ξ (more on it in a minute). With respect to types, we specified $\exists$ and $\mathfrak{W}$ to be second-order predicates, $\neg$ to be a unary predicate over type t and $\wedge$ to be a binary predicate over type t. As already mentioned in the beginning of the chapter, we will abbreviate $(\exists\ \lambda x.E)$ as $\exists x.E$ (and analogously for $\mathfrak{W}$). Furthermore, we write $(\neg\ E)$ as $\neg E$ and use the conjunction connective as an infix operator, i.e. we will not write $((\wedge\ E_1)\ E_2)$ but the familiar $E_1 \wedge E_2$. Furthermore, it is convenient to define implication and universal quantification in the usual way:

$$E_1 \Rightarrow E_2 =_{\text{def}} \neg(E_1 \wedge \neg E_2)$$
$$\forall x.E =_{\text{def}} \neg\exists x.\neg E$$

We will call expressions in **E** *impure* if they contain one or more shifts, and we will call them *pure* if they do not.

Now let us examine the shift a bit closer. It is a variable binding operator that binds a function variable of type $(\tau \to \alpha)$ (for which we use k, in order to distinguish it from other variables) in an expression of type β, thereby yielding an expression of type τ_α^β. This new type τ_α^β (taken from Shan [99]) expresses

that the expression carrying this type occurs in a position of type τ, thus locally it behaves like an expression of type τ. Moreover it induces a control transfer, which requires the captured context to be of type α (so that abstracting over the expression's original position forms a reified context of type $\tau \to \alpha$). Once this context is captured, an expression of type β is created.

For example, a generalized quantifier, standardly assumed to be of type $(e \to t) \to t$, will have the type e^t_t. This means that it locally behaves like an e and moreover captures a context of type t (e.g. a sentence), yielding an expression of type t again. Similarly for wh-phrases: They will be of type e^q_t. Thus they locally also behave like an e and capture a context of type t. The difference is that they do not return an expression of type t again but one of type q. That is, they transform a declarative sentence into an interrogative sentence. How exactly this works, we will see soon.

To complete the type system, we finally specify how impure types distribute over pure types. The two typing rules express that application can happen independent of whether the involved expressions are pure or impure; the encoding of the control effect is simply inherited.

$$\frac{E_1 :: \tau \to \tau' \qquad E_2 :: \tau^\beta_\alpha}{(E_1\ E_2) :: \tau'^\beta_\alpha}$$

$$\frac{E_1 :: (\tau \to \tau')^\beta_\alpha \qquad E_2 :: \tau}{(E_1\ E_2) :: \tau'^\beta_\alpha}$$

Next we define evaluation contexts. As informally explained in the previous section, they are expressions with a hole [ ]. Actually, for the fragment we will build, it suffices to consider applications.

Definition 13. *Evaluation contexts* **D** *and* **C** *are defined as follows:*

$$\mathbf{D} ::= [\,] \mid (\mathbf{E}\ \mathbf{D}) \mid (\mathbf{D}\ \mathbf{E})$$
$$\mathbf{C} ::= \mathbf{D} \mid \langle \mathbf{C} \rangle$$

This definition distinguishes two kinds of contexts: A context D does not contain any resets and is called *subcontext* or *delimited evaluation context*. A context C, on the other hand, is an arbitrary evaluation context in the sense that it can contain any number of resets. In the following, I will write $C[E]$ for the context C where the expression E was plugged in the hole.

The operational semantics for the calculus given in Definition 12 is now assumed to specify the usual eta- and beta-reduction as given on page 45 of Chapter 3. They are not repeated here; important is only that since they do not mention any evaluation contexts, they are not restricted, thus can be applied

in any context. Additionally, the operational semantics specifies the following reduction rules for the control operators, where E, E_1, E_2 are variables for arbitrary expressions **E**, and F is a variable for pure expressions (i.e. expressions not containing any shift).

$$C[\langle D[\xi k.E]\rangle] \rhd C[\langle E\{k \mapsto \lambda x.\langle D[x]\rangle\}\rangle]$$
$$\langle F\rangle \rhd F$$

Let us start with the second rule for reset. It states that a reset can be deleted if it surrounds a pure expression, i.e. an expression without any shifts. This is obvious because in that case the delimiter is not needed anymore. The rule does not specify an evaluation context, thus can be applied in all contexts. The first rule for shift, on the other hand, does mention an evaluation context. It may be applied in any context C that contains some subcontext enclosed by a reset. This is the context required for reduction of the ξ-expression as we described it informally in the last section. It proceeds as follows. The context up to the nearest enclosing reset, which is D, is captured, reified as a function, namely $\lambda x.D[x]$, and substituted for all occurrences of k in E.

The rewriting rule for shift thus provides expressions with access to their evaluation context. The expressions of our fragment that will be granted such access are noun phrase denotations. By assuming these denotations to be impure expressions, they will be able to take scope over the expression in which they occur. As a consequence, an expression does not need to be displaced in order to establish non-local scope; delimited control does the work for us.

There are a few important things to note about the reduction rule for shift. First, note that if $\xi k.E$ is of type τ_α^β, then the captured context D has to be of type α. Otherwise substitution of the reified context $\lambda x.\langle D[x]\rangle$ yields an expression that is not well-typed. Also note that the reduction rule rewrites the whole expression of some type ${\tau'}_\alpha^\beta$ into an expression of type β.

Second, the enclosing delimiter gets reinstalled instead of being deleted. Why is that necessary? Suppose we have another shift inside E, which is delimited by exactly the same reset. If we deleted the reset upon reducing one shift, we would thereby remove the delimiter of the other shift, which could then capture a much wider context than it is supposed to.

Third, the continuation $D[x]$ gets wrapped in an additional delimiter. This serves to prevent another shift inside D to capture a context spanning wider than D, for example also capturing E. This scoping is called *static*, as opposed to *dynamic scoping*, which differs in not introducing a new delimiter around $D[x]$. We will rely on this for preventing certain quantifiers from outscoping others.

Before turning to the treatment of quantificational noun phrases, let us reconsider the mapping $^\circ$ between syntactic and semantic types from Chapter

3 (defined in Definition 2 on page 45) in the light of the new semantic types that were introduced in this section. The main point is that $^\circ$ cannot relate anymore one syntactic type to exactly one semantic type. This is because semantic types encode information that are not present in the syntactic types, namely control effects in the meaning dimension, which have no counterpart in the form dimension. For example, in Chapter 2, the syntactic category NP was mapped to the semantic type e. Now it could also be mapped to type e^t_t, since the noun phrase denotation could induce a control transfer. Therefore, we now take $^\circ$ to be a mapping from categories to sets of semantic types. Its core idea, however, stays exactly the same.

Here is some new notation we need: I will write $[\tau]$ for the set of types τ with an arbitrary amount of control effects encoded. That is, formally, $[\tau]$ is the minimal set of types satisfying the following two conditions:

- $\tau \in [\tau]$
- If $\tau' \in [\tau]$, then also ${\tau'}^{\beta}_{\alpha} \in [\tau]$.

Now let us turn to the revised definition of the mapping $^\circ$.

Definition 14. *We assume a mapping $^\circ$ from syntactic types to sets of semantic types, such that:*

$$\begin{aligned} \mathrm{NP}^\circ &= [e] \\ \mathrm{N}^\circ &= [[e] \to [t]] \\ \mathrm{VP}^\circ &= [t] \\ \mathrm{CP}^\circ &= [r] \\ (c^<)^\circ &= [c^\circ] \\ (c_1 \to c_2)^\circ &= [c_1^\circ \to c_2^\circ] \end{aligned}$$

For example, the category NP is related to the set containing e, e^t_t, and so on, the category CP is related to the set of result types t and q, possibly with control effects (i.e. t^t_t, t^q_t, and so on), and the category N is related to the set containing $e \to t$, $(e \to t)^t_t$, $e \to t^q_t$, and the like.

Note that if we disregarded all control effects, the mapping $^\circ$ would be exactly like it was defined in Chapter 3. This reflects the way impure types τ^β_α are devised. The information encoded by ${}^\beta_\alpha$ determines the expression's non-local behaviour, i.e. its behavior with respect to the reduction rule for shift. The information encoded by τ, on the other hand, encodes the expression's local behavior, i.e. its behavior with respect to functional application. Thus for the operation of the base grammar (merging expressions) only τ matters.

5.4 Quantificational noun phrases

Now let us look at how to use the control operator ξ to establish non-local scope of quantificational noun phrases. We will start with strong quantifiers like every and the scope ambiguities they give rise to, and after that, turn to weak quantifiers, which seem to not allow ambiguities (cf. Section 2.3). I show how the delimited control mechanism can account for their behavior by adjusting the evaluation contexts in which the reduction rule for shift can apply. After that, I demonstrate how the same strategy can be used to capture exceptional wide scope as well.

5.4.1 Strong quantifiers

Let us start with a particular example, say the quantificational noun phrase everyone. It will be assigned the following denotation:

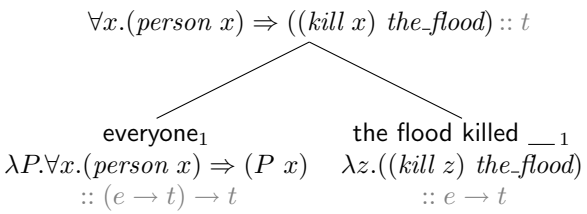
$$\xi k.\forall x.(\mathit{person}\ x) \Rightarrow (k\ x)$$

Recall the generalized quantifier denotation that is usually assumed:

$$\lambda P.\forall x.(\mathit{person}\ x) \Rightarrow (P\ x)$$

So we changed only a small yet important detail: replacing the binder λ by ξ. As a consequence, the operator $\forall$ is provided with access to its evaluation context, while the generalized quantifier denotation itself is in fact retained.

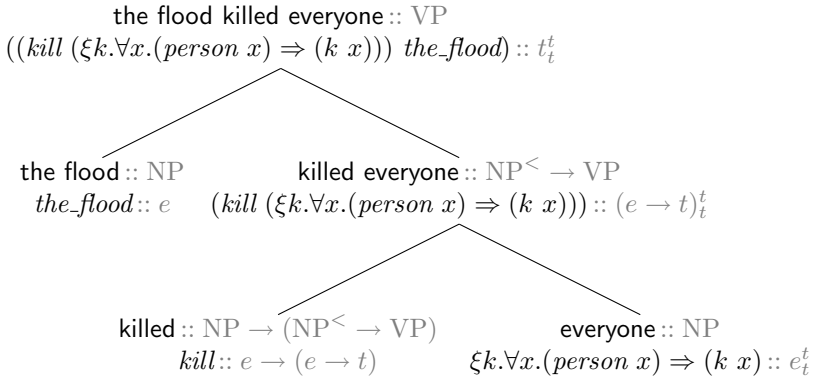
Let us look at the types. The generalized quantifier is of type $(e \to t) \to t$. It expects a predicate into which it can be inserted as argument. In the case of everyone, it expects a predicate and fills its argument position with a universally bound variable, thereby yielding a sentence of type t. So if we have a sentence like The flood killed everyone, due to a type mismatch, the verb denotation cannot be applied to the denotation of everyone, nor the other way around. (We saw that in Section 5.1 above.) Rather we would need to apply everyone to the whole sentence denotation, where the argument position of everyone was abstracted over:



Our new denotation using ξ instead of λ, on the other hand, is of type e^t_t. It will locally behave like an expression of type e, i.e. the verb denotation can take it as an argument just like every other noun phrase of type e. This is because the variable x is actually all that will remain in that argument position once the

reduction rule for shift was applied. The rest, i.e. the quantificational part, will be enabled to take scope over the rest of the expression. There it will behave like a generalized quantifier of type $(e \to t) \to t$.

A derivation for the sentence The flood killed everyone will proceed by simply merging the verb killed with the object noun phrase everyone and after that with the subject noun phrase the flood. No displacement is involved and the semantic dimension will simply amount to functional application. Assume, for the sake of simplicity, that the denotation of the flood is some entity of type e. Then the derivation tree looks as follows:



Suppose we wrap the semantic expression at the top in a delimiter:

$$\langle\, ((\mathit{kill}\ (\xi k.\forall x.(\mathit{person}\ x) \Rightarrow (k\ x)))\ \mathit{the\_flood})\, \rangle$$

Then we can apply the reduction rules for shift and reset and arrive at the desired expression of type t:

$$\forall x.(\mathit{person}\ x) \Rightarrow ((\mathit{kill}\ x)\ \mathit{the\_flood})$$

This way, the two roles everyone has to play are reconciled: upon merge it serves as an argument of type e, and upon evaluating the resulting semantic expression it takes logical scope over the whole expression.

And this is exactly how derivations will proceed in this chapter.

Figure 5.1 gives the denotations we assume for everyone and someone and the corresponding determiners every and some. They are exactly like the familiar denotations on page 93, with the only difference that we use ξ instead of λ as the variable binder. Also note that they satisfy the mapping $^\circ$ between syntactic and semantic types from Definition 14 on page 103 above.

Now we still need to introduce a reset into the derivation, that delimits the context that noun phrases can take scope over. Here I assume that the complementizer introduces this reset, because since CP is the designated category, it should be associated with an interpretation that has no unfinished business like not yet executed control transfers. (In Section 5.6, we will look at alternatives.) Furthermore, this ensures that scope construal takes place at the

Form	Meaning
everyone :: NP	$\xi k.\forall x.(person\ x) \rightarrow (k\ x) :: e_t^t$
someone :: NP	$\xi k.\exists x.(person\ x) \wedge (k\ x) :: e_t^t$
every :: N → NP	$\lambda P \xi k.\forall x.(P\ x) \rightarrow (k\ x) :: (e \rightarrow t) \rightarrow e_t^t$
some :: N → NP	$\lambda P \xi k.\exists x.(P\ x) \wedge (k\ x) :: (e \rightarrow t) \rightarrow e_t^t$

Figure 5.1: Lexical entries for the quantificational noun phrases everyone and someone, and the corresponding determiners every and some.

sentence level. An immediate consequence of the fact that a shift always captures the context up to the nearest enclosing delimiter is that quantificational noun phrases occurring in some clause can take scope only over this clause. We will see this later in this section. (I will consider indefinites, which are known to be able to take much wider scope, in Section 5.4.3 below.)

Complementizers are of category VP → CP, like in our example grammar at the end of Chapter 3. The corresponding type is $t \rightarrow t$. For the sake of completeness, we do not only include that and the empty complementizer, which return a declarative sentence of type t, but also add a complementizer whether returning an interrogative sentence. However, we will not use it before Section 5.5.

Form	Meaning
that :: VP → CP	$\lambda p.\langle p \rangle :: t \rightarrow t$
ϵ :: VP → CP	$\lambda p.\langle p \rangle :: t \rightarrow t$
whether :: VP → CP	$\lambda p.\langle p \rangle :: t \rightarrow q$

Figure 5.2: Lexical entries for the complementizers that and whether.

Let us walk through some examples. First consider a simple sentence containing only one quantificational noun phrase:

(5.43) Ishtar admires some human.

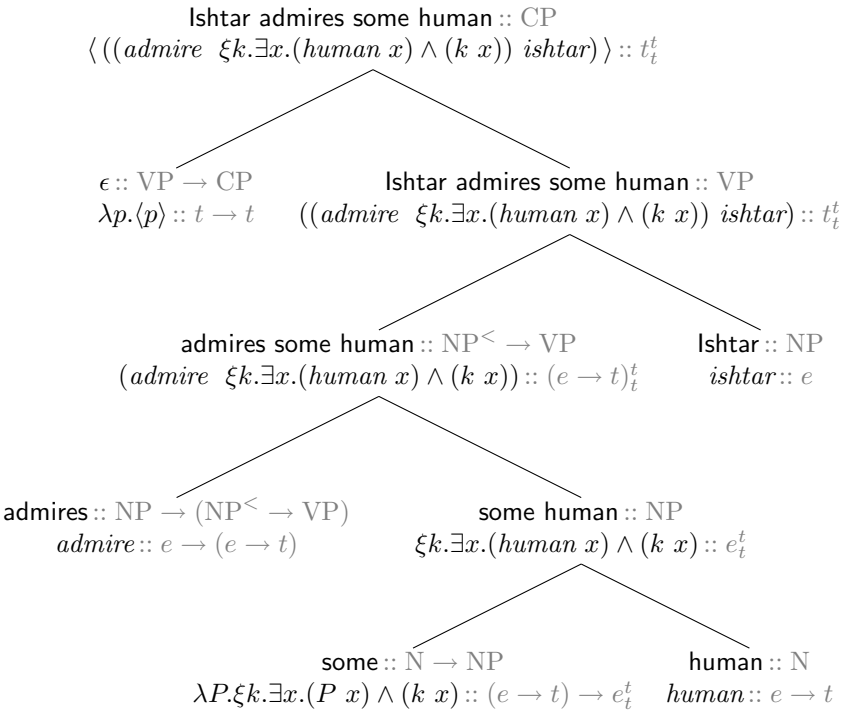
The derivation tree is given in Figure 5.3. The semantic expression that is constructed is the following:

$$\langle\, ((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ ishtar) \,\rangle$$

According to our operational semantics, it reduces like follows. First, the shift captures the context up to the nearest enclosing reset, which is:

$$((admire\ [\,])\ ishtar)$$

Figure 5.3: Derivation tree for Ishtar admires some human.



Then this context is enclosed by an additional reset and reified as a function, which amounts to $\lambda z.\langle\,((admire\ z)\ ishtar)\,\rangle$. Next, this function is substituted for k in the expression $\exists x.(human\ x) \wedge (k\ x)$. We thus arrive at:

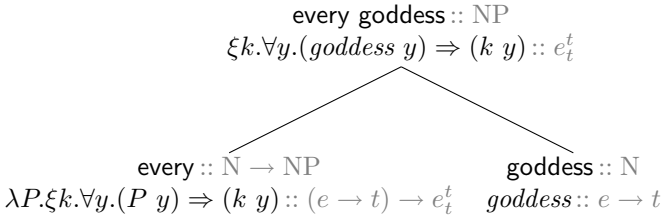
$$\langle\,\exists x.(human\ x) \wedge (\lambda z.\langle\,((admire\ z)\ ishtar)\,\rangle\ x)\,\rangle$$

Applying beta-reduction, this reduces to $\langle\,\exists x.(human\,x) \wedge \langle\,((admire\,x)\ ishtar)\,\rangle\rangle$. Finally we can get rid of the resets because they enclose pure expressions without any further shifts. We thereby arrive at the semantic expression for (5.43) that we aimed for: $\exists x.(human\ x) \wedge ((admire\ x)\ ishtar)$.

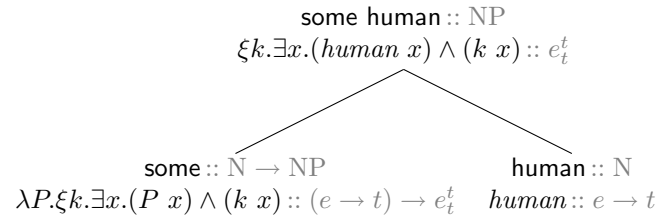
Next, let us consider an example with two quantificational noun phrases:

(5.44) Every goddess admires some human.

The NP every goddess is built by merging the determiner with the noun:



The NP some human is built analogously:

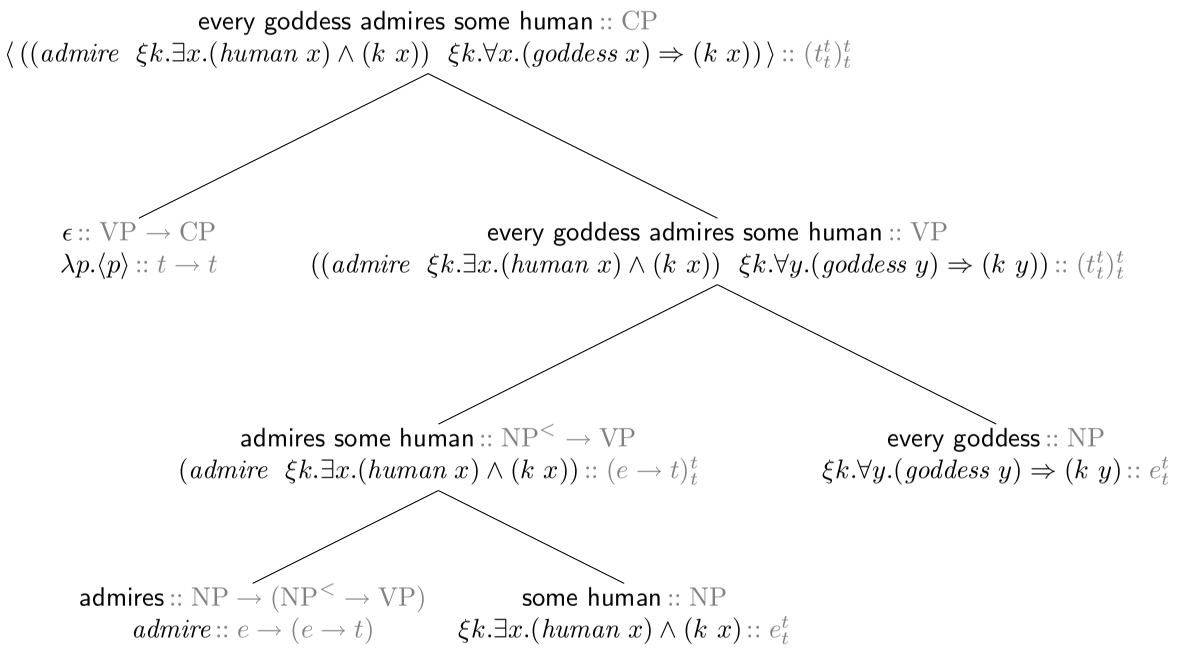


The derivation tree for the whole sentence is given in Figure 5.4. The final semantic expression at the top node is the application of the predicate *admire* to the denotation of some human and the denotation of every goddess, enclosed by a reset:

$$\langle\,((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ \ \xi k.\forall x.(goddess\ x) \Rightarrow (k\ x))\,\rangle$$

Since we did not restrict the application of the reduction rule for shift in applications, there are two ways to reduce this expression. One is to first reduce the shift introduced by every goddess and only afterwards reduce the shift introduced by some human. This way, the universal quantifier captures the context first and takes scope over it, enclosing this context with a new reset. Inside that context we still have the existential quantifier. It then captures that context up to the new delimiter and thus takes scope below the universal quantifier.

Figure 5.4: Derivation tree for Every goddess admires some human.



Thereby we derive the linear scope reading where the every goddess takes scope over some human. Here is how the reduction proceeds:

$$\begin{aligned}
&\langle\,((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ \ \xi k.\forall y.(goddess\ y) \Rightarrow (k\ y))\,\rangle\\
&\rhd \langle\,\forall y.(goddess\ y) \Rightarrow (\lambda z.\langle\,((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ z)\,\rangle\ y)\,\rangle\\
&\rhd \langle\,\forall y.(goddess\ y) \Rightarrow \langle\,((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ y)\,\rangle\rangle\\
&\rhd \langle\,\forall y.(goddess\ y) \Rightarrow \langle\,\exists x.(human\ x) \wedge (\lambda z.\langle\,((admire\ z)\ y)\,\rangle\ x)\,\rangle\rangle\\
&\rhd \langle\,\forall y.(goddess\ y) \Rightarrow \langle\,\exists x.(human\ x) \wedge \langle\,((admire\ x)\ y)\,\rangle\rangle\rangle\\
&\rhd\ \ \forall y.(goddess\ y) \Rightarrow \exists x.(human\ x) \wedge ((admire\ x)\ y)
\end{aligned}$$

The second possibility is to reduce the quantifiers in the opposite order: first reduce the shift expression introduced by some human, and after that reduce the shift expression introduced by every goddess. This way, the existential quantifier is the first to capture the context and the universal quantifier will eventually be assigned scope below it. The result is the inverse scope reading, where some human takes scope over every goddess. Here is how the reduction proceeds:

$$\begin{aligned}
&\langle\,((admire\ \ \xi k.\exists x.(human\ x) \wedge (k\ x))\ \ \xi k.\forall y.(goddess\ y) \Rightarrow (k\ y))\,\rangle\\
&\rhd \langle\,\exists x.(human\ x) \wedge (\lambda z.\langle\,((admire\ z)\ \ \xi k.\forall y.(goddess\ y) \Rightarrow (k\ y))\,\rangle\ x)\,\rangle\\
&\rhd \langle\,\exists x.(human\ x) \wedge \langle\,((admire\ x)\ \ \xi k.\forall y.(goddess\ y) \Rightarrow (k\ y))\,\rangle\rangle\\
&\rhd \langle\,\exists x.(human\ x) \wedge \langle\,\forall y.(goddess\ y) \Rightarrow (\lambda z.\langle\,((admire\ x)\ z))\,\rangle\ y)\,\rangle\rangle\\
&\rhd \langle\,\exists x.(human\ x) \wedge \langle\,\forall y.(goddess\ y) \Rightarrow \langle\,((admire\ x)\ y)\,\rangle\rangle\rangle\\
&\rhd\ \ \exists x.(human\ x) \wedge \forall y.(goddess\ y) \Rightarrow ((admire\ x)\ y)
\end{aligned}$$

So as long as no particular order of evaluation is fixed, all orders are licit. For two quantifiers this leads to two possible orders which result in two different scope readings. This correctly derives the observed scope ambiguity. And of course this does not only work for a verb with two quantificational noun phrases as arguments but more generally in all kinds of cases. Most importantly, quantifiers can take scope independent of the syntactic position they occur in. (We will turn to restrictions on this below.) Let us consider one more example, the sentence in (5.45).

(5.45) Someone from every city hates Gilgamesh.

Just like before, we have two quantifiers that take scope over the whole sentence, and within this sentence both linear and inverse scope readings can be derived, depending on which quantifier we evaluate first. The derivation proceeds like in the examples above. To understand this, let us first assume a lexical entry for the preposition from:

$$(\textsf{from} :: \mathrm{NP} \rightarrow (\mathrm{NP}^{<} \rightarrow \mathrm{NP}), \mathit{from} :: e \rightarrow (e \rightarrow e))$$

It is merged with the NP every city, and the result is merged with the NP someone. Both combinations are given in Figure 5.5. The resulting semantic

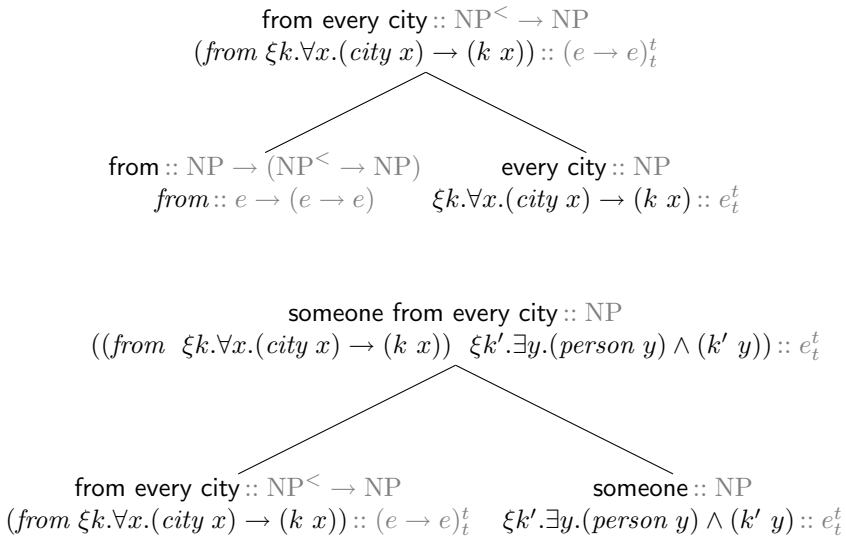


Figure 5.5: Derivation trees for someone from every city.

expression is the following:

$$((\mathit{from}\ \ \xi k.\forall x.(\mathit{city}\ x) \rightarrow (k\ x))\ \ \xi k'.\exists y.(\mathit{person}\ y) \wedge (k'\ y)) :: e_t^t$$

For better readability, I will abbreviate the bodies of the quantificational noun phrases as E_{person} and E_{city}. The expression then reads like this:

$$((\mathit{from}\ \ \xi k.\forall x.E_{\mathsf{city}})\ \ \xi k'.\exists y.E_{\mathsf{person}}) :: e_t^t$$

The rest of the derivation tree, combining someone from every city with hates Gilgamesh, is given in Figure 5.6. The resulting semantic expression is:

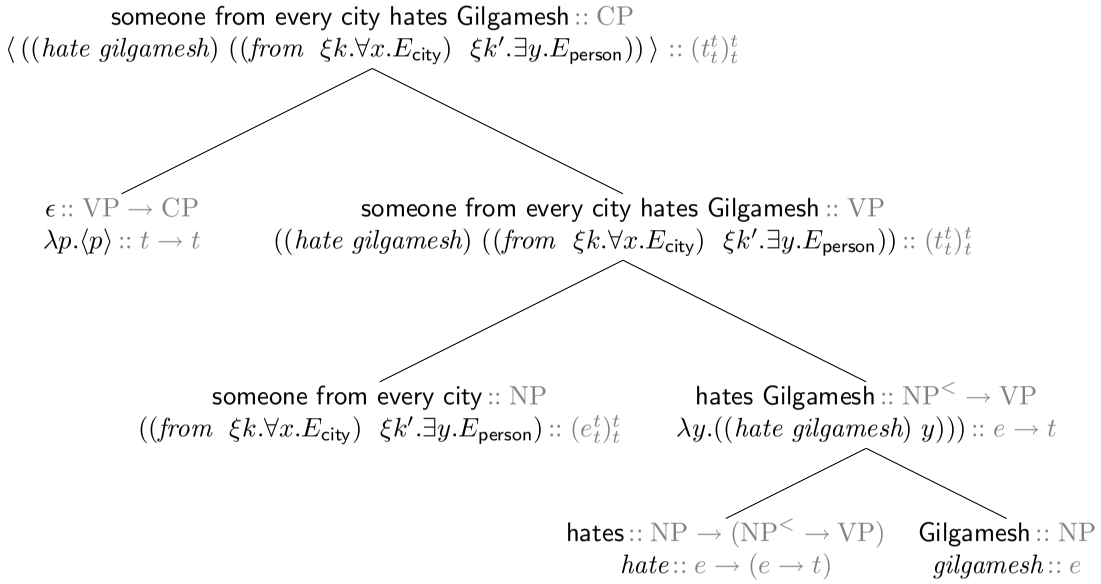
$$\langle\, ((\mathit{hate}\ \mathit{gilgamesh})\ ((\mathit{from}\ \ \xi k.\forall x.E_{\mathsf{city}})\ \ \xi k'.\exists y.E_{\mathsf{person}}))\,\rangle$$

It contains two ξ-expressions, so there are two possible ways to reduce the expression, depending on the order in which we reduce the two subexpressions. The two possibilities result in two scope readings. If we first reduce the existential quantifier, then the existential quantifier will have wide scope, we thus arrive at the linear scope reading (5.46a). If we first reduce the universal quantifier, then the universal quantifier will have wide scope, i.e. we arrive at the inverse scope reading (5.46b). I refrain from spelling out the reductions; they proceed exactly parallel to the ones for (5.44) above.

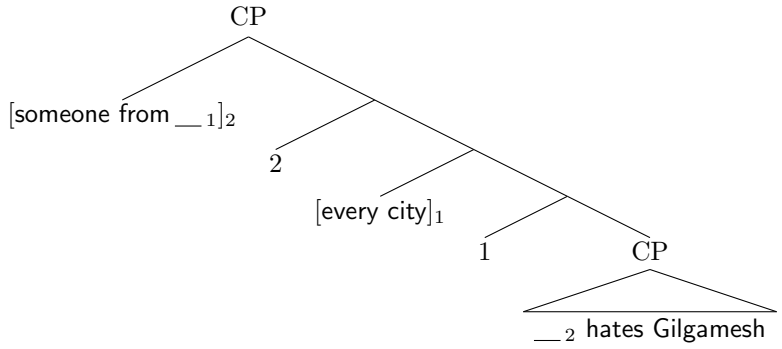
(5.46) a. $\exists y.(\mathit{person}\ y) \wedge \forall x.(\mathit{city}\ x) \Rightarrow ((\mathit{hate}\ \mathit{gilgamesh})\ ((\mathit{from}\ x)\ y))$

b. $\forall x.(\mathit{city}\ x) \Rightarrow \exists y.(\mathit{person}\ y) \wedge ((\mathit{hate}\ \mathit{gilgamesh})\ ((\mathit{from}\ x)\ y))$

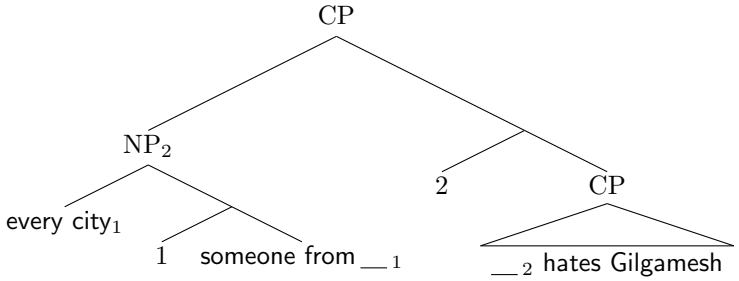
Figure 5.6: Derivation trees for Someone from every city hates Gilgamesh.



How would an approach using Quantifier Raising treat examples like (5.45)? In order to derive the linear scope reading (5.46a), first the noun phrase every city would have to be raised and then the remnant noun phrase someone from $\_\__1$ would have to be raised to a higher position in order to take wide scope. This, however, would leave the gap $\_\__1$ unbound:



May's solution [75] is to conclude that noun phrases are scope islands, that is, that the noun phrase every city can only raise inside the containing NP but not any higher:



This requires some additional work to get the types right, but it seems natural given that Quantifier Raising is extraction and NPs are usually extraction islands (see (5.47)).

(5.47) a. * [From where]$_1$ did [someone $\_\__1$] hate Gilgamesh?
b. * Where$_1$ did [someone from $\_\__1$] hate Gilgamesh?

The advantage of our approach (and similar approaches like Barker's [5]) is that it does not need any further assumptions or type shifting rules. Scope in (5.45) is established exactly like in other sentences. This also extends to other examples. For example, pied piping constructions like (5.48) can straightforwardly get an interpretation in the same way as every other sentence containing wh-phrases. (We did not yet see the denotation of wh-phrases but they will be interpreted similar to quantificational noun phrases). Like with all other examples so far, the wh-expression does not need to escape its syntactic domain in order to take semantic scope.

(5.48) [The god of whose ancestors] did Shamhat pray to?

With all this freedom to establish scope, let us now look at restrictions. First, we return to the clause boundedness of quantifiers, that we briefly mentioned when introducing the complementizer denotations. The quite robust observation from Section 2.3 is that quantifiers can take scope only over the clause they occur in, especially they cannot outscope quantifiers occurring in a higher clause. This is illustrated in (5.49), which only has the linear scope reading (5.49a) but not the inverse scope reading (5.49b).

(5.49) [$_{\text{CP1}}$ Everyone knows [$_{\text{CP2}}$ that Ishtar is angry with most humans]].

a. Every x knows that for most humans y, Ishtar is angry with y.

b. For most humans y, every x knows that Ishtar is angry with y.

In the account of scope construal sketched so far, this follows directly from the fact that the complementizers involved in building both CPs introduce a delimiter. For example in (5.49), most humans can capture the context only up to the nearest enclosing reset, which is introduced by the embedded that. The way we specified the evaluation contexts used in the reduction rule for shift, there is no way to skip this reset. Hence most humans can take scope only over CP2 but not over CP1. This captures that (5.49) does not have a reading where most humans outscopes everyone.

Also, scope islands like relative clauses and complex NPs, as in (5.50a) and (5.50b) respectively, are straightforward, because in both cases the quantificational noun phrase occurs inside a CP which delimits its scope.

(5.50) a. A beast [which slaughtered every sheep] was hunted down.

b. Anu heard the rumor [that every beast died].

Nevertheless, the theory of scope we have right now is both too permissive and too restrictive. First, it is too permissive, because scope ambiguities do not arise with all quantifiers. Recall from Chapter 2 that weak quantifiers such as no human cannot outscope other quantifiers. And second, our treatment of scope is too restrictive, because indefinites are extremely free in taking scope at an arbitrarily high point in the structure. We will first focus on modelling the behavior of weak quantifiers and turn to indefinites only in Section 5.4.3.

5.4.2 Weak quantifiers

Let us look at how we could restrict scope amibguities, such that weak quantifiers cannot outscope other quantifiers but only derive linear scope readings. What allowed us to derive scope amibiguities in the last section was that we did not restrict evaluation contexts, i.e. we did not specify a certain order of evaluation. For sentences with two quantifiers, there were two possibilities to reduce it, depending on which quantifie is evaluated first. Those two possibilities gave

rise to two readings. So fixing an evaluation order would leave only one of the possibilities and rule out the other one. This is done by specifying a slightly different version D' of subcontexts D, where $\mathbf{F}$ ranges over pure expressions:

$$\mathbf{D}' ::= [\,] \mid (\mathbf{E}\ \mathbf{D}') \mid (\mathbf{D}'\ \mathbf{F})$$

I will call such contexts *weak delimited evaluation contexts*, or weak contexts for short.

The way D' is defined now, an applicand can be reduced only when the argument is a pure expression, i.e. does not contain any shifts. That is, the reduction of impure expressions in an application has to proceed from outside to inside. Suppose we have the application of a two-place predicate *pred* to two ξ-expressions:

$$\langle\,((\mathit{pred}\ \ \xi k_1.E_1)\ \ \xi k_2.E_2)\,\rangle$$

Changing the context a shift can capture to D' leaves only one possibility to reduce this expression: First $\xi k_2.E_2$ has to be evaluated and only then $\xi k_1.E_1$ can also be evaluated. This is because if $\xi k_1.E_1$ was evaluated first, it would capture the context $((\mathit{pred}\ [\,])\ \ \xi k_2.E_2)$, which is not a licit weak context D'. If you consider *pred* as a verb denotation, $\xi k_1.E_1$ as the object noun phrase of the verb and $\xi k_2.E_2$ as the subject noun phrase, then the fixed evaluation order amounts to E_2 taking scope over E_1, i.e. derives the linear scope reading if E_1 and E_2 contain scope taking operators. We will see an explicit example in more detail a bit later.

Now we have two subcontexts that we could use in the reduction rule for shifts. Accordingly, we want to have two shifts at our disposal: one that captures contexts D, and one that captures weak contexts D'. To this end, the definition of ξ-expressions is changed such that it encodes which context is captured. We do this by means of a superscript **Mode**.

$$\begin{aligned}\mathbf{E} &::= \ldots \mid (\xi^{\mathbf{Mode}} k :: \tau \to \alpha.\mathbf{E} :: \beta) :: \tau^{\beta}_{\alpha}\\ \mathbf{Mode} &::= \text{weak} \mid \text{strong}\end{aligned}$$

Now we need two reduction rules, one for ξ^{strong} using D, and one for ξ^{weak} using D':

$$\begin{aligned}C\,[\langle D\,[\xi^{\text{strong}} k.E]\rangle] &\rhd C\,[\langle E\{k \mapsto \lambda x.\langle D[x]\rangle\}\rangle]\\ C\,[\langle D'[\xi^{\text{weak}} k.E]\rangle] &\rhd C\,[\langle E\{k \mapsto \lambda x.\langle D'[x]\rangle\}\rangle]\end{aligned}$$

Note that the reduction rule itself is exactly the same; the only difference is which kind of context is captured. In the following, I will often write ξ as short

for ξ^{strong}. This way, all instances of ξ from the last section rightly correspond to the ξ^{strong} of this section. Furthermore, I will usually write ξ^{weak} as ξ'. This is a form that is slightly better readable and corresponds to the use of the apostrophe in the definition of contexts D and D'.

The main consequence for the semantics of our grammar fragment is the following: For quantifier denotations using ξ^{weak}, the evaluation order is fixed, for quantifier denotations using ξ^{strong}, the evaluation order is free. That is, the former derive only linear readings, while the latter allow for scope ambiguities.

To see weak quantifiers in action, consider example (5.51a), which has a linear reading (for most gods there is no human they admire) and does not have an inverse scope reading (no human is such that most gods admire him). When deriving the corresponding semantic interpretation, we arrive at (5.51b), using ξ^{weak}. The denotation of no is given by $\neg\exists$ and the denotation of most P are Q is represented as $\mathbf{Most}\, x{:}(P\ x).(Q\ x)$. Thus most can be seen as relating two sets; the exact modeltheoretic interpretation, however, is not of concern here.

(5.51) a. Most gods admire no human.

b. $\langle\, ((admire\ \ \xi' k.\neg\exists\, x.(human\ x) \wedge (k\ x))\ \ \xi' k.\mathbf{Most}\, y{:}(god\ y).(k\ y))\,\rangle$

There is only one possibility to reduce the expression in (5.51b): First, the denotation of most gods has to be evaluated. This is because reducing the denotation of no human first would capture the following illicit weak context:

$$((admire\ [\,])\ \ \xi k.\mathbf{Most}\, y{:}(god\ y).\wedge (k\ y))$$

The reduction proceeds as follows, according to the reduction rule for ξ^{weak}:

$$\begin{aligned}
&\langle\, ((admire\ \ \xi' k.\neg\exists x.(human\ x) \wedge (k\ x))\ \ \xi' k.\mathbf{Most}\, y{:}(god\ y).(k\ y))\,\rangle\\
&\rhd \langle\, \mathbf{Most}\, y{:}(god\ y).(\lambda z.\,\langle\, ((admire\ \ \xi' k.\neg\exists x.(human\ x) \wedge (k\ x))\ z)\,\rangle\ y)\,\rangle\\
&\rhd \langle\, \mathbf{Most}\, y{:}(god\ y).\langle\, ((admire\ \ \xi' k.\neg\exists x.(human\ x) \wedge (k\ x))\ y)\,\rangle\rangle
\end{aligned}$$

Note the role of static scoping here: The reduction rule introduces a new reset around the captured context. This limits the context the yet unreduced ξ-expression will capture to the context below the operator **Most**. Hence, no human is not able to outscope most gods. The reduction proceeds as follows:

$$\begin{aligned}
&\langle\, \mathbf{Most}\, y{:}(god\ y).\langle\, ((admire\ \ \xi' k.\neg\exists x.(human\ x) \wedge (k\ x))\ y)\,\rangle\rangle\\
&\rhd \langle\, \mathbf{Most}\, y{:}(god\ y).\langle\, \neg\exists x.(human\ x) \wedge (\lambda z.\langle\, ((admire\ z)\ y)\,\rangle\ x)\,\rangle\rangle\\
&\rhd \langle\, \mathbf{Most}\, y{:}(god\ y).\langle\, \neg\exists x.(human\ x) \wedge \langle\, ((admire\ x)\ y)\,\rangle\rangle\rangle
\end{aligned}$$

Finally the resets can be deleted and the result is:

$$\mathbf{Most}\, y{:}(god\ y).\neg\exists x.(human\ x) \wedge ((admire\ x)\ y)$$

This is the linear scope reading and it is the only reading that can be derived with the evaluation order fixed like above.

The results are accordingly when considering sentences with a strong quantifier and a weak quantifier: the strong one can outscope the weak one but not vice versa.

Let me finally note that the approach to modeling different scope behavior demonstrated here is similar to one that Shan ([99],[100]) proposed. He employs a hierarchy of generalized shifts and resets superscripted with strength levels. Quantifiers can take control at different levels and depending on the level, they can or cannot outscope each other. Only if they take control at the same level, scope ambiguities occur. Shan's approach thus differs from ours in maintaining a fixed evaluation order and relying on the hierarchy of quantifiers to determine scope behavior. My approach in this chapter, on the other hand, invokes two different shifts (for weak and strong quantifiers) together with one delimiter for both of them, and enforced an evaluation order for weak quantifiers which prevents them from outscoping other quantifiers.

A yet different possibility was proposed by Stabler [105], who adopted the idea that different quantifiers check features in different functional domains. In our terms, this can be modeled by introducing a different shifts and resets for all those domains. We would then derive that quantifiers can take scope only in their specific domain and cannot outscope quantifiers in higher domains.

Now, before moving on, let us turn to the exceptional wide scope behavior of indefinites. I will sketch how the mechanism employed for strong and weak quantifiers can be extended to also capture those cases. The strategy should be familiar by now: leave the reduction rule for shift as it is but change the context that the shift captures.

5.4.3 Free scope

Existential noun phrases like indefinites are known for being able to take unrestricted wide scope, as already mentioned in Section 2.4. This is illustrated again in (5.52), which arguably has the reading that there is a specific zombie for which everyone knows that no-one believes the rumor that this zombie is in the garden. That is, although embedded inside a complex noun phrase that occurs itself in an embedded clause, the indefinite seems to be able take scope over the matrix clause.

(5.52) Everyone knows [$_{\text{CP1}}$that no-one believes [the claim [$_{\text{CP2}}$that some zombie is in the garden]]].

Note that the mechanism of the previous two sections does not allow this possibility: since already the embedded clause CP2 introduces a delimiter, all ξ-expressions inside that clause can capture only the context up to that delimiter. What existential noun phrases are able to do amounts in our terms to capturing an arbitrary context, i.e. a context up to any enclosing delimiter, not necessarily the closest one.

As already anticipated, this can be achieved by changing the context that the shift can capture, more specifically by specifying the reduction rule not by of using D or D', which both do not contain any resets, but using C, which was defined as being arbitrary contexts containing any number of resets. In order to keep the shift operator with this exceptional wide scope behavior apart from the shift operators ξ^{weak} and ξ^{strong} from above, we add a new mode that we call *free*.

$$\textbf{Mode} ::= \ldots \mid \text{free}$$

We therefore have a new operator ξ^{free} with the following reduction rule:

$$C[\langle C[\xi^{\text{free}} k.E]\rangle] \rhd C[\langle E\{k \mapsto \lambda x.\langle C[x]\rangle\}\rangle]$$

This way, an expression of form $\xi^{\text{free}} k.E$ is not restricted to capturing the context up to the nearest enclosing reset, but can capture the context up to an arbitrary reset.

Let us illustrate this with an example. Consider (5.53a). The existential some zombie can either take narrow scope, yielding a reading where everyone takes scope over the existential, or wide scope, yielding the reading that there is some specific zombie that everyone believes to be able to run. That is, the intermediate clause boundary does not restrict the scope of the existential, although it would, in contrast, restrict the scope of the universal (see (5.53b), which only has a linear scope reading).

(5.53) a. Everyone believes [that some zombie is able to run].

b. Someone believes [that every zombie is able to run].

The denotation of the noun phrase will be an impure generalized quantifier denotation as familiar from the previous sections, using the operator ξ^{free}:

$$\xi^{\text{free}} k.\exists x.(\mathit{zombie}\ x) \wedge (k\ x)$$

Let us assume that the denotation of is able to can be represented as a predicate *isAbleTo* of type $(e \to t) \to (e \to t)$, i.e. that applies to a property (presumably expressing some action like running or hunting bears) as well as to an individual, and predicates of this individual that it is able to do the specified action. Then the denotation of the embedded clause that some zombie is able to run amounts to the following expression:

$$\langle\, ((\mathit{isAbleTo}\ \ \mathit{run})\ \ \xi^{\text{free}} k.\exists x.(\mathit{zombie}\ x) \wedge (k\ x))\,\rangle$$

I abbreviate the denotation of some zombie as $\xi^{\text{free}}k.E_{\text{zombie}}$. And analogously, I abbreviate the denotation of everyone, $\xi k.\forall y.(person\ y) \Rightarrow (k\ y)$, as $\xi k.E_{\text{person}}$.The semantic expression corresponding to the whole sentence then is:

$$\langle\,((believe\ \langle\,((isAbleTo\ \ run)\ \ \xi^{\text{free}}k.E_{\text{zombie}})\,\rangle)\ \ \xi k.E_{\text{person}})\,\rangle$$

The expression $\xi k.E_{\text{person}}$ captures the context up to the nearest enclosing delimiter, which is the outer reset. The existential $\xi^{\text{free}}k.E_{\text{zombie}}$, on the other hand, captures the context up to an arbitrary enclosing delimiter, which hence can either be the inner reset or the outer reset. Capturing the inner reset leads to the narrow scope reading (5.54a), and capturing the outer reset leads to the wide scope reading in (5.54b).

(5.54) a. $\forall y.(person\ y) \Rightarrow ((believe\ \ \exists x.(zombie\ x) \wedge ((isAbleTo\ \ run)\ x))\ y)$

b. $\exists x.(zombie\ x) \wedge \forall y.(person\ y) \Rightarrow ((believe\ ((isAbleTo\ \ run)\ x))\ y)$

For the unambiguous sentence (5.53b), however, we would derive only one reading. The sentence denotation would be like above with the difference that the strong quantifier resides in the embedded clause and the existential resides in the matrix clause. Abbreviating the denotation of the universal every zombie as $\xi k.E_{\text{zombie}}$ and the existential someone as $\xi^{\text{free}}k.E_{\text{person}}$, it amounts to:

$$\langle\,((believe\ \langle\,((isAbleTo\ \ run)\ \ \xi k.E_{\text{zombie}})\,\rangle)\ \ \xi k^{\text{free}}.E_{\text{person}})\,\rangle$$

Again, the universal (here $\xi k.E_{\text{zombie}}$) captures the context up to the nearest enclosing delimiter, which is the inner reset. The existential, on the other hand, captures a context up to an arbitrary delimiter. Since there is only one in this case, namely the outer reset, only one reading, given in (5.55), is derived.

(5.55) $\exists y.(person\ y) \wedge ((believe\ \ \forall x.(zombie\ x) \Rightarrow ((isAbleTo\ \ run)\ x))\ y)$

A desirable consequence of the treatment of indefinites sketched here is that it automatically derives the right truth conditions for sentences like (5.56), which prove problematic for unselective binding approaches.

(5.56) If some human is sacrificed, Cthulhu will awake.

First note that (5.56) has two readings, depending on whether the existential a human takes narrow scope over the if-clause, which is schematically presented in (5.57a), or wide scope over the whole sentence, which is represented in (5.57b).

(5.57) a. ($\exists x.\ x$ is a human and x is sacrificed) $\Rightarrow$ Cthulhu will awake

b. $\exists x.\ x$ is a human and (x is sacrificed $\Rightarrow$ Cthulhu will awake)

c. $\exists x.$((x is a human and x is sacrificed) $\Rightarrow$ Cthulhu will awake)

The interpretation that unselective binding approaches usually derive is the one in (5.57c). However, this is not correct because it would already be true in case there is an x which is either not a human or is not sacrificed.

Let us look at what happens in our approach. The denotation for some human is the expected impure generalized quantifier using the free shift operator:

$$\xi^{\text{free}}k.\exists x.(human\ x) \wedge (k\ x)$$

An important fact here is that k occurs only in the second conjunct. That is, whatever context is captured, it will be plugged into that second conjunct and not affect the restriction $(human\ x)$. Let us see what this means for the construction of the sentence denotation. Assume that if. . . then subcategorizes two CPs, i.e. there are three resets introduced: one enclosing the whole sentence, one enclosing the if-clause, and one enclosing the then-clause. Then we derive the following denotation for the whole sentence:

$$\langle\, \langle\, (sacrificed\ \ \xi^{\text{free}}k.\exists x.(human\ x) \wedge (k\ x))\, \rangle \Rightarrow \langle\, (awake\ Cthulhu)\, \rangle$$

The ξ^{free}-expression now captures a context up to some enclosing delimiter. This is either the reset enclosing the if-clause or the reset enclosing the whole sentence. The former gives (5.58a) corresponding to the narrow reading (5.57a) and the latter gives (5.58b) corresponding to the wide scope reading (5.57b).

(5.58) a. $(\exists x.(human\ x) \wedge (sacrificed\ x)) \Rightarrow (awake\ Cthulhu)$

b. $\exists x.(human\ x) \wedge ((sacrificed\ x) \Rightarrow (awake\ Cthulhu))$

Thus, the restriction $(human\ x)$ ends up in the right place in both cases. This is due to the above mentioned fact that the captured context is plugged into the second conjunct in the noun phrase denotation, separate from the restriction. Something like in (5.57c) could therefore not happen.

Let me end this subsection with a remark concerning the intention of this subsection. I do not want to make a claim with respect to the question whether wide scope existentials are in fact quantifiers or rather referential expressions. This subsection rather served to demonstrate how exceptional wide scope behavior can in principle be modeled in the advocated approach to quantifiers. Whether one wants to adopt this mechanism or rather rely on different tools is a matter of taste and conviction. In either case, the free scope account of this subsection will become useful again in the next section, when we treat in situ wh-phrases.

So let us now concentrate on yet another kind of scope-taking operators: wh-expressions.

5.5 Wh-phrases

This section sets out to account for the interpretation of the following three types of wh-phrases:

- displaced wh-phrases taking scope at the top of the displacement dependency (e.g. English)

- in situ wh-phrases with a corresponding scope marker in the clause they take scope over (e.g. Japanese)
- true in situ wh-phrases taking scope independent of displacement and scope marking (e.g. Chinese and Hindi)

We will start by looking at displaced wh-phrases. This will require the introduction of indices on control operators, which determine their scope domain in accordance with the syntactic features that are involved in displacement. This approach then straightforwardly extends to wh-phrases in languages with scope markers. Next, we will turn to true in situ wh-phrases, which can actually already be accounted for with the tools introduced so far. In doing so, two kinds of wh-phrases have to be distinguished: wh-phrase whose scope is clause-bound, as in Hindi, and wh-phrases whose scope is free, as in Mandarin Chinese. These two possibilities already suggest the use of the shift operators introduced in the previous section (ξ^{strong} or ξ^{weak} for clause-bound scope and ξ^{free} for free scope).

In the end we will see that the interpretation of wh-phrases can be accounted for in the same line as the interpretation of quantificational noun phrases. That is, we can use one and the same scope mechanism for all operator expressions.

5.5.1 Displaced wh-phrases

Displaced wh-phrases behave differently from quantificational noun phrases in that their scope is determined neither at the closest nor at an arbitrary clause level. Rather it depends on where the wh-phrase is displaced to. In our terms, the scope of the wh-operator is determined by the clausal head that checks its wh-feature. Consider the following examples (5.59) and (5.62). In both cases, the wh-phrase which fight originates in the embedded clause. In (5.59), it is moved to the clause-initial position of CP1 and the corresponding wh-operator accordingly takes scope over the whole sentence. In (5.62), on the other hand, the wh-phrase is fronted only inside CP2 and the corresponding wh-operator accordingly takes scope only over the embedded clause.

(5.59) $[_{\text{CP1}}$ [Which fight]$_1$ did Inana know $[_{\text{CP2}}$ every brave warrior feared $\_\__1]]$?

(5.60) Inana knew $[_{\text{CP}}$ [which fight]$_1$ every brave warrior feared $\_\__1]$.

In order to account for this behavior, we need to be able to capture the feature domain in the semantic dimension as well. To this end, we introduce subscripts to the control operators. The idea is that just like a syntactic feature determines the syntactic domain into which a wh-phrase is displaced, a semantic subscript determines the semantic domain over which a wh-operator takes scope.

The language is extended as follows. First, the definition of semantic expressions **E** is slightly changed. Instead of defining ξ-expressions as being of form $\xi^{\mathbf{Mode}}k.\mathbf{E}$, they are now defined as being of form $\xi^{\mathbf{Mode}}_{\mathbf{Flavor}}k.\mathbf{E}$, where **Flavor** is a feature value (in the case of displaced expressions) or Q (in the case

of quantificational noun phrases). Furthermore, this flavor is also encoded in the type of an impure expression. Impure types are thus not anymore τ_α^β but $\tau_{f:\alpha}^{f:\beta}$, with f being some flavor.

$$\begin{aligned}\mathbf{E} ::= \ldots \mid (\xi^{\mathbf{Mode}}_{\mathbf{Flavor}} k :: \tau \to \alpha.\mathbf{E} :: \beta) :: \tau^{\mathbf{Flavor}:\beta}_{\mathbf{Flavor}:\alpha} \\ \mid \langle \mathbf{E} :: \tau \rangle_{\mathbf{Flavor}} :: \tau \\ \mathbf{Flavor} ::= \mathbf{Value} \mid Q\end{aligned}$$

The type of ξ-expressions is restricted such that the encoded flavor corresponds to the flavor subscript of the shift, according to the following rule:

$$\frac{k :: \tau \to \alpha \qquad E :: \beta}{\xi_f^m k.E :: \tau_{f:\alpha}^{f:\beta}}$$

Since, for now, only wh-features are relevant, we will consider only three new shifts ξ_{wh}, ξ'_{wh}, and ξ^{free}_{wh} together with the corresponding new reset $\langle\,\rangle_{wh}$. But of course we could have more, for the flavor comprise all feature values of the language. This could be useful, for example, for the scope of focus operators. The only flavor that is not a feature value is Q. It will be used for quantificational noun phrases, which we assumed to not be involved in displacement, thus not carry a feature. When talking about quantificational noun phrases of flavor Q, I will drop the subscript, i.e. I will write ξ^m_Q as ξ^m (where m is some mode) and $\langle\,\rangle_Q$ as $\langle\,\rangle$. This way, the abbreviated expressions correspond to the impure expressions we used in the previous sections.

The reduction rule for shifts should then specify that the evaluation context up to a certain matching delimiter is captured. That is, ξ_{wh}-expressions are supposed to capture the context up to some delimiter $\langle\,\rangle_{wh}$. Once the *wh*-domain and the Q-domain are thus separated, the scope of wh-operators will not interfere with the scope of quantifiers. (We will see this with an example at the end of the section.) So evaluation contexts need to be refined in order to encode the flavor of the domain.

Definition 15. *Let f, f' range over* **Flavor**. *Then a family of evaluation contexts D_f, D'_f, C_f, C is defined as follows:*

$$\begin{aligned}\mathbf{D}_f &::= [\,] \mid (\mathbf{E}\ \mathbf{D}_f) \mid (\mathbf{D}_f\ \mathbf{E}) \mid \mathbf{C}_{f'} \\ \mathbf{D}'_f &::= [\,] \mid (\mathbf{E}\ \mathbf{D}'_f) \mid (\mathbf{D}'_f\ \mathbf{F}) \mid \mathbf{C}_{f'} \\ \mathbf{C}_f &::= \mathbf{D}_{f'} \mid \langle \mathbf{C} \rangle_f \\ \mathbf{C} &::= \mathbf{C}_f\end{aligned}$$

Note that we added some clauses compared to the original definition of those contexts. For example, originally, a subcontext D did not contain any resets. Now, a subcontext D_f can contain resets, however no f-resets. This is because we want a ξ_f-expression to capture the context up to the nearest matching delimiter $\langle\,\rangle_f$, however want to allow that this context contains delimiters of another flavor. For example, a wh-denotation will capture a delimiter $\langle\,\rangle_{wh}$ and in doing so can skip arbitrarily many delimiters $\langle\,\rangle_Q$, that in fact do not play a role for its scope construal.

The reduction rules for the control operators now specify that the context up to a matching delimiter is captured. Besides the subscripts, the rules are exactly like before.

$$C[\langle D_f[\xi_f^{\text{weak}} k.E]\rangle_f] \rhd C[\langle E\{k \mapsto \lambda x.\langle D_f[x]\rangle_f\}\rangle_f]$$
$$C[\langle D'_f[\xi_f^{\text{strong}} k.E]\rangle_f] \rhd C[\langle E\{k \mapsto \lambda x.\langle D'_f[x]\rangle_f\}\rangle_f]$$
$$C[\langle C_f[\xi_f^{\text{free}} k.E]\rangle_f] \rhd C[\langle E\{k \mapsto \lambda x.\langle C_f[x]\rangle_f\}\rangle_f]$$

$$\langle F\rangle_f \rhd F$$

Now let us turn to some concrete examples. We start with the denotation of wh-noun phrases like who and wh-determiners like which. They are actually parallel to those of quantificational noun phrases and the according determiners. The only two differences are that they do not use the control operator ξ but ξ_{wh}, and that they change the result type t of the context they capture to q (the type we assumed for questions).

Form	**Meaning**
who^{wh} :: NP	$\xi_{wh} k.\mathfrak{W}\, x.(\mathit{person}\ x) \wedge (k\ x) :: e^{wh:q}_{wh:t}$
which^{wh} :: N → NP	$\lambda P.\xi_{wh} k.\mathfrak{W}\, x.(P\ x) \wedge (k\ x) :: (e \to t) \to e^{wh:q}_{wh:t}$

Figure 5.7: Lexical entries for the wh-noun phrase who and the wh-determiner which.

These denotations employ shifts of the strong mode, because this captures the behavior of wh-phrases adequately. First of all, wh-phrases are displaced to the closest matching feature, thus they should take scope with respect to the closest matching delimiter. And second, no evaluation order needs to be fixed.

Additionally, we need the denotation of a clausal head with a $\bullet wh$-feature that introduces the matching delimiter $\langle\,\rangle_{wh}$. Recall the complementizer de-

notation assumed in the previous sections: $\lambda p.\langle p\rangle$. If the complementizer now carries a probe feature $\bullet wh$, it should introduce not only a Q-flavored delimiter but also a wh-flavored one. The denotation should thus be: $\lambda p.\langle\langle p\rangle\rangle_{wh}$. To arrive there in a systematic way, I assume that the original denotation of the complementizer is kept and that, additionally, the probe feature receives an interpretation that introduces the wh-flavored reset. Moreover, a form string^f is interpreted compositionally as the application of the denotation of the feature f to the denotation of string. Here is how the denotations would like for the empty complementizer ϵ carrying a probe feature $\bullet wh$ (where $[\![\,]\!]$ represents a function mapping forms to meanings):

$$\begin{aligned}[\![\epsilon]\!] &= \lambda p.\langle p\rangle :: t \to t\\ [\![\bullet wh]\!] &= \lambda P\lambda p.\langle (P\ p)\rangle_{wh} :: (t \to t) \to (t \to q)\\ [\![\epsilon^{\bullet wh}]\!] &= ([\![\bullet wh]\!]\ [\![C]\!]) = \lambda p.\langle\langle p\rangle\rangle_{wh} :: t \to t\end{aligned}$$

Let us first consider an easy example, say the derivation of (5.61):

(5.61) What_1 did Gilgamesh think $[_{\mathsf{CP}}$ that Ishtar wanted $\_\__1\,]$?

Here are the lexical items involved:

Form		**Meaning**	
Gilgamesh	$::$ NP	*gilgamesh*	$:: e$
Ishtar	$::$ NP	*ishtar*	$:: e$
what^{wh}	$::$ NP	$\xi_{wh}k.\mathfrak{W}\,x.(k\ x)$	$:: e^{wh:q}_{wh:t}$
think	$::$ CP $\to$ (NP$^<$ $\to$ VP)	*think*	$:: t \to (e \to t)$
wanted	$::$ NP $\to$ (NP$^<$ $\to$ VP)	*want*	$:: e \to (e \to t)$
that	$::$ VP $\to$ CP	$\lambda p.\langle p\rangle$	$:: t \to t$
$\epsilon^{\bullet wh}$	$::$ VP $\to$ CP	$\lambda p.\langle\langle p\rangle\rangle_{wh}$	$:: t \to t$

The first step of the derivation is to merge the embedded verb wanted and the wh-expression what, which thereupon is split. Then the derivation proceeds until the embedded VP is built. I skip these steps, since nothing new or exciting happens. The result is:

$$\big\langle \mathsf{what}^{wh}, \big(\mathsf{Ishtar\ wanted} :: \mathrm{VP}, ((want\ \ \xi_{wh}k.\mathfrak{W}\,x.(k\ x))\ ishtar) :: t^{wh:q}_{wh:t}\ \big)\big\rangle$$

Next the embedded complementizer that is merged, which gives:

$$\big\langle \mathsf{what}^{wh}, \big(\mathsf{Ishtar\ wanted} :: \mathrm{VP}, \langle\,((want\ \ \xi_{wh}k.\mathfrak{W}\,x.(k\ x))\ ishtar)\,\rangle :: t^{wh:q}_{wh:t}\big)\big)\big\rangle$$

Since that does not carry a wh-feature, **remerge** is not triggered. Also, the reduction rule for shift does not yet apply, because the flavors of ξ_{wh} and $\langle\,\rangle$ do not match. The derivation thus proceeds by passing the whole expression to

the matrix verb think, followed by the subject noun phrase Gilgamesh. Ignoring how do-support comes about, the result is the following:

$$\Big\langle \mathsf{what}^{wh}, \big(\mathsf{did\ Gilgamesh\ think\ that\ Ishtar\ wanted} :: \mathrm{VP},\\ ((\mathit{think}\ \langle((\mathit{want}\ \ \xi_{wh}k.\mathfrak{W}\,x.(k\ x))\ \mathit{ishtar})\rangle\ \mathit{gilgamesh}) :: t^{wh:q}_{wh:t}\big)\Big\rangle$$

Now the clausal head carrying the $\bullet wh$-feature is merged.

$$\Big\langle \mathsf{what}^{wh}, \big(\mathsf{did\ Gilgamesh\ think\ that\ Ishtar\ wanted}^{\bullet wh} :: \mathrm{CP},\\ \langle((\mathit{think}\ \langle((\mathit{want}\ \ \xi_{wh}k.\mathfrak{W}\,x.(k\ x))\ \mathit{ishtar})\rangle\ \mathit{gilgamesh})\rangle_{wh} :: t^{wh:q}_{wh:t}\big)\Big\rangle$$

On the syntactic side, this is a configuration in which **remerge** applies. The matching wh-features are checked and the form what at the edge is concatenated with the form of the nucleus. On the semantic side, the reduction rule for shift applies, since the denotation of the clausal head had introduced a matching delimiter $\langle\rangle_{wh}$. The reduction of the semantic expression proceeds by capturing the context up to the outermost reset and substituting it for k. The type thereby changes to the result type q. Finally, the resets can be deleted. The resulting expression if the following:

$$\big(\mathsf{what\ did\ Gilgamesh\ think\ that\ Ishtar\ wanted} :: \mathrm{CP},\\ \mathfrak{W}\,x.((\mathit{think}\ ((\mathit{want}\ x)\ \mathit{ishtar})\ \mathit{gilgamesh}) :: q\big)$$

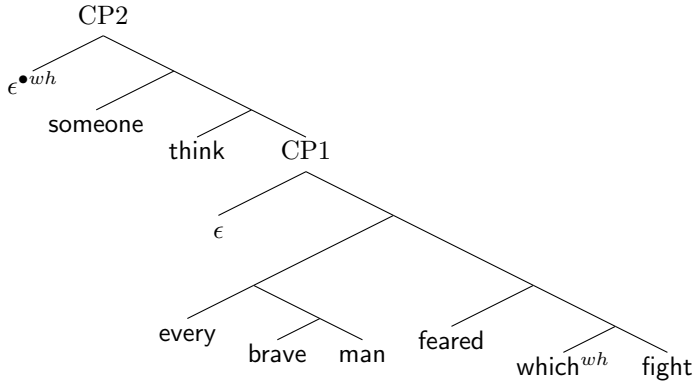
This derivation already showed that the Q-delimiter of the embedded clause does not interfere with the scope construal of the wh-expression. Still, let us walk through one more example, containing both a wh-expression and a quantificational noun phrase, to see that different flavors create different, independent domains. Consider the question (5.62):

(5.62) $[_{\mathrm{CP2}}$ [Which fight]$_1$ did someone think $[_{\mathrm{CP1}}$ every brave man feared $\_\__1]$?

Here are the lexical items involved:

Form		**Meaning**	
someone	$:: \mathrm{NP}$	$\xi k.\forall x.(\mathit{person}\ x) \Rightarrow (k\ x)$	$:: e^t_t$
every	$:: \mathrm{N} \to \mathrm{NP}$	$\lambda P.\xi k.\forall x.(P\ x) \Rightarrow (k\ x)$	$:: (e \to t) \to e^t_t$
man	$:: \mathrm{N}$	man	$:: e \to t$
fight	$:: \mathrm{N}$	fight	$:: e \to t$
brave	$:: \mathrm{N} \to \mathrm{N}$	$\lambda P.\lambda x.(P\ x) \wedge (\mathit{brave}\ x)$	$:: (e \to t) \to (e \to t)$
which^{wh}	$:: \mathrm{N} \to \mathrm{NP}$	$\lambda P.\xi_{wh}k.\mathfrak{W}\,x.(P\ x) \wedge (k\ x)$	$:: (e \to t) \to e^{wh:q}_{wh:t}$
feared	$:: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP})$	fear	$:: e \to (e \to t)$
think	$:: \mathrm{CP} \to (\mathrm{NP}^{<} \to \mathrm{VP})$	think	$:: t \to (e \to t)$
ϵ	$:: \mathrm{VP} \to \mathrm{CP}$	$\lambda p.\langle p\rangle$	$:: t \to t$
$\epsilon^{\bullet wh}$	$:: \mathrm{VP} \to \mathrm{CP}$	$\lambda p.\langle\langle p\rangle\rangle_{wh}$	$:: t \to t$

Here is a schematic derivation tree that shows the order of merging those lexical items:



I will skip over most parts of the derivation, since it proceeds as expected, and only highlight the important points. We start by constructing the NP which fight and merging it with the embedded verb feared. Note that when merging which^{wh} and fight, the wh-feature projects because the determiner is the head:

$$\begin{aligned}&\textbf{merge}\ (\mathsf{which}^{wh} :: \mathrm{N} \to \mathrm{NP}, \ldots)\ (\mathsf{fight} :: \mathrm{N}, \ldots)\\ &= (\mathsf{which\ fight}^{\,wh} :: \mathrm{NP}, \xi_{wh} k.\mathfrak{W}\, x.(\mathit{fight}\ x) \wedge (k\ x) :: e^{wh:q}_{wh:t})\end{aligned}$$

When the NP is merged with the verb, it is split. Again, I leave out the lexical semantics, because it is given in the table above, and the semantic expression we already derived. Furthermore, I abbreviate $\xi_{wh} k.\mathfrak{W}\, x.(\mathit{fight}\ x) \wedge (k\ x)$ as $\xi_{wh} k.E_{\mathsf{fight}}$.

$$\begin{aligned}&\textbf{merge}\ (\mathsf{feared} :: \mathrm{NP} \to (\mathrm{NP}^{<} \to \mathrm{VP}), \ldots)\ (\mathsf{which\ fight}^{\,wh} :: \mathrm{NP}, \ldots)\\ &= \big\langle \mathsf{which\ fight}^{\,wh}, \big(\mathsf{feared} :: \mathrm{NP}^{<} \to \mathrm{VP}, (\mathit{fear}\ \ \xi_{wh} k.E_{\mathsf{fight}}) :: e \to t^{wh:q}_{wh:t}\big)\big\rangle\end{aligned}$$

The derivation proceeds by constructing and merging the subject NP every brave man. The result has the denotation is $\xi k.\forall x.(\mathit{man}\ x) \wedge (\mathit{brave}\ x) \Rightarrow (k\ x)$, which I will abbreviate as $\xi k.E_{\mathsf{man}}$. It is fed to the complementizer, which does not introduce a probe feature but a Q-flavored delimiter. The result is the embedded clause CP1:

$$\begin{aligned}\big\langle \mathsf{which\ fight}^{\,wh}, \big(&\mathsf{every\ brave\ man\ feared} :: \mathrm{CP},\\ &\big\langle\, ((\mathit{fear}\ \ \xi_{wh} k.E_{\mathsf{fight}})\ \ \xi k.E_{\mathsf{man}})\,\big\rangle :: t^{wh:q}_{wh:t}\,\big)\big\rangle\end{aligned}$$

On the semantic side, the shift rule for the Q-flavored quantificational noun phrase applies. The resulting semantic expression is:

$$\langle\, \forall x.(\mathit{man}\ x) \wedge (\mathit{brave}\ x) \Rightarrow ((\mathit{fear}\ (\xi_{wh} k.E_{\mathsf{fight}})\ x)\,\rangle :: t^{wh:q}_{wh:t}$$

Whether we delete the reset now or later does not play a role because in either case it will not interfere with the scope taking of the wh-operator. The derivation proceeds constructing the matrix CP. We skip these steps (and again ignore do-support). The resulting expression corresponding to the whole sentence is

$$\big\langle \textsf{which fight}^{\,wh}, \big(\textsf{did someone think every brave man feared}^{\,\bullet wh} :: \mathrm{CP}, \dots\big)\big\rangle$$

with the following semantic expression of type $(t_t^t)_{wh:t}^{wh:q}$, where the denotation of someone is abbreviated as $\xi k.E_{\textsf{person}}$:

$$\langle\langle\, ((\mathit{think}\ \langle\, \forall x.(\mathit{man}\ x) \wedge (\mathit{brave}\ x) \Rightarrow ((\mathit{fear}\ \ \xi_{wh} k.E_{\textsf{fight}})\ x)\,\rangle\ \ \xi k.E_{\textsf{person}})\,\rangle\rangle_{wh}$$

On the syntactic side, the configuration triggers **remerge**, which checks the features, concatenates the form at the edge with the form of the nucleus and eventually yields the final string which fight did someone think every brave man feared. On the semantic side, we still have two control transfers to execute. First observe that the order in which we do this is not free. Suppose we first applied the reduction rule to the ξ_{wh}-expression corresponding to which fight. It captures a context of type t and changes its type to q. If we then wanted to apply the reduction rule to the ξ-expression corresponding to someone, we would fail, because it requires to capture a context of type t, however finds only one of type q. Therefore, the reduction rule first has to apply to the denotation of someone (which does not change the result type but returns an expression of type t) and only after that to the denotation of which fight. The result of applying the reduction rule first to the denotation of someone is the following (where I abbreviate $(\mathit{brave}\ x) \wedge (\mathit{man}\ x)$ as $(\mathit{braveMan}\ x)$):

$$\langle\langle\, \exists y.(\mathit{person}\ y) \wedge ((\mathit{think}\ \langle\, \forall x.(\mathit{braveMan}\ x) \Rightarrow (\mathit{fear}\ \ \xi_{wh} k.E_{\textsf{fight}})\ x)\,\rangle)\ y)\,\rangle\rangle_{wh}$$

The remaining control transfer then yields the final result:

$$\mathfrak{W}\, z.(\mathit{fight}\ z) \wedge \exists y.(\mathit{person}\ y) \wedge ((\mathit{think}\ (\forall x.(\mathit{braveMan}\ x) \Rightarrow ((\mathit{fear}\ x)\ z)))\ y)$$

In prose it says: For which fight is it the case that there is someone who thought that every brave man feared it? (Ignoring tense, that is.)

This concludes the general mechanism for scope construal of displaced wh-phrases. Note that it works independently of whether the wh-phrase is displaced overtly or covertly. As long as it checks its wh-feature with a head carrying a corresponding probe feature, its scope will be construed in that domain. Let us now look at how this mechanism naturally extends to the scope of wh-phrases in scope marking languages, although no displacement is involved.

5.5.2 Scope marking

In Section 4.4.1 of Chapter 4 we observed that Japanese wh-phrases can obviate islands. Here is one of the examples we saw:

(5.63) *Japanese* (Tsai [117])

John-wa [[dare-o aisiteiru] onna-o] nagutta no
John-TOP who-ACC loves woman-ACC hit Q
'Who is the person x such that John hit the woman who loves x?'

From the island insensitivity we concluded that those wh-expressions are not displaced, not even covertly. That is, they do not check features with a clausal head and, as a consequence, have to be assumed to not carry a wh-feature. (If they did carry a wh-feature, they would automatically be split and carried along until the feature can be checked.)

Recall the observation from Chapter 2 that the scope of a Japanese in situ wh-phrase is determined by the occurrence of a question particle, here ka. The examples we saw were the following: In (5.64a), the particle occurs in the embedded clause, the wh-phrase thus takes scope over the embedded clause only. In (5.64b), on the other hand, the particle occurs in the matrix clause, the wh-phrase thus takes scope over the whole sentence. Moreover, in the presence of two question particles (in the embedded as well as the matrix clause) as in (5.64c), each of them can be taken to determine the scope, the sentence is thus ambiguous between a narrow and a wide scope reading of the wh-phrase.

(5.64) *Japanese* (Bošković [122], Cresti [29])

a. Peter-wa [anata-ga dare-o mita-ka] tazuneta.
Peter-TOP you-NOM who-ACC saw-Q asked
'Peter asked whom you saw.'

b. Kimi-wa [dare-ga kai-ta hon-o yomi-masi-ta]-ka?
you-TOP who-NOM wrote book-ACC read Q
'Which person x is such that you read a book that x wrote?'

c. Hikaru-wa [Akira-ga dare-o hometa-ka] siri-tagatte-imasu-ka?
Hikaru-TOP Akira-NOM who-ACC praised-Q know-want-be-Q
'Does Hikaru want to know whom Akira praised?'
'Which person x is such that Hikaru wants to know whether Akira praised x?'

Let us look at how to specify lexical entries for the wh-expressions and the question particle in order to derive the correct scope effects.

Concerning the wh-phrase dare ('who'), we need to decide on the mode and the flavor of the involved shift operator. The flavor will be assumed to be wh, because the captured context depends not so much on the presence of a clause boundary but rather on its being question marked. As to the mode, Tanaka [116] proposes that a Japanese wh-phrase must take scope according to the closest question marker. (He assumes this to be an LF principle but that does not matter here.) We would therefore assume the mode to be strong or weak, but not free. Let us assume a strong mode. Here is a lexical entry that takes

this into account and is, in fact, completely parallel to the entries for English wh-expressions:

$$(\textsf{dare} :: \text{NP}, \xi_{wh} k.\mathfrak{W}x.(\mathit{person}\ x) \wedge (k\ x) :: e_t^q)$$

Now, what provides the delimiter for the scope of dare? In the previous subsection, it was assumed that the probe feature $\bullet wh$ of the clausal head plays this role. Here we do not have such a feature. However, the clausal head is the question particle ka (according to Takahashi [115]). Since it determines the scope of the wh-expression, it seems straightforward to connect the delimiter to its denotation. In specifying that denotation, I follow Takahashi in taking ka to be ambiguous between a yes/no-question marker and a wh-question marker. He argues for this ambiguity on the basis of different requirements they have: both have to be governed by a tense feature, and the wh-particle ka has to be additionally governed by a politeness feature, whereas the yes/no-particle ka does not. From this point of view, the two readings of (5.64c) are due to the different nature of the two occurrences of ka. If the embedded ka is the wh-particle and the matrix one is the yes/no-particle, the first, direct question reading comes about. If the distribution is vice versa, i.e. the embedded ka is the yes/no-particle and the matrix one is the wh-particle, then the second, wh-question reading comes about.

The tense requirement of both can be captured by assuming an additional projection TP that constitutes the level at which tense features are checked. The politeness requirement, on the other hand, I will omit. Then the following two lexical entries can be specified for ka: (5.65a) for the wh-particle, introducing a wh-flavored delimiter, and (5.65b) for the yes/no-particle, introducing a question operator **?** that turns a declarative clause into an interrogative one (the exact modeltheoretic interpretation is again of no concern here).

(5.65) a. $(\textsf{ka} :: \text{TP}^{<} \to \text{CP}, \lambda p.\langle p \rangle_{wh} :: t \to t)$

b. $(\textsf{ka} :: \text{TP}^{<} \to \text{CP}, \lambda p.\textbf{?}p :: t \to q)$

I will not give any details of the derivation of (5.64c) because it proceeds pretty much like all derivations we saw in this chapter. But I will specify the lexical entries of the expressions involved (see the table below) and show the result they lead to. I ignore case, simplify the denotation of siri-tagatte-imasu, and abbreviate the types $e \to (e \to t)$ and $q \to (e \to t)$ as $e(et)$ and $q(et)$.

Form		**Meaning**	
Hikaru-wa	$:: \text{NP}$	*hikaru*	$:: e$
Akira-ga	$:: \text{NP}$	*akira*	$:: e$
dare-o	$:: \text{NP}$	$\xi_{wh}^{\text{free}} k.\mathfrak{W}x.(\mathit{person}\ x) \wedge (k\ x)$	$:: e_t^q$
hometa	$:: \text{NP}^{<} \to (\text{NP}^{<} \to \text{VP})$	*admire*	$:: e(et)$
siri-tagatte-imasu	$:: \text{CP}^{<} \to (\text{NP}^{<} \to \text{VP})$	*wantToKnow*	$:: q(et)$

Note that *wantToKnow* is assumed to expect a question denotation. This is mainly because it is applied to a question denotation in the considered example. If we wanted to be more general, we would assume it to have a polymorphic type $r \rightarrow (e \rightarrow t)$, where r can be instantiated by any result type, in our case t or q. This is what we will in fact do for the predicate *know* in the next subsection.

The semantic expression that results from generating (5.64c) by using the wh-particle (5.65a) in the embedded clause and the yes/no-particle (5.65b) in the matrix clause is the following expression of type q^q_t:

$$\textbf{?}\,((wantToKnow\ \langle\,((admire\ \ \xi_{wh}k.\mathfrak{W}\,x.(person\ x) \wedge (k\ x))\ \ akira)\,\rangle_{wh})\ \ hikaru)$$

Reducing it yields an expression corresponding to the direct question reading of (5.64c):

$$\textbf{?}\,((wantToKnow\ \ \mathfrak{W}\,x.(person\ x) \wedge ((admire\ x)\ \ akira))\ \ hikaru)$$

Generating (5.64c) by using the yes/no-particle in the embedded clause and the wh-particle in the matrix clause, on the other hand, results in the following semantic expression of type t^q_t:

$$\langle\,((wantToKnow\ \textbf{?}\,((admire\ \ \xi_{wh}k.\mathfrak{W}\,x.(person\ x) \wedge (k\ x))\ \ akira))\ \ hikaru)\,\rangle_{wh}$$

Reducing it yields an expression corresponding to the wh-question reading of (5.64c):

$$\mathfrak{W}\,x.(person\ x) \wedge ((wantToKnow\ \textbf{?}\,((admire\ x)\ \ akira))\ \ hikaru)$$

So we can treat in situ wh-phrases in scope marking languages along the same lines as wh-phrases in languages invoking displacement, the only difference being that the delimiter is not introduced by a probe feature but by a specific lexical item, usually a question particle.

5.5.3 In situ wh-phrases

Now we want to look at in situ wh-phrases in languages that do not invoke scope markers, at least no obligatory ones. We will consider Chinese and Hindi, because they show two different possibilities to determine in situ wh-scope. In Chinese, a wh-phrase can take scope at an arbitrary clause level, as long as it does not lead to type clashes, see e.g. (5.66).

(5.66) *Mandarin Chinese*

Zhangsan zhidao shei du-le shu (ma)?
Zhangsan knows who read-ASP books (Q)
'Who does Zhangsan know read books?'
'Zhangsan knows who read books.'

In Hindi, on the other hand, in situ wh-phrases can have scope only over the clause they appear in (cf. Bhatt [10]), as (5.67) illustrates.[1]

(5.67) *Hindi* (Bhatt [10])

Wajahat jaan-taa hai [ki Rima kis-ko pasand kar-tii hai]
Wajahat know-M.SG be that Rima who-ACC like do-F be.PRS.SG
'Wahajat knows who Rima likes.'
* 'Who does Wahajat know Rima likes?'

Both strategies are already familiar to us, because they amount to the distinction between existential wide scope quantifiers on the one hand, and weak and strong quantifiers on the other hand. So we can, in fact, simply employ the quantifier denotations from the previous section for true in situ wh-phrases. For example, the Hindi wh-expression kis ('who') can be assumed to denote:

$$\xi k.\mathfrak{W}x.(person\ x) \wedge (k\ x)$$

Since it uses a Q-flavored shift operator, it will behave like a strong quantifier in capturing the context up to the nearest enclosing Q-flavored delimiter. In the example above this is introduced by the complementizer ki ('that'). Since ki is the head of the embedded clause, the wh-operator can end up taking scope over the embedded clause only.

In Chinese, a true in situ wh-phrase is not clause-bound but may take scope over any clause that allows an interrogative interpretation. To see what this means, let us consider the example from above, here repeated in (5.68a) without the optional question particle, and another simple example given in (5.68b).

(5.68) *Mandarin Chinese*

a. Zhangsan zhidao [shei du-le shu]?
Zhangsan knows who read-ASP books
'Who does Zhangsan know read books?'
'Zhangsan knows who read books.'

b. Zhangsan yiwei [$_{\text{CP}}$ Lisi du-le shenme]
Zhangsan think Lisi read-ASP what
'What does Zhangsan think Lisi read?'

The difference is that the matrix verb in (5.68a) is zhidao ('know'), which allows declarative as well as interrogative complements, whereas the matrix verb in (5.68b) is yiwei ('think'), which allows only for declarative complements. Here are the lexical entries we need for the two examples:

[1] Hindi in fact knows several strategies for question formation. Besides the in situ strategy, it is also invokes displacement and a scope marking strategy, which however are not of concern here.

Form		**Meaning**	
Zhangsan	:: NP	*zhangsan*	:: e
Lisi	:: NP	*lisi*	:: e
shu	:: NP	*books*	:: e
shei	:: NP	$\xi^{\text{free}} k.\mathfrak{W}x.(person\ x) \wedge (k\ x)$	:: e_t^q
shenme	:: NP	$\xi^{\text{free}} k.\mathfrak{W}x.(k\ x)$	:: e_t^q
du-le	:: NP $\rightarrow$ (NP$^<$ $\rightarrow$ VP)	*read*	:: $e \rightarrow (e \rightarrow t)$
zhidao	:: CP $\rightarrow$ (NP$^<$ $\rightarrow$ VP)	*know*	:: $r \rightarrow (e \rightarrow t)$
yiwei	:: CP $\rightarrow$ (NP$^<$ $\rightarrow$ VP)	*think*	:: $t \rightarrow (e \rightarrow t)$
ϵ	:: VP $\rightarrow$ CP	$\lambda p.\langle p \rangle$	:: $t \rightarrow t$

Note that *know* has the polymorphic type $r \rightarrow (e \rightarrow t)$, i.e. allows two instances: $t \rightarrow (e \rightarrow t)$ and $q \rightarrow (e \rightarrow t)$. The wh-phrases use ξ^{free} and will therefore take scope in the same way an existential quantifier would.

Now, generating the first sentence, (5.68a), constructs the following semantic expression as sentence denotation:

$$\langle\, ((know\ \langle\, ((read\ \ books)\ \ \xi^{\text{free}} k.\mathfrak{W}x.(person\ x) \wedge (k\ x))\,\rangle)\ zhangsan)\,\rangle$$

When the reduction rule for shift applies, there are two contexts that can be captured: first, the one up to the inner reset, which results in the expression (5.69a) and second, the one up to the outer reset, which results in the expression (5.69b). They correspond to the two readings that (5.68a) has. Note that in the first case *know* is applied to an interrogative argument of type q, and in the second case it is applied to a declarative argument of type t. Since it allows both, both cases are well-typed.

(5.69) a. $((know\ \ \mathfrak{W}x.(person\ x) \wedge ((read\ \ books)\ x))\ zhangsan)$

b. $\mathfrak{W}x.(person\ x) \wedge ((know\ (read\ \ books))\ zhangsan)$

Now assume (5.68b). Its denotation corresponds to the following expression:

$$\langle\, ((think\ \langle\, ((read\ \ \xi^{\text{free}} k.\mathfrak{W}x.(k\ x))\ lisi)\,\rangle)\ zhangsan)\,\rangle$$

We can do exactly the same: the shift expression either captures the context up to the inner reset or the context up to the outer reset. These two possibilities result in the following two expressions respectively:

(5.70) a. $((think\ \ \mathfrak{W}x.((read\ x)\ lisi))\ zhangsan)$

b. $\mathfrak{W}x.((think\ ((read\ x)\ lisi))\ zhangsan)$

Note that only the second one, (5.70b), is well-typed. In (5.70a), on the other hand, *think* is applied to an expression of type q, although it does not allow interrogative arguments. This type mismatch prevents the wh-expression to take scope over the embedded clause and forces it to take scope over the whole

sentence. And the opposite happens, in fact, if we consider (5.71). It allows only an embedded question reading, because xiang-zhidao ('wonder') allows only interrogative complements.

(5.71) *Mandarin Chinese* (Watanabe [127])

Zhangsan xiang-zhidao [Lisi du-le shenme]?
Zhangsan wonders Lisi read-ASP what
'Zhangsan wonders what Lisi read.'

Before summing up, let us take a digression on the exact source of the delimiter.

5.6 A note on the source of the delimiter

In all examples above, we assumed that the clausal head introduces the Q-flavored delimiter for quantifiers, which straightforwardly accounted for the clause-boundedness of quantifiers. Let us look at whether this is justified or only a simplification of what is really going on.

Hindi seems to support the hypothesis that it is the head of CP that introduces the delimiter. Recall from the previous section that a Hindi in situ wh-phrase without a scope marker can take scope only over the clause it occurs in. Now, Dayal [31] (following Butt [15]) argues that infinitival clauses in Hindi are not CPs but TPs. The CP hypothesis would therefore predict that infinitival clauses do not introduce a delimiter and therefore do not constitute a context that can be captured by an operator. That is, they should be transparent with respect to scope. And this is indeed the case: If an in situ wh-phrase occurs in an infinitival complement clause, it can take wide scope over the matrix clause, see (5.72a). In fact, it has to, for infinitival clauses in Hindi are not a domain for question formation, see (5.72b).

(5.72) *Hindi* (Mahajan [72], Dayal [31])

a. Ram-ne [kis-ko dekh-naa] chaah-aa
Ram-ERG who-ACC see-INF want-PFV
'Who did Ram want to see?'

b. Tum [kyaa kar-naa] jaan-te ho
you.PL what do-INF know-HAB.M.PL be.PRS.2PL
'What do you know to do?'
* 'You know what to do.'

So being a CP seems necessary for delimiting scope. But is it also sufficient? Does every CP restrict the scope of non-existential quantifiers? The answer is not so clearly yes. Consider the English control construction (5.73). The embedded infinitival clause is commonly assumed to be a full clause, i.e. a CP. Nevertheless it is transparent for scope: the quantifier every god can take scope

over the whole sentence, thus scope out of the embedded CP, giving rise to an inverse scope reading.

(5.73) Someone promised to worship every god.

It thus seems that not the categorial status of the CP but rather one of its finiteness plays a crucial role in delimiting scope. How could this be captured? Recall that we modeled syntactic properties that are not connected to simple subcategorization with the help of syntactic features. So a straightforward way to introduce a distinction between finite and infinite CPs would be to consider a feature *fin* which finite CPs carry while infinite CPs do not, and let this feature be interpreted as introducing the delimiter $\langle\,\rangle$ – just like the probe feature $\bullet wh$ was assumed to denote the delimiter $\langle\,\rangle_{wh}$. Then, finite CPs would introduce a delimiter and restrict quantifier scope, and infinite CPs would not. The general picture emerging would be that it is in general features that introduce delimiters, i.e. that restricting scope is a property of lexical items and not of categories or structures.

Whether this picture is indeed the right one is not that easy to decide, though. Assuming a structure like (5.74) for the example above, it could also be that the infinitival CP indeed is a scope domain but that is enough for the universal to outscope the PRO in order to derive inverse scope. Also, the data is not conclusive. Hornstein [54] notes that all speakers of English seem to get inverse scope readings for at least some control verbs. The examples in (5.75), for instance, do in fact allow for the universal to outscope the existential.

(5.74) Someone promised [PRO to worship every god].

(5.75) a. Someone tried to solve every problem.

b. The king asked a guard to escort every perturbator out of the palace.

c. Enlil persuaded a boy to dance with every girl.

I leave the issue of scope in control constructions to further research.

5.7 Summary

The starting point of this chapter was that all expressions are interpreted in their original syntactic position, i.e. where they are merged first, without recourse to displacement or storage mechanisms. This assumption is shared with other 'in situ' theories like Barker's continuation account of quantifier scope [5] and Park's and Steedman's accounts within Combinatory Categorial Grammar ([85],[109]). However, the approach in this chapter differs from Barker, Park and Steedman in what 'in situ' means. They take the syntactic position of an expression to be the surface position, while here it means the position where it originally enters the derivation, independent of whether it is displaced afterwards or not.

The semantic procedure devised for scope construal in this chapter consisted in adding the control operator shift and the corresponding delimiter reset to the language of semantic expressions, together with a reduction rule that allows shift to capture the evaluation context up to an enclosing reset. The shift was assumed to be a crucial part of the denotation of operator expressions, while the delimiter was assumed to be introduced by some feature of a clausal head (a probe feature $\bullet wh$ or possibly a finiteness feature *fin*).

Since an impure expression can only be reduced when it contains both a shift and a reset, shift and reset can be imagined as being a lock and a key: Shift locks an expression by preventing it from being evaluated until a matching reset occurs that unlocks it. In terms of derivations: the evaluation (and thereby the scope construal) of an operator expression is delayed until its scope domain is constructed.

The main part of this chapter was about deriving different scope behavior. To this end, two parameters were introduced: the mode and the flavor of a shift. The mode (weak, strong, or free) determined the evaluation context that can be captured by shift. This allowed to derive the difference between clause-boundedness and free scope, as well as the ability or inability to outscope other operators. The flavor (Q or some feature value) served to keep apart the scope domain of different kinds of operators (quantifiers vs wh-operators, possibly also vs focus operators and the like). Most importantly, the parametrization only concerned evaluation contexts. The rewriting rule for establishing scope itself was the same for all operator expressions.

A summary of the operational semantics responsible for establishing operator scope is given in Figure 5.8.

Here is an overview of the operator expressions that were considered and the control operators that accounted for them.

- strong quantifiers, true in situ wh-phrases (Hindi): ξ_Q^{strong}
- weak quantifiers: ξ_Q^{weak}
- indefinites, true in situ wh-phrases (Chinese): ξ_Q^{free}
- displaced wh-phrases: ξ_{wh}^{strong} (possibly also ξ_{wh}^{weak})
- in situ wh-phrases with a scope marker (Japanese): ξ_{wh}^{free}

One remarkable thing to notice here is that quantifiers on the one hand and wh-phrases on the other hand do not form a natural class. All of them are handled by the same procedure of scope construal and the parametrization of this procedure does not provide a clear cut between quantifiers and wh-phrases. What the two kind of parameters in fact separate is the scope domain (e.g. Q vs wh) and the scope behavior inside this domain (e.g. the (in)ability to outscope other operators). This allows for several ways to cut the operator cake. While operators can differ in the domain they scope over, they can show the same

Figure 5.8: Summary of the operational semantics

Semantic expressions:

$$\begin{aligned}\mathbf{E} ::= \ldots \mid (\xi^{\mathbf{Mode}}_{\mathbf{Flavor}} k :: \tau \to \alpha.\mathbf{E} :: \beta) :: \tau^{\mathbf{Flavor}:\beta}_{\mathbf{Flavor}:\alpha} \\ \mid \langle \mathbf{E} :: \tau \rangle_{\mathbf{Flavor}} :: \tau\end{aligned}$$

$$\begin{aligned}\mathbf{Mode} &::= \text{weak} \mid \text{strong} \mid \text{free} \\ \mathbf{Flavor} &::= Q \mid \mathbf{Value}\end{aligned}$$

Evaluation contexts (where f, f' range over **Flavor**, and **F** ranges over pure expressions):

$$\begin{aligned}\mathbf{D}_f &::= [\,] \mid (\mathbf{E}\ \mathbf{D}_f) \mid (\mathbf{D}_f\ \mathbf{E}) \mid \mathbf{C}_{f'} \\ \mathbf{D}'_f &::= [\,] \mid (\mathbf{E}\ \mathbf{D}'_f) \mid (\mathbf{D}'_f\ \mathbf{F}) \mid \mathbf{C}_{f'} \\ \mathbf{C}_f &::= \mathbf{D}_{f'} \mid \langle \mathbf{C} \rangle_f \\ \mathbf{C} &::= \mathbf{C}_f\end{aligned}$$

Reduction rules:

$$\begin{aligned}C[\langle D_f[\xi^{\text{weak}}_f k.E] \rangle_f] &\rhd C[\langle E\{k \mapsto \lambda x.\langle D_f[x] \rangle_f\} \rangle_f] \\ C[\langle D'_f[\xi^{\text{strong}}_f k.E] \rangle_f] &\rhd C[\langle E\{k \mapsto \lambda x.\langle D'_f[x] \rangle_f\} \rangle_f] \\ C[\langle C_f[\xi^{\text{free}}_f k.E] \rangle_f] &\rhd C[\langle E\{k \mapsto \lambda x.\langle C_f[x] \rangle_f\} \rangle_f]\end{aligned}$$

$$\langle F \rangle_f \rhd F$$

behavior with respect to these domains. This is case for strong quantifiers and displaced wh-phrases, for example. Also, operators can scope over the same domain while showing different behavior with respect to it. This is the case for weak and strong quantifiers, for example.

This still leaves aside a great deal of issues, for quantifier scope is connected to a wide range of other phenomena that receive no discussion in this thesis, such as topicality, polarity, the scope of negation and adverbials, and also processing issues as soon as sentences contain more than two quantifiers. Since all these discussions lie outside the scope of the thesis, they did not concern us. The approach of this chapter is a purely structural one, demonstrating how we can model different scopal behavior in a simple and systematic way, without making any claims about the semantic properties that might cause this behavior.

6

Implementation

In this chapter I give an implementation of the proposed syntactic and semantic procedures. However, it does not cover is the extension for remnant movement in Section 4.5, in order to keep definitions of data types and functions according to how they are used throughout the dissertation.

The language of choice is the purely functional programming language Haskell, because this way the step from formal definition to implementation is particularly small. The interested reader will see that the implementation indeed follows the definitions almost to the letter. For a documentation of Haskell, see the Haskell homepage `www.haskell.org`. For a textbook introduction to Haskell in a linguistic context and its use in natural language semantics, see van Eijck & Unger [118].

Code will be typeset in `typewriter font`. It is available for download at `http://code.google.com/p/synseminterface/`.

6.1 Data types

First I declare a module that defines data types for categories, semantic types, features, and expressions, along with show functions that print them on the screen in a readable form.

```
module Datatypes where
```

Expressions `Exp` are recursively defined as being either simple (pairs of form and meaning) or complex (pairs of a form and another expression). The

only difference to Definition 6 on page 54 is that the meaning dimension is implemented here not as a single meaning but a list of meanings. The list is an easy way to capture non-determinism. This will not play a role for lexical entries, but when constructing denotations containing operators that give rise to more than one scope reading, then all possible readings will be collected in that list of meanings.

```
data Exp = Simple  Form [Meaning]
         | Complex Form Exp
     deriving Eq

instance Show Exp where
  show (Simple  f  ms) = "("++ show f  ++","++ show ms ++")"
  show (Complex e1 e2) = "<"++ show e1 ++","++ show e2 ++">"
```

A form is defined as a string together with a category `Cat` and a list of features `Feat`, according to Definition 4 on page 52, and a meaning is defined as a semantic expression `Term` together with a semantic type.

```
data Form    = Syn String Cat [Feat]   deriving Eq
data Meaning = Sem Term Type           deriving Eq

instance Show Form where
  show (Syn s c fs) = s ++ "::" ++ show c ++" "++ show fs
instance Show Meaning where
  show (Sem t tau)  = show t ++ "::" ++ show tau
```

Syntactic categories `Cat` are given as defined in Definition 1 on page 41. Functional categories $c \rightarrow c'$ are represented as `Slash c c'`. For the diacritic $^<$ marking linearization to the left, I use the data constructor `L`. I also add an additional category `Str`, that stands for strings and is used for forms at the edge of a complex expression (cf. page 58).

```
data Cat  = NP
          | N
          | VP
          | CP
          | L Cat
          | Slash Cat Cat
          | Str
     deriving Eq

instance Show Cat where
  show NP           = "NP"
  show N            = "N"
  show VP           = "VP"
  show CP           = "CP"
  show (L c)        = (show c) ++ "<"
  show (Slash c c') = "("++(show c)++" -> "++(show c')++")"
  show Str          = "String"
```

The feature inventory comprises two classes of features: goal features f and probe features $\bullet f$ (see Definition 5 on page 53), implemented as values with an according data constructor `Goal` or `Probe`. As values I assume `Wh` and `Top`; this can be extended if other features are to be considered as well. Additionally, I add a value `Q`, which corresponds to the flavor Q used in Chapter 5.

```
data Feat  = Goal  Value
           | Probe Value
     deriving (Eq,Show)

data Value = Wh
           | Top
           | Q
     deriving (Eq,Show)
```

Semantic types are given according to Definition 11 on page 99. The case `Cont Type Value Type Type` captures impure types and can, for example, be instantiated as `Cont Entity Wh Bool Question`, representing the type $e^{wh:t}_{wh:q}$.

```
data Type = Entity
          | Bool
          | Question
          | Type :->: Type
          | Cont Type Value Type Type
     deriving Eq

instance Show Type where
  show Entity       = "e"
  show Bool         = "t"
  show Question     = "q"
  show (t1 :->: t2) = "("++ show t1 ++" -> "++ show t2 ++")"
  show (Cont t1 v t2 t3) = show t1
      ++ "_(" ++ show v ++ ":" ++ show t2 ++ ")"
      ++ "^(" ++ show v ++ ":" ++ show t3 ++ ")"
```

Finally, I specify semantic expressions. Here, constants are implemented as strings and variables are implemented as integers. Expressions with operators are applications of a second-order predicate to a function, cf. Section 5.1.

```
data Term = Const String
          | Var Int
          | Lambda Int Term
          | Term :@: Term
          | Op Operator
          | Not Term
          | Term :/\: Term
          | Shift Mode Flavor Int Term
          | Reset Flavor Term
     deriving Eq
```

```
data Operator = Exists
              | ForAll
              | Most
              | W
     deriving (Eq,Show)
```

The expression $(\exists\ \lambda x.(P\ x))$, for example, will correspond to the term:

```
((Op Exists) :@: Lambda 1 ((Const "P") :@: (Var 1)))
```

The construct `Shift Mode Flavor Int Term` represents a semantic expression $\xi_f^m x.E$, and `Reset Flavor Term` represents an expression $\langle E \rangle_f$. `Mode` specifies the mode of the shift (weak, strong, or free), and `Flavor` is defined to be a feature value.

```
data Mode   = Weak
            | Strong
            | Free
     deriving Eq

type Flavor = Value
```

Here is a show function for semantic expressions that displays them in a way close to how they were displayed in the previous chapters. E.g. the term `Reset Wh (Shift Weak Q 2 ((Op Exists) :@: Lambda 1 ((Var 1):@:(Var 2))))` is displayed as `<Shift' 2.Exists 1.(1 2)>_Wh`.

```
instance Show Mode where
  show Weak   = "'"
  show Strong = ""
  show Free   = "^free"

instance Show Term where
  show (Const s)       = s
  show (Var n)         = show n
  show ((Op o) :@: Lambda n t) = show o ++
                                 " "++ show n ++ "." ++ show t
  show (Lambda n t)    = "Lambda "++ show n ++ "." ++ show t
  show (t1 :@: t2)     = "("++ show t1 ++" "++ show t2 ++")"
  show (Not (t1 :/\: (Not t2))) = show t1 ++" => "++ show t2
  show (Not t)         = "(Not " ++ show t ++ ")"
  show (t1 :/\: t2)    = "("++show t1++" And "++show t2++")"
  show (Shift m Q n t) = "Shift" ++ show m ++
                             " " ++ show n ++ "." ++ show t
  show (Shift m f n t) = "Shift" ++ show m ++ "_" ++ show f
                           ++" " ++ show n ++ "." ++ show t
  show (Reset Q t)     = "<" ++ show t ++ ">"
  show (Reset f t)     = "<" ++ show t ++ ">_" ++ show f
```

For convenience, I define implication, as on page 100:

```
impl :: Term -> Term -> Term
impl t1 t2 = Not (t1 :/\: (Not t2))
```

Now that all data types are in place, we can define a lexicon.

6.2 Lexicon

```
module Lexicon where

import Datatypes
```

Lexical entries are simple expressions, i.e. pairs consisting of a syntactic form (`Syn String Cat [Feat]`) and a list of meanings (`Sem Term Type`).

Let us start with noun phrases. Proper names are strings of category `NP`. They do not carry any features that require checking and their meaning corresponds to a nullary constant of type `Entity`.

```
enkidu,gilgamesh,ishtar :: Exp

enkidu    = Simple (Syn "Enkidu"    NP [])
                   [Sem (Const "enkidu")    Entity]
gilgamesh = Simple (Syn "Gilgamesh" NP [])
                   [Sem (Const "gilgamesh") Entity]
ishtar    = Simple (Syn "Ishtar"    NP [])
                   [Sem (Const "ishtar")    Entity]
```

Quantifier expressions are the same with respect to the form dimension but they differ with respect to meaning. They are associated with a denotation of type `Cont Entity Bool Bool` representing e_t^t, which is abbreviated as `gq` (reminiscent of generalized quantifier).

```
everyone,everything,someone,something :: Exp

everyone   = Simple (Syn "everyone"   NP [])
                    [Sem  everyone'   gq]
everything = Simple (Syn "everything" NP [])
                    [Sem  everything' gq]
someone    = Simple (Syn "someone"    NP [])
                    [Sem  someone'    gq]
something  = Simple (Syn "something"  NP [])
                    [Sem  something'  gq]

gq = Cont Entity Q Bool Bool
```

Now the denotations are specified as follows (cf. Figure 5.1 on page 106).

```
someone'    = Shift Free Q 1   ((Op Exists) :@: Lambda 2
                    (((Const "person") :@: (Var 2))
                         :/\: ((Var 1) :@: (Var 2))))
```

```
something' = Shift Free Q 1   ((Op Exists) :@: Lambda 2
                    (((Const "thing")  :@: (Var 2))
                         :/\: ((Var 1) :@: (Var 2))))

everyone'  = Shift Strong Q 1 ((Op ForAll) :@: Lambda 2
               (impl ((Const "person") :@: (Var 2))
                              ((Var 1) :@: (Var 2))))

everything' = Shift Strong Q 1 ((Op ForAll) :@: Lambda 2
               (impl ((Const "thing")  :@: (Var 2))
                             ((Var 1)  :@: (Var 2))))
```

Furthermore, I specify wh-noun phrases, which are of category `NP` and carry a goal feature `Wh`. Their denotation is given below and is of type `Cont Entity Bool Question`, which corresponds to e_t^q. They thus behave like quantificational noun phrases, with the only difference that they change the result type of the context they capture to q (the type we assumed for questions).

```
who,whom,what ::Exp

who   = Simple (Syn "who"  NP [Goal Wh])
               [Sem  who'  (Cont Entity Wh Bool Question)]
whom  = Simple (Syn "whom" NP [Goal Wh])
               [Sem  who'  (Cont Entity Wh Bool Question)]
what  = Simple (Syn "what" NP [Goal Wh])
               [Sem  what' (Cont Entity Wh Bool Question)]

who'  = Shift Strong Wh 1 ((Op W) :@: Lambda 2
          (((Const "person") :@: (Var 2))
               :/\: ((Var 1) :@: (Var 2))))

what' = Shift Strong Wh 1 ((Op W) :@: Lambda 2
          (((Const "thing")  :@: (Var 2))
               :/\: ((Var 1) :@: (Var 2))))
```

For the implementation of verbs, I first introduce abbreviations for their syntactic categories and their semantic types.

```
intransVerb = Slash (L NP) VP
transVerb   = Slash NP (Slash (L NP) VP)

et  = Entity :->: Bool
eet = Entity :->: (Entity :->: Bool)
```

Now, transitive verbs subcategorize for two NPs and denote a two-place predicate constant.

```
sought,liked,met,fought :: Exp

sought = Simple (Syn "sought" transVerb [])
                [Sem (Const "seek")  eet]
```

```
liked  = Simple (Syn "liked"  transVerb [])
                [Sem (Const "like")  eet]
met    = Simple (Syn "met"    transVerb [])
                [Sem (Const "meet")  eet]
fought = Simple (Syn "fought" transVerb [])
                [Sem (Const "fight") eet]
```

Intransitive verbs similarly subcategorize for one NP and denote a one-place predicate constant.

```
left,died :: Exp

left = Simple (Syn "left" intransVerb [])
              [Sem (Const "leave") et]
died = Simple (Syn "died" intransVerb [])
              [Sem (Const "die")   et]
```

Next, I specify lexical entries for determiners (cf. Figure 5.1 on page 106 for `some` and `every`, and Figure 5.7 on page 123 for `which`). They subcategorize for a noun and return a noun phrase. Their semantic type expresses that they require a one-place predicate (a noun denotation) and return a noun phrase denotation with control effects.

```
some,every,which :: Exp

some  = Simple
        (Syn "some"  (Slash N NP) [])
        [Sem  some'  (et :->: Cont Entity Q  Bool Bool)]
every = Simple
        (Syn "every" (Slash N NP) [])
        [Sem  every' (et :->: Cont Entity Q  Bool Bool)]
most  = Simple
        (Syn "most"  (Slash N NP) [])
        [Sem most'   (et :->: Cont Entity Q  Bool Bool)]
which = Simple
        (Syn "which" (Slash N NP) [Goal Wh])
        [Sem  which' (et :->: Cont Entity Wh Bool Question)]

some'  = Lambda 3 (Shift Strong Q 1 ((Op Exists):@: Lambda 2
           (((Var 3) :@: (Var 2))
        :/\: ((Var 1) :@: (Var 2)))))

every' = Lambda 3 (Shift Strong Q 1 ((Op ForAll):@: Lambda 2
       (impl ((Var 3) :@: (Var 2))
             ((Var 1) :@: (Var 2)))))

most'  = Lambda 3 (Shift Strong Q 1 ((Op Most)  :@: Lambda 2
           (((Var 3) :@: (Var 2))
        :/\: ((Var 1) :@: (Var 2)))))
```

```
which' = Lambda 3 (Shift Strong Wh 1 ((Op W)    :@: Lambda 2
           (((Var 3) :@: (Var 2))
        :/\: ((Var 1) :@: (Var 2)))))
```

Nouns are of category `N` and denote one-place predicate constants.

```
king,beast,man,citizen,goddess :: Exp

king    = Simple (Syn "king"    N [])
                 [Sem (Const "king")    et]
beast   = Simple (Syn "beast"   N [])
                 [Sem (Const "beast")   et]
man     = Simple (Syn "man"     N [])
                 [Sem (Const "man")     et]
citizen = Simple (Syn "citizen" N [])
                 [Sem (Const "citizen") et]
goddess = Simple (Syn "goddess" N [])
                 [Sem (Const "goddess") et]
```

Additionally, I specify lexical entries for adjectives. They are of syntactic category `Slash N N`, i.e. subcategorize for a noun and return a noun. Their denotation modifies the denotation of the noun – for the sake of simplicity I assume only intersective adjectives. I start by abbreviating their denotation and semantic type.

```
adjType = (Entity :->: Bool) :->: (Entity :->: Bool)

adjMeaning :: String -> Term
adjMeaning s = Lambda 1 (Lambda 2 (((Var 1)  :@: (Var 2))
                              :/\: ((Const s) :@: (Var 2))))

wild,great,brave :: Exp

wild  = Simple (Syn "wild"  (Slash N N) [])
               [Sem  (adjMeaning "wild") adjType]
great = Simple (Syn "great" (Slash N N) [])
               [Sem (adjMeaning "great") adjType]
brave = Simple (Syn "brave" (Slash N N) [])
               [Sem (adjMeaning "brave") adjType]
```

Finally, I also implement lexical entries for complementizers (cf. Figure 5.2 on page 106).

```
that,epsilon,epsilonWh,whether :: Exp

tt = Bool :->: Bool
tq = Bool :->: Question

that      = Simple
            (Syn "that" (Slash VP CP) [])
            [Sem (Lambda 1 (Reset Q (Var 1))) tt]
```

```
epsilon   = Simple
            (Syn "" (Slash VP CP) [])
            [Sem (Lambda 1 (Reset Q (Var 1))) tt]

epsilonWh = Simple
            (Syn "" (Slash VP CP) [Probe Wh])
            [Sem (Lambda 1 (Reset Wh (Reset Q (Var 1)))) tt]

whether   = Simple
            (Syn "whether" (Slash VP CP) [])
            [Sem (Lambda 1 (Reset Wh (Reset Q (Var 1)))) tq]
```

6.3 Displacement

```
module Displacement where

import Datatypes
import Eval
import Data.List ((\\),intersect)
```

I start by defining some simple auxiliary functions that will be needed later on: `feats` and `featsForm` for accessing the feature list of an expression and a form, and `edge` that collects the forms at the edge of an expression.

```
feats :: Exp -> [Feat]
feats (Simple  f _) = featsForm f
feats (Complex _ e) = feats e

featsForm :: Form -> [Feat]
featsForm (Syn _ _ fs) = fs

edge :: Exp -> [Form]
edge (Simple _ _)    = []
edge (Complex e1 e2) = e1 : (edge e2)
```

In the following, two parameters will be needed. `SpecPar` specifies whether all forms that check a feature can be concatenated (`Multiple`) or whether this is possible only for one (`Single`). On page 67 of Chapter 4 this was interpreted as whether a language allows multiple specifiers or uniquely filled specifiers only. The second parameter, `WhPar`, specifies whether a language leaves all wh-phrases in situ like Japanese (`InSitu`), requires to front all wh-phrases like Bulgarian (`Fronting`), or fronts one wh-phrase and leaves other wh-phrases in situ like English (`Mixed`).

```
data SpecPar = Single | Multiple
data WhPar   = InSitu | Fronting | Mixed
```

The function `split` splits a simple expression into a complex one, according to its definition on page 58. Its exact outcome depends on the parameter `WhPar`:

it associates the phonological content of the expression with the edge in the case of `Fronting` (cf. page 67), with the nucleus in the case of `InSitu` (cf. page 69), and it allows both possibilities in the case of `Mixed` (cf. page 63). The indeterminism in the mixed case is captured by returning a list of all possibilities.

```
split :: WhPar -> Exp -> [Exp]

split Fronting (Simple (Syn s c fs) sem) =
  [Complex (Syn s  Str fs) (Simple (Syn "" c []) sem)]

split InSitu (Simple (Syn s c fs) sem) =
  [Complex (Syn "" Str fs) (Simple (Syn s  c []) sem)]

split Mixed  (Simple (Syn s c fs) sem) =
  [Complex (Syn s  Str fs) (Simple (Syn "" c []) sem),
   Complex (Syn "" Str fs) (Simple (Syn s  c []) sem)]
```

The function `merge` implements Definition 7 on page 57. It takes an instantiation of the parameter `WhPar` as input because it needs to hand it to the function `split` in case splitting is necessary.

```
merge :: WhPar -> Exp -> Exp -> [Exp]

merge p e1@(Simple form1 ms1) e2@(Simple form2 ms2)
 | feats e2 /= [] = concat $ map (merge p e1) (split p e2)
 | otherwise      = [Simple form (funcapp ms1 ms2) |
                              form <- (form1 `oplus` form2)]

merge p e1@(Complex form exp) e2@(Simple _ _)
 | feats e2 /= [] = concat $ map (merge p e1) (split p e2)
 | otherwise      = [Complex form e | e <- merge p exp e2]

merge p e1 (Complex form exp)
                  = [Complex form e | e <- merge p e1 exp]
```

Next I give the auxiliary functions that `merge` uses: `oplus`, an implementation of $\oplus$, and `funcapp`, an implementation of functional application.

```
oplus :: Form -> Form -> [Form]
oplus (Syn s1 (Slash (L c) c') fs1) (Syn s2 c2 _)
        | c == c2   = [Syn (s2++" "++s1) c' fs1]
        | otherwise = []
oplus (Syn s1 (Slash c c') fs1) (Syn s2 c2 _)
        | c == c2   = [Syn (s1++" "++s2) c' fs1]
        | otherwise = []
oplus _ _ = []

funcapp :: [Meaning] -> [Meaning] -> [Meaning]
funcapp ms1 ms2 = concat $ map (uncurry fa) (zip ms1 ms2)
```

```
  where
     fa :: Meaning -> Meaning -> [Meaning]
     fa (Sem t1 tau1) (Sem t2 tau2) =
        [Sem t (infer tau t) | t   <- evaluate (t1 :@: t2),
                               tau <- tau1 `at` tau2 ]
```

In the definition of `funcapp`, `at` is a function that determines the type of the result of functional application, in the case of control effects according to the typing rules on page 101. If the input types do not match, it returns the empty list and as a consequences `funcapp` will also not return a result. The function `infer` removes encoded control effects in the type in case the control transfer was already performed. It uses an auxiliary test `hasShift`, which will make sense only after we have seen the implementation of the semantics in the next section.

```
at :: Type -> Type -> [Type]
at t (Cont t' f a b) = [ Cont tau f a b | tau <- t `at` t' ]
at (Cont t' f a b) t = [ Cont tau f a b | tau <- t' `at` t ]
at (t1 :->: t2)  t | t1 == t   = [t2]
                   | otherwise = []

infer :: Type -> Term -> Type
infer tau@(Cont t1 v t2 t3) term
       | hasShift v term = Cont (infer t1 term) v t2 t3
       | otherwise       = infer t1 term
infer tau ((Op W) :@: _) = Question
infer tau _              = tau

hasShift :: Flavor -> Term -> Bool
hasShift v (Shift _ f _ t) | f == v    = True
                           | otherwise = hasShift v t
hasShift v (Lambda _ t) = hasShift v t
hasShift v (t1 :@:  t2) = hasShift v t1 || hasShift v t2
hasShift v (t1 :/\: t2) = hasShift v t1 || hasShift v t2
hasShift v (Not t)      = hasShift v t
hasShift v (Reset _ t)  = hasShift v t
hasShift _ _            = False
```

The following function `remerge` implements Definition 9 on page 60. It uses additional auxiliary functions that are defined below.

```
remerge :: SpecPar -> Exp -> Exp
remerge _ e@(Simple  _ _) = e
remerge p e@(Complex _ _) = deleteEmpty $
                            (a `without` cs)
                            `plus`
                            (check p cs (e `minus` a))
   where a  = head (edge e)
         cs = (featsForm a) `intersect` (mirror $ feats e)
```

The auxiliary function `minus` removes a form from the edge of an expression, and `without` removes features from a form.

```
minus :: Exp -> Form -> Exp
minus e@(Simple _ _) f = e
minus (Complex f e) f' | f == f'   = e
                       | otherwise = Complex f (e `minus` f')

without :: Form -> [Feat] -> Form
without (Syn s c fs) fs' = Syn s c (fs \\ fs')
```

The function `plus` concatenates a form with the form of the nucleus of an expression, just like + in the definition of **remerge**.

```
plus :: Form -> Exp -> Exp
plus (Syn s _ _) (Simple (Syn s' c' fs') sem) =
     Simple (Syn (s++" "++s') c' fs') sem
plus form (Complex f exp) = Complex f (form `plus` exp)
```

The function `check` takes an expression and the relevant features, and plays the role of checking all instances of these features at its edge, concatenating the edge expressions if possible and dependening on the parameter `SpecPar`. It uses an auxiliary function `mirror` for turning goal features into probe features and vice versa.

```
check :: SpecPar -> [Feat] -> Exp -> Exp

check Single cs (Simple (Syn s c fs) sem) =
  Simple (Syn s c (fs \\ mirror cs)) sem
check Single cs (Complex f e) = Complex (f `without` cs)
                                        (check Single cs e)

check Multiple cs (Simple (Syn s c fs) sem) =
  Simple (Syn s c (fs \\ mirror cs)) sem
check Multiple cs (Complex f e)
    | fcs == [] =  Complex f (check Multiple cs e)
    | otherwise = if   all (`elem` cs) fs
                  then f `plus` (check Multiple cs e)
                  else Complex (f `without` cs)
                               (check Multiple cs e)
  where   fs  = featsForm f
          fcs = cs `intersect` fs

mirror :: [Feat] -> [Feat]
mirror [] = []
mirror ((Probe v):fs) = (Goal  v):(mirror fs)
mirror ((Goal  v):fs) = (Probe v):(mirror fs)
```

Finally, `deleteEmpty` deletes empty forms at the edge.

```
deleteEmpty :: Exp -> Exp
deleteEmpty e@(Simple _ _) = e
deleteEmpty (Complex form e)
              | empty form = deleteEmpty e
              | otherwise  = Complex form (deleteEmpty e)

empty :: Form -> Bool
empty (Syn "" _ []) = True
empty _             = False
```

6.4 Operator scope

Now we turn to an implementation of the operational semantics. The main goal is to have an evaluation function that takes a term and reduces it according to the reduction rules specified in Chapter 5.

```
module Eval where

import Datatypes
import Data.List ((\\),nub)
```

The evaluation function will be based on Oleg Kiselyov's lambda calculator algorithm in the rewritten form proposed by Chung-Chieh Shan, see `http://okmij.org/ftp/Haskell/Lambda_calc.lhs`. Ignoring control operators for the moment, the evaluation function could be implemented as `eval'`.

```
eval' :: Term -> Term
eval' c@(Const _)  = c
eval' v@(Var _)    = v
eval' (Not t)      = Not (eval' t)
eval' (t1 :/\: t2) = (eval' t1) :/\: (eval' t2)
eval' (Lambda n t) = etaReduce $ Lambda n (eval' t)
eval' ((Lambda n t1) :@: t2) = eval' $ substitute t1 n t2
eval' (t1@(_:@:_):@:t2) = case eval' t1 of
                            t@(Lambda _ _) -> eval' (t:@:t2)
                            t              -> t:@:(eval' t2)
eval' (t1 :@: t2) = t1 :@: (eval' t2)
eval' t           = t
```

Where `substitute` implements capture avoiding substitution (as given later) and `etaReduce` implements eta-reduction as follows. It uses a test `notFree`, which returns true if there are no free occurrences of a particular variable in a particular term.

```
etaReduce :: Term -> Term
etaReduce t@(Lambda n (t' :@: (Var n')))
  | n == n' && notFree n t'  = t'
  | otherwise                = t
etaReduce t                  = t
```

```
notFree :: Int -> Term -> Bool
notFree n (Var n')      = n /= n'
notFree n (Not t)       = notFree n t
notFree n (Lambda n' t) | n == n'   = True
                        | otherwise = notFree n t
notFree n (t1 :/\: t2) = notFree n t1 && notFree n t2
notFree n (t1 :@:  t2) = notFree n t1 && notFree n t2
notFree _ _             = True
```

I will explain the details of the evaluation later when giving the final evaluation fuction we will use.

The function `eval'` works fine for pure expression but we cannot simply add a case for impure ones. The reason is that the evaluation rule for impure expressions, i.e. expressions containing shifts, is not so much about manipulating the content in our data structure `Term` but rather about manipulating the location of a subexpression. (Recall how the reduction rule for shift captured the context and plugged it into the ξ-expression, thereby transfering a part of the expression to a position above the captured context.) In order to manipulate locations, I make use of Huet's idea of a *zipper* (cf. [56]) and let our evaluation function not operate on expression but on expressions with zipper. The zipper is an idiom for traversing a data structure and manipulating locations in it in a non-destructive way. It uses the idea of storing information about the location of a substructure (a subexpression in our case) with the help of contexts. To this end, we define a data structure `Context` in parallel to our definition for `Term`.

```
data Context = Hole
             | CAppL Context Term
             | CAppR Term Context
             | CAndL Context Term
             | CAndR Term Context
             | CNot Context
             | CLambda Int Context
             | CReset Flavor Context
           deriving (Eq,Show)
```

A list of such contexts is called *thread*. A zipper is specified as a thread together with a term. The term is some subexpression t of a bigger expression t', and the thread encodes the location of t in t'. The thread can thus be thought of as storing the path that was traversed in t' in order to get to t's location.

```
type Thread = [Context]
type Zipper = (Thread,Term)
```

As an example, consider the example expression $(\lambda x.\neg(\mathit{fish} \wedge x)\ \mathit{chips})$:

```
exampleTerm = (Lambda 1 (Not ((Const "fish") :/\: (Var 1))))
          :@: (Const "chips")
```

Focusing on `Var 1` as a subterm of `exampleTerm`, we can represent it as a zipper, where its location is stored in the thread.

```
exampleZipper = ([CAndR (Const "fish") Hole,
                  CNot Hole,
                  CLambda 1 Hole,
                  CAppL Hole (Const "chips")],
                 Var 1)
```

Now we can go from a zipper to a term by unwinding the thread.

```
unwind :: Zipper -> Term
unwind ([],t) = t
unwind ((CAppR t _)  :ts,t') = unwind (ts,t  :@:  t')
unwind ((CAppL _ t)  :ts,t') = unwind (ts,t' :@:  t )
unwind ((CAndR t _)  :ts,t') = unwind (ts,t  :/\: t')
unwind ((CAndL _ t)  :ts,t') = unwind (ts,t' :/\: t )
unwind ((CNot _)     :ts,t') = unwind (ts,Not t')
unwind ((CLambda n _):ts,t ) = unwind (ts,Lambda n t)
unwind ((CReset f _) :ts,t ) = unwind (ts,Reset  f t)
```

Going back to our example, `unwind exampleZipper` would return the expression `exampleTerm` we started with. Later we will use a function `unwind1` that unwinds not the whole thread but only its head (i.e. is not recursive).

```
unwind1 :: Zipper -> Zipper
unwind1 z@([],t) = z
unwind1 ((CAppR t _)  :ts,t') = (ts,t  :@:  t')
unwind1 ((CAppL _ t)  :ts,t') = (ts,t' :@:  t )
unwind1 ((CAndR t _)  :ts,t') = (ts,t  :/\: t')
unwind1 ((CAndL _ t)  :ts,t') = (ts,t' :/\: t )
unwind1 ((CNot _)     :ts,t') = (ts,Not t')
unwind1 ((CLambda n _):ts,t ) = (ts,Lambda n t)
unwind1 ((CReset f _) :ts,t ) = (ts,Reset  f t)
```

Now we intersperse the evaluation function `eval'` from above with a zipper. This means that the type of `eval` is not `Term -> Term` but `Zipper -> [Zipper]`. (Returning not a single zipper but a list of zippers is again a way to implement non-determinism.) Every time we move to a subexpression, we store its context and every time we move up again, we unwind this context.

```
eval :: Zipper -> [Zipper]
```

Constants and variables evaluate to themselves, so `eval` simply returns the zipper it got as input.

```
eval z@(_,Const _) = [ z ]
eval z@(_,Var _)   = [ z ]
```

For lambda expressions, we evaluate the body. The only difference to the corresponding clause of `eval'` above is that here we do not eta-reduce the result. We will perform eta-reduction later on the whole expression. This

is because otherwise we would reduce $(\exists\ \lambda x.(P\ x))$ to $(\exists\ P)$, for example. Interspersing the clause with the zipper (i.e. stacking the context and later unwinding it) yields the following.

```
eval (thread,Lambda n t) = map unwind1 $
                           eval ((CLambda n Hole):thread,t)
```

The reduction of applications are handled by distinguishing three cases, like with `eval'`. The first one is the case of an expression $(\lambda x.t_1\ \ t_2)$. This is reduced by substituting x in t_1 by t_2.

```
eval (th,(Lambda n t1):@:t2) = eval (th,substitute t1 n t2)
```

The second case is that of stacked applications. Since we do not want to fix a certain evaluation order, the expression is evaluated twice: once from left to right and once from right to left. For pure expressions, this will always yield the same result, but for impure expressions both directions may differ (e.g. may result in different scopings). The way the reduction itself works is parallel to the clause in `eval'`.

```
eval (thread,t1@(_:@:_) :@: t2) =
     (concat $ map (evalL . unwind1) $
               eval ((CAppR t1 Hole):thread,t2))
     ++
     (concat $ map (evalR . unwind1) $
               eval ((CAppL Hole t2):thread,t1))

  where

     evalL (th,y) = case y of
                    (a:@:b) -> map unwind1 $
                               eval ((CAppL Hole b):th,a)
                    t       -> eval (th,t)
     evalR (th,y) = case y of
                    (a@(Lambda _ _):@:b) -> eval (th,a:@:b)
                    (a :@: b) -> map unwind1 $
                                 eval ((CAppR a Hole):th,b)
                    t         -> eval (th,t)
```

The third case comprises all remaining possibilities.

```
eval (thread,t@(t1:@:t2)) = map unwind1 $
                            eval ((CAppR t1 Hole):thread,t2)
```

The reduction of a negated term is straightforward.

```
eval (thread,Not t) = map unwind1 $
                      eval ((CNot Hole):thread,t)
```

With conjunctions, free evaluation order can be emulated like in the case of applications, namely by executing both left-to-right and right-to-left order and collecting the results in a list.

```
eval (thread,t1 :/\: t2) = (concat $ map (evalR . unwind1) $
                            eval ((CAndL Hole t2):thread,t1))
                           ++
                           (concat $ map (evalL . unwind1) $
                            eval ((CAndR t1 Hole):thread,t2))

   where

    evalL (th,y) = case y of
                   (a :/\: b) -> map unwind1 $
                                 eval ((CAndL Hole b):th,a)
                   t          -> eval (th,t)
    evalR (th,y) = case y of
                   (a :/\: b) -> map unwind1 $
                                 eval ((CAndR a Hole):th,b)
                   t          -> eval (th,t)
```

Our fragment will not make use of this. If we are sure that no control effects occur in either conjunct, a simpler evaluation clause could be given:

```
eval (th,t1 :/\: t2) = [ (th,x :/\: y) | (_,x) <- eval (th,t1),
                                         (_,y) <- eval (th,t2) ]
```

Now let us turn to the control operators. The reduction of a term enclosed by a reset depends on whether the term is pure or not, i.e. whether it contains shift operators of the same mode or not. (The test `pure` will be defined below.) If it is pure, we simply drop the reset and evaluate the term. If it is impure, on the other hand, we keep the reset by storing it in the context and unwinding it after reducing the term.

```
eval (thread,Reset f t)
     | pure f t  = eval (thread,t)
     | otherwise = concat $ map (eval . unwind1) $
                   eval ((CReset f Hole):thread,t)
```

Finally, we implement the shift reduction rule as follows.

```
eval z@(thread,Shift m f n t)
  | noResets f thread = [ z ]
  | otherwise         =  map unwind1 $ concat
                  [ eval (th2,substitute t n (reify f th1)) |
                    (th1,th2) <- filter (admissible m f) $
                                 splitt f [] thread ]

eval z@(_,_) = [ z ]
```

The expression `substitute t n (reify f th1)` corresponds to the body of the shifted expression where the variable bound by the operator shift (`n`) is replaced by the reified context. The context is every possible context `th1` up to an enclosing reset that matches in flavor. These contexts are determined by a function `splitt` that splits the thread in position of a matching reset, and

then filtered with `admissible`, depending on the mode of the shift. For weak and strong shifts, only the context up to the nearest enclosing reset is allowed (for weak shifts moreover only contexts not containing other shifts), while for free shifts, all contexts are possible.

```
splitt :: Flavor -> Thread -> Thread -> [(Thread,Thread)]
splitt _ _ [] = []
splitt f thread (c@(CReset f' _):cs)
  | f == f'   = (thread,(c:cs)) : (splitt f (thread++[c]) cs)
  | otherwise = splitt f (thread++[c]) cs
splitt f thread (c:cs) = splitt f (thread++[c]) cs

admissible :: Mode -> Flavor -> (Thread,Thread) -> Bool
admissible Weak   f ((c:cs),_) = noResets f cs
                              && pureThread f cs
admissible Strong f ((c:cs),_) = noResets f cs
admissible _ _ _               = True
```

The function `admissible` uses the auxiliary functions `noResets`, which returns true if a thread does not contain resets of a certain flavor, and `pureThread`, which checks whether a thread contains shifts of a certain flavor. This serves the purpose of determining whether the evaluation is skipping over a shift expression that should actually be evaluated first. (For convenience, `pureThread` only checks arguments of applications, because these are the relevant cases for us. However, it is easy to extend the function to all other cases, if needed.) The latter uses another auxiliary function `pure`, which checks whether a term is pure with respect to a particular flavor.

```
noResets :: Flavor -> Thread -> Bool
noResets f thread = null $ filter (== CReset f Hole) thread

pureThread :: Flavor -> Thread -> Bool
pureThread f []                = True
pureThread f ((CAppL c t):cs)  = pure f t
                              && pureThread f cs
pureThread f ((CReset _ c):cs) = pureThread f [c]
                              && pureThread f cs
pureThread f (_:cs)             = pureThread f cs

pure :: Flavor -> Term -> Bool
pure f (Lambda _ t)      = pure f t
pure f (t1 :@: t2)       = pure f t1 && pure f t2
pure f (Not t)           = pure f t
pure f (t1 :/\: t2)      = pure f t1 && pure f t2
pure f (Reset _ t)       = pure f t
pure f (Shift _ f' _ _)  = f /= f'
pure _ _                 = True
```

Reifying the context as a function is taken care of by the function `reify`, defined as follows. The auxiliary functions `variablesC` and `variables` traverse

context lists and terms, respectively, in order to collect all occuring variables, from which the maximum is chosen (or zero if the list of occuring variables is empty) and added by 1. The resulting integer is a fresh variable `fresh` that can be used for reifying the context without risking variable clashes.

```
reify :: Flavor -> Thread -> Term
reify f thread = Lambda fresh
                 (Reset f (unwind (thread,Var fresh)))
  where  fresh = (maxList $ concat $ map varsC thread) + 1

varsC :: Context -> [Int]
varsC Hole          = []
varsC (CAppL _ t)   = variables t
varsC (CAppR t _)   = variables t
varsC (CAndL _ t)   = variables t
varsC (CAndR t _)   = variables t
varsC (CLambda n _) = [n]
varsC (CNot _ )     = []
varsC (CReset _ _)  = []

variables :: Term -> [Int]
variables (Var n)          = [n]
variables (Not t)          = variables t
variables (t1 :/\: t2)     = variables t1 ++ variables t2
variables (t1 :@:  t2)     = variables t1 ++ variables t2
variables (Lambda n body) = n : (variables body)
variables (Shift _ _ n t) = n : (variables t)
variables (Reset _ t)      = variables t
variables _                = []

maxList :: [Int] -> Int
maxList [] = 0
maxList xs = maximum xs
```

Now we also define a top-level function `evaluate` of type `Term -> [Term]`, that evaluates the input expression by initiating `eval` with an empty thread, picking the resulting expression (the thread is empty after application of `eval` because every time a context is appended to the thread, it is unwound afterwards), performing eta-reduction and removing duplicates.

```
evaluate :: Term -> [Term]
evaluate t = map eta $ nub $ map snd $ eval ([],t)

eta :: Term -> Term
eta t@((Op _) :@: Lambda _ _) = t
eta t@(Lambda n (t' :@: (Var n')))
    | n == n' && notFree n t' = t'
    | otherwise               = t
eta (t1 :@:  t2)              = (eta t1) :@:  (eta t2)
```

```
eta (t1 :/\: t2)              = (eta t1) :/\: (eta t2)
eta (Not t)                   = Not (eta t)
eta (Reset f t)               = Reset f (eta t)
eta (Shift m f n t)           = Shift m f n (eta t)
eta t                         = t
```

The only thing that is still missing is an implementation of capture-avoiding substitution. The function `substitute` that achieves this is as one would expect. Here, `substitute t1 n t2` means that the variable `n` is substituted in `t1` for `t2`.

```
substitute :: Term -> Int -> Term -> Term
```

Constants are not substituted, and a variable is substituted if it is the input variable.

```
substitute c@(Const _) _ _             = c
substitute v@(Var n1) n2 t | n1 == n2  = t
                           | otherwise = v
```

For negated terms, conjunctions, and applications, the substitution is simply passed to the subexpressions.

```
substitute (Not t') n t     = Not $ substitute t' n t
substitute (t1 :/\: t2) n t = (substitute t1 n t)
                          :/\: (substitute t2 n t)
substitute (t1 :@:  t2) n t = (substitute t1 n t)
                           :@: (substitute t2 n t)
```

In principle the same happens with lambda expressions, but it needs to be ensured that variables in the expression we are substituting do note accidentally get bound by the lambda. If this happens (which is the second case below), the variable is renamed in the lambda expression.

```
substitute (Lambda n' body) n t
    | n' == n = Lambda n' body
    | n' `elem` (variables t) =
         let
            fresh = (maximum (n':(variables body))) + 1
            body' = substitute body n' (Var fresh)
         in
            substitute (Lambda fresh body') n t
    | otherwise = Lambda n' (substitute body n t)
```

And pretty much the same is done for expressions with a shift operator: A substitution in an expression enclosed by a reset is simply passed on, and operators can be ignored althogether because they do not contain variables.

```
substitute (Shift m f n body) n' t
     | n' `elem` (variables t) =
          let
             fresh = (maxList $ (n':(variables body))) + 1
             body' = substitute body n' (Var fresh)
          in
             substitute (Shift m f fresh body') n' t
     | otherwise = Shift m f n (substitute body n' t)

substitute (Reset f t') n t = Reset f (substitute t' n t)

substitute t@(Op _) _ _ = t
```

This concludes the implementation of the semantic mechanism employed in Chapter 5.

6.5 Front end

Let us now move to implementing a front end that facilitates playing with the implementation.

```
module FrontEnd where

import Datatypes
import Lexicon
import Displacement
import Eval
import Data.Char (toLower,isSpace)
import Data.List ((\\))
```

For example, when we want to build the sentence

Which great man died?

we have to apply **merge** to great and man, then merge the result with which, and finally merge the resulting NP with the verb died. We would have to input the following:

```
remerge Single (merge Mixed epsilonWh (merge Mixed died
               (merge Mixed which (merge Mixed great man))))
```

In this section I define some auxiliaries so we can instead input the following, which is not only clearer but also prevents us from specifying different parameters inside one sentence:

```
build Mixed Single "(RM (M1 epsilonWh (M1 died (M1 which
                                       (M1 great man)))))"
```

Where `M1` represents **merge** and `RM` represents **remerge**.

To this end, I define a datatype `SO` (reminiscent of syntactic object) that specifies the operation that is to be applied to its arguments: `M1` for merging a one-place function with an argument, `M2` for merging a two-place function with two arguments, and `RM` for remerge. Additionally, `LI` represents lexical items.

```
data SO = LI String
        | M1 SO SO
        | M2 SO SO SO
        | RM SO
     deriving (Eq,Show)
```

The first step now is to parse an input string and construct the corresponding SO. This is actually only to avoid having to write `(LI "word")` and instead be able to simply write `word`.

```
parse :: String -> SO
parse s@('(' : xs) = parseCon ((init . tail)
                               (upToClosingBracket s))
parse s            = LI s

parseCon :: String -> SO
parseCon s = case op of
                 "M1" -> M1 (parse func) (parse arg1)
                 "M2" -> M2 (parse func) (parse arg1)
                                         (parse arg2)
                 "RM" -> RM (parse func)
  where op   = head (wordz s)
        func = head (tail (wordz s))
        arg1 = head (tail (tail (wordz s)))
        arg2 = last (wordz s)

wordz :: String -> [String]
wordz [] = []
wordz s | head s == '(' = (upToClosingBracket s) :
            (wordz $ remove (length (upToClosingBracket s)+1) s)
        | otherwise     = firstWord :
            (wordz $ remove (length firstWord + 1) s)
  where firstWord = fst $ break (==' ') s
```

For example, calling `wordz` on the string `"(wise man) enki"` returns the list `["(wise man)","enki"]`.

```
remove :: Int -> String -> String
remove n s = s \\ (take n s)

upToClosingBracket :: String -> String
upToClosingBracket s = head [ x | x <- prefixes s,
                                  bracketsFine x ]
```

```
prefixes s = [ ys | ys <- [take i s | i <- [1..(length s)]]]

bracketsFine x = (count '(' x) == (count ')' x)

count y [] = 0
count y (z:zs) | y == z    = 1 + count y zs
               | otherwise = count y zs
```

The next step is to translate the SO we created into the corresponding applications of operations. For `LI`s, the function `trans` simply looks up the string in the lexicon (see below for `lookUp`). For SOs encoding merge or remerge operations, `trans` is defined recursively. It uses two auxiliary functions that stack the operations that have to be applied.

```
trans :: SO -> WhPar -> SpecPar -> [Exp]
trans (LI string)      _  _    = [lookUp string]
trans (M1 so1 so2)     wh spec = glueM wh (trans so1 wh spec)
                                          (trans so2 wh spec)
trans (M2 so1 so2 so3) wh spec =
                 glueM wh (glueM wh (trans so1 wh spec)
                                    (trans so2 wh spec))
                          (trans so3 wh spec)
trans (RM so) wh spec = glueRM spec (trans so wh spec)
```

The auxiliary functions `glue` and `glueRM` are specified as follows.

```
glueM :: WhPar -> [Exp] -> [Exp] -> [Exp]
glueM p xs ys = [ z | x <- xs,
                      y <- ys,
                      z <- merge p x y ]

glueRM :: SpecPar -> [Exp] -> [Exp]
glueRM p xs = map (remerge p) xs
```

Finally, I define a top-level function `build` that takes two parameters and a string as input, and returns the successful results of applying the operations as specified in the input string.

```
build :: WhPar -> SpecPar -> String -> [Exp]
build wh spec string = filter converged $
                       trans (parse string) wh spec

converged :: Exp -> Bool
converged (Simple _ _) = True
converged _            = False
```

As a last thing, we only need a function `lookUp` for looking up strings in the lexicon. Its implementation is very straightforward because the expressions in the lexicon were named like the string with which we refer to them.

```
lookUp :: String -> Exp
```

```
lookUp "enkidu"     = enkidu
lookUp "gilgamesh"  = gilgamesh
lookUp "ishtar"     = ishtar
lookUp "who"        = who
lookUp "whom"       = whom
lookUp "what"       = what
lookUp "king"       = king
lookUp "beast"      = beast
lookUp "man"        = man
lookUp "citizen"    = citizen
lookUp "goddess"    = goddess
lookUp "that"       = that
lookUp "epsilon"    = epsilon
lookUp "sought"     = sought
lookUp "liked"      = liked
lookUp "met"        = met
lookUp "fought"     = fought
lookUp "left"       = left
lookUp "died"       = died
lookUp "great"      = great
lookUp "wild"       = wild
lookUp "some"       = some
lookUp "every"      = every
lookUp "which"      = which
lookUp "epsilonWh"  = epsilonWh
lookUp "everyone"   = everyone
lookUp "someone"    = someone
lookUp "everything" = everything
lookUp "something"  = something
```

Now we can use the frontend by feeding `build` with strings like the following:

```
s1 = "(M1 that (M2 fought (M1 some beast) gilgamesh))"
s2 = "(RM (M1 epsilonWh (M2 liked whom enkidu)))"
s3 = "(RM (M1 epsilonWh (M1 died (M1 which (M1 great man)))))"
s4 = "(RM (M1 epsilonWh (M2 liked what who)))"
s5 = "(RM (M1 epsilonWh (M2 fought whom who)))"
s6 = "(M1 epsilon (M2 liked something everyone))"
```

For a language like English, we would use `build Mixed Single`, while for most Slavic languages we would use `build Fronting Multiple`, and for Japanese and other in situ languages we can use both `build InSitu Single` and `build InSitu Mixed`. Note that the semantic procedure for scope construal applies automatically because the evaluation of the constructed semantic expression is part of the definition of **merge**.

7

Concluding remarks and future perspectives

Finally, I will wrap up by briefly recapitulating the main points of the thesis and considering its implications for the syntax/semantics interface.

The main subject of this thesis was the syntax and semantics of non-local dependencies, focusing on wh-displacement and operator scope. These dependencies give rise to the question that lies at the heart of this thesis: What is the relationship between form and meaning and to which extent do syntax and semantics operate in parallel?

A common view of formal grammar theories is to impose a strict correspondence between syntax and semantics and assume that syntactic displacement and semantic scope construal go hand in hand, or even are two sides of the same coin. This seems to be half right and half wrong, for the syntactic position of a displaced operator expression in some cases does coincide with its semantic scope position and in some cases does not. From the strict correspondence point of view, the observed parallels are expected and do not require further ado. The mismatches, on the other hand, constitute exceptions and require the adjustment of either the syntactic rule for displacement or the semantic rule for establishing scope.

In this thesis I set out to explore the opposite approach: displacement and scope construal are two distinct mechanisms. That is, the mismatches between the syntactic position of an operator expression and its semantic scope position are the normal case, parallels are the exception – they emerge simply as a result

of the general relation between syntax and semantics.

The overall picture of the syntax/semantics interface drawn in this thesis rests on the assumption that grammar consists of two parts: a core system for establishing local dependencies, with syntax and semantics operating in parallel, and extensions to this core system for establishing non-local dependencies, with syntax and semantics operating independently of one another. I proposed that it is those extensions that are responsible for displacement and operator scope. More specifically, I took displacement to be a syntactic procedure that does not have a semantic counterpart, and scope construal to be a semantic procedure that does not have a syntactic counterpart.

Chapter 2 started by carving out the core system. Expressions were taken to be form-meaning pairs and combining such expressions consisted of two operations: string concatenation on the form side, serving to fulfill subcategorization requirements, and functional application on the meaning side, serving to fulfill argument requirements. Recall that all expressions that could be built in this way were pairs of a well-typed syntactic form and a corresponding well-typed semantic term. That is, there was a hardwired connection between form and meaning that ensured them to be synchronized.

This core system was then extended by procedures operating only on one of the two dimensions. Displacement, on the one hand, operated on the form dimension only. It was driven by the need to check syntactic features in a position different from the one where the expression originated. Scope construal, on the other hand, operated on the meaning dimension only, resulting in the establishment of logical scope in a position different from where the expression was originally interpreted. There were cases where both overlap, because clausal heads play two roles: they trigger displacement and they delimit scope. But there was no intrinsic tie between them. Displacement was driven only by the need to fulfil formal requirements, and semantics was blind to the results this fulfilment led to. That is, syntax and semantics parted company. Now, what happened to the bond between form and meaning that we had in the base grammar?

Let us first look at the case of syntax parting company with semantics. Expressions were extended to not only comprise simple form-meaning pairs but also complex expressions consisting of a form-meaning pair together with a number of forms at the edge. Remember that the complex expressions inherited its properties (the syntactic type, the features, as well as the meaning) from the underlying form-meaning pair. That is, carrying along forms at the edge had no effect on the properties of the whole expression. The displacement procedure thus did not affect the core system of combining expressions. In this sense, it is a safe extension of the base grammar: It does not cut the bond between form and meaning. The question it raised, however, is which purpose displacement serves if it does not receive an interpretation. The answer I hinted at is that displacement can indeed have an interpretative effect, even if it is not directly input to a semantic interpretation, namely by creating new configurations from

which information-structural notions are derived. The claim of this thesis is thus not to deprive displacement of its general purpose, but only to deprive it of its role in establishing operator scope.

Let us now turn to the case of semantics parting company with syntax. I invoked a mechanism for control transfer that basically consisted of a rewriting rule operating on semantic expressions in order to establish operator scope. Since control transfers occurred in the meaning dimension only and since they were encoded in the semantic types but were not reflected in the syntactic category, they had no effect on syntactic well-formedness or behavior. They also did not affect the basic operations for combining expressions. Therefore also the semantic procedure leaves the base grammar untouched.

The syntax/semantics interface encoded in the base grammar in Chapter 2 is thus unaffected by the extensions in Chapter 4 and Chapter 5. That is, we can give up a strict correspondence between form and meaning without losing the important connection between them.

Although this thesis focused on quantifiers and wh-question formation, the proposal of loosening the tie between syntax and semantics has far-reaching consequences for the modeling of the syntax/semantics interface in general. Additional to the phenomena considered in this thesis, there are a lot more which seem to require a tight interaction between syntax and semantics, for example bound variable pronouns, crossover effects, VP ellipsis and antecedent-contained deletion, quantifier intervention effects as discussed by Beck [7], and so on. All of them would require a substantial amount of further research.

In order to sketch that a treatment of such phenomena in our approach to the syntax/semantics interface could not only be possible but also simple and attractive, I will briefly consider crossover effects. Crossover effects subsume the generalization that a wh-expression can bind a pronoun only if it c-commands it in its base position, or in other words: displacement of a wh-expression does not create new binding possibilities. This is illustrated in the following examples. In (7.76), the pronoun can be bound by the wh-expression, whereas this is not possible in (7.77).

(7.76) a. Who$_1$ __$_1$ searched for his$_1$ friend?

b. Who$_1$ __$_1$ thought that he$_1$ can defeat death?

(7.77) a. * Whom$_1$ did he$_1$ search for __$_1$?

b. * Whom$_1$ did [his$_1$ friend] search for __$_1$?

c. * Whom$_1$ did he$_1$ think death will defeat __$_1$?

The way we keep track of the derivational order at the edge of complex expressions and discard all other information actually allows for two straightforward ways to explain crossover effects – a syntactic and a semantic one, depending on whether one wants to consider conditions on pronominal binding to be of a syntactic or a semantic nature.

First assume that pronominal binding is semantic. Then a very natural way to account for it in the present proposal is to assume that a binding relation is established whenever the expression denoting the binder enters the derivation. This is because at that point all semantic information is contributed and processed. Later operations such as splitting and percolating an expression only keep phonological and syntactic information but is not subject to semantic interpretation. So if a semantic relation is to be established between two expressions, it has to happen when they are merged. Now, this actually already gives us the crossover effects above: When the binder who in (7.76a) enters the derivation, it is merged with the predicate searched for his friend. At this point, the pronoun is present and therefore can be bound. When who enters the derivation in (7.77b), on the other hand, it is merged with the verb search for. The pronoun is not yet part of the expression built and cannot be bound. Later, when the pronoun finally is present, the semantics of who has already been processed. Even if it is still at the edge, it is only a form and does not carry any semantic content, thus cannot provide a binder at this stage.

If we instead assume that pronominal binding is a syntactic notion, we could derive the same pattern. We have to rely on two straightforward assumptions. First, syntactic binding requires c-command. And second, c-command is defined in the derivational sense of Epstein (recall Section 2.2.1 from the beginning): an expression x c-commands another expression y if x is merged with y in the course of a derivation, or with an expression that contains y. This is, in fact, the only way to express c-command in our approach, since we do not preserve structural configurations. Recall that two expressions are merged only when they enter the derivation (**merge** is not part of the **remerge** operation), therefore c-command is defined only with respect to the base position of a displaced expression and disregards its target position. Now, in (7.76), who is merged with an expression that contains the pronoun, thus c-commands the pronoun according to the derivational definition of c-command. In (7.77), on the other hand, whom is merged with an expression that does not yet contain the pronoun, thus according to the definition it does not c-command the pronoun. The requirement for syntactic binding is thus satisfied in (7.76), but is not in (7.77).

I hope that this thesis has demonstrated that the parallel assembly of form and meaning does not require a tight link between syntactic and semantic procedures. I claim that a proper amount of independence not only offers promising analyses for phenomena on the border between syntax and semantics, but also facilitates a general and simple approach to the syntax/semantics interface.

Bibliography

[1] Hiyan Alshawi and Richard Crouch. Monotonic semantic interpretation. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL-92), Newark, NJ*, pages 32–39, 1992.

[2] Joseph Aoun and Yen-Hui Audrey Li. *Essays on the representational and derivational nature of grammar*. MIT Press, 2003.

[3] Carl L. Baker. Notes on the description of English questions: The role of an abstract question morpheme. *Foundations of Language*, 6:197–219, 1970.

[4] Henk Barendregt and Erik Barendsen. Introduction to the lambda calculus, revised edition. Technical report, University of Nijmegen, 2000. `ftp://ftp.cs.kun.nl/pub/CompMath.Found/lambda.pdf`.

[5] Chris Barker. Continuations and the nature of quantification. *Natural Language Semantics*, 10(3):211–242, 2002.

[6] Jon Barwise and Robin Cooper. Generalized quantifiers and natural language. *Linguistics and Philosophy*, 4(2):159–219, 1981.

[7] Sigrid Beck. Quantified structures as barriers for LF movement. *Natural Language Semantics*, 4:1–56, 1996.

[8] Sigrid Beck and Shin-Sook Kim. On wh- and operator scope in Korean. *Journal of East Asian Linguistics*, 6:339–384, 1997.

[9] Raffaella Bernardi and Michael Moortgat. Continuation semantics for the Lambek-Grishin calculus. *Information and Computation*, to appear.

[10] Rajesh Bhatt. Topics in the syntax of the modern Indo-Aryan languages: wh-in-situ and wh-movement. Handout, 2003. `http://web.mit.edu/rbhatt/www/24.956/wh.pdf`.

[11] Loren Billings and Catherine Rudin. Optimality and superiority: A new approach to overt multiple-wh ordering. In J. Toman, editor, *Proceedings of Annual Workshop on Formal Approaches to Slavic Linguistics. The College Park Meeting 1994*, pages 35–60. Michigan Slavic Publications, 1996.

[12] Johan Bos. Predicate logic unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, pages 133–142, 1995.

[13] Michael Brody. On the status of representations and derivations. In D. Epstein S. and D. Seely T. editors, *Derivation and Explanation in the Minimalist Program*, pages 19–41. Blackwell, 2002.

[14] Ulf Brosziewski. *Syntactic Derivations. A Nontransformational View.* Linguistische Arbeiten 470. Niemeyer, 2003.

[15] Miriam Butt. Object specificity and agreement in Hindi/Urdu. In *Papers from the 29th Regional Meeting of the Chicago Linguistics Society.* Chicago Linguistics Society, Chicago, 1993.

[16] Lisa Lai-Shen Cheng. *On the Typology of Questions.* PhD thesis, Massachusetts Institute of Technology, 1991.

[17] Noam Chomsky. The logical structure of linguistic theory. PhD thesis draft, MIT, 1955. Published in 1975 by Plenum Press, New York.

[18] Noam Chomsky. Conditions on transformations. In S. Anderson and P. Kiparsky, editors, *A Festschrift for Morris Halle*, pages 232–286. Academic Press, New York, 1973.

[19] Noam Chomsky. A minimalist program for linguistic theory. In K. Hale and J. Keyser S. editors, *The view from Building 20: Essays in linguistics in honor of Sylvain Bromberger*, pages 1–52. MIT Press, Cambridge, Mass., 1993.

[20] Noam Chomsky. *The Minimalist Program.* MIT Press, 1995.

[21] Noam Chomsky. Minimalist Inquiries: The framework. In D. Michaels R. Martin and J. Uriagereka, editors, *Step by Step. Essays on Minimalist Syntax in Honor of Howard Lasnik.* MIT Press, Cambridge, MA, 2000.

[22] Noam Chomsky. Derivation by phase. In M. Kenstowicz, editor, *Ken Hale: a Life in Language.* MIT Press, 2001.

[23] Noam Chomsky. On phases. Ms., MIT, 2005.

[24] Peter Cole and Gabriella Hermon. Is there LF wh-movement? *Linguistic Inquiry*, 25:239–262, 1994.

[25] Peter Cole and Gabriella Hermon. The typology of wh-movement: Wh-questions in Malay. *Syntax*, 1(33):221–258, 1998.

[26] Robin Cooper. *Montague's semantic theory and transformational syntax.* PhD thesis, University of Massachusetts at Amherst, 1975.

[27] Robin Cooper. *Quantification and Syntactic Theory*, volume 21 of *Synthese Language Library.* Reidel, 1983.

[28] Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. Minimal Recursion Semantics: An introduction. *Research on Language and Computation*, 3(4):281–332, 2005.

[29] Diana Cresti. Some considerations on wh-decomposition and unselective binding. In G. Katz, S.-S. Kim, and H. Winhart, editors, *Sprachtheoretische Grundlagen fr Computerlinguistik: Arbeitspapiere des Sonderforschungsbereichs 340.* Universitt Tübingen, 1998.

[30] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, pages 151–160, New York, 1990. ACM Press.

[31] Veneeta Dayal. *Locality in Wh-quantification: Questions and Relative Clauses in Hindi*, volume 62 of *Studies in Linguistics and Philosophy.* Kluwer, Dordrecht, 1996.

[32] Philippe de Groote. Type raising, continuations, and classical logic. In R. van Rooy and M. Stokhof, editors, *Thirteenth Amsterdam Colloquium*, pages 97–101. Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2001.

[33] Philippe de Groote. Towards a Montagovian account of dynamics. In *Proceedings of Semantics and Linguistic Theory 16.* CLC Publications, 2006.

[34] Elisabeth Engdahl. *Constituent Questions: With Special Reference to Swedish.* Reidel, 1986.

[35] Samuel David Epstein. Unprincipled syntax and the derivation of syntactic relations. Manuscript, Harvard, 1995. Published in: S.D. Epstein & N. Hornstein: Working Minimalism, pp 317–345. MIT Press, 1999.

[36] Samuel David Epstein, Erich M. Groat, Ruriko Kawashima, and Hisatsugu Kitahara. *A Derivational Approach to Syntactic Relations.* Oxford University Press, 1998.

[37] Samuel David Epstein and T. Daniel Seely. Rule applications as cycles in a level-free syntax. In S.D. Epstein and T.D. Seely, editors, *Derivation and Explanation in the Minimalist Program.* Blackwell Publishers, Oxford, 2002.

[38] Sam Featherstone. Magnitude estimation and what it can do for your syntax: Some wh-constraints in german. *Lingua*, 115(11):1525–1550, 2005.

[39] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, 1988.

[40] Janet Dean Fodor and Ivan A. Sag. Referential and quantificational indefinites. *Linguistics and Philosophy*, 5(3):355–398, 1982.

[41] Robert Frank. *Phrase structure composition and syntactic dependencies*. MIT Press, Cambridge, Mass., 2002.

[42] Gerald Gazdar. Unbounded dependencies and coordinate structure. *Linguistic Inquiry*, 12:155–184, 1981.

[43] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, and Basil Blackwell, Oxford, 1985.

[44] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[45] Jeroen Groenendijk and Martin Stokhof. *Studies on the Semantics of Questions and the Pragmatics of Answers*. PhD thesis, Universiteit van Amsterdam, 1984.

[46] C. L. Hamblin. Questions in Montague English. *Foundations of Language*, 10:41–53, 1973.

[47] Fabian Heck and Gereon Müller. Successive cyclicity, long-distance superiority, and local optimization. In Roger Billerey and Brook D. Lillehaugen, editors, *Proceedings of WCCFL 19*, pages 218–231. Somerville, MA: Cascadilla Press, 2000.

[48] Irene Heim. *The Semantics of Definite and Indefinite Noun Phrases*. PhD thesis, Umass Amherst, 1982.

[49] Irene Heim and Angelika Kratzer. *Semantics in Generative Grammar*. Blackwell Textbooks in Linguistics. Blackwell Publishers Ltd, Oxford, 1998.

[50] Herman Hendriks. Type change in semantics: the scope of quantification and coordination. In E. Klein and J. van Benthem, editors, *Categories, Polymorphism and Unification*, pages 96–119. ITLI, Amsterdam, 1988.

[51] Herman Hendriks. *Studied Flexibility*. PhD thesis, ILLC Dissertation Series, Amsterdam, 1993.

[52] James Higginbotham and Robert May. Questions, quantifiers, and crossing. *The Linguistic Review*, 1:41–80, 1981.

[53] Jerry Hobbs and Stuart Shieber. An algorithm for generating quantifier scoping. *Computational Linguistics*, 13:47–63, 1987.

[54] Norbert Hornstein. On A-chains: A reply to Brody. *Syntax*, 3(2):129–143, 2000.

[55] Cheng-Teh James Huang. *Logical relations in Chinese and the theory of grammar*. PhD thesis, MIT, Cambridge, Mass., 1982.

[56] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[57] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *Journal of Computer and System Science*, 10:136–163, 1975.

[58] Lauri Karttunen. Syntax and semantics of questions. *Linguistics and Philosophy*, 1:1–44, 1977. Also published in: Portner & Partee (eds.): *Formal Semantics. The Essential Readings*. Blackwell, 2003, pp 382–420.

[59] Richard S. Kayne. Connectedness. *Linguistic Inquiry*, 14:223–250, 1983.

[60] Richard S. Kayne. *The Antisymmetry of Syntax*. Linguistic Inquiry Monograph Twenty-Five. The MIT Press, Cambridge, 1994.

[61] William R. Keller. Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 432–447. Reidel, Dordrecht, 1988.

[62] Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, 2005.

[63] Oleg Kiselyov. Call-by-name linguistic side effects. ESSLLI 2008 Workshop on Symmetric calculi and Ludics for the semantic interpretation. 4-7 August, 2008. Hamburg, Germany. Available at `http://okmij.org/ftp/Computation/gengo/gengo-side-effects-cbn.pdf`, 2008.

[64] Robert Kluender. On the distinction between strong and weak islands: a processing perspective. *Syntax and Semantics*, 29:241–279, 1998.

[65] Gregory M. Kobele. *Generating Copies: An investigation into structural identity in language and grammar*. PhD thesis, UCLA, 2006.

[66] Masatoshi Koizumi. Layered specifiers. In *Proceedings of the North Eastern Linguistic Society*, volume 24, pages 255–269. University of Massachusetts, Amherst, 1994.

[67] Jan Koster. Variable-free grammar. Ms., University of Groningen, 2000.

[68] Anthony Kroch. Asymmetries on long distance extraction in a Tree Adjoining Grammar. In M. Baltin and A. Kroch, editors, *Alternative conceptions of phrase structure*, pages 66–98. University of Chicago Press, 1989.

[69] Anthony Kroch and Aravind K. Joshi. The linguistic relevance of Tree Adjoining Grammar. Technical report, University of Pennsylvania Department of Computer and Information Sciences Technical Report MS-CIS-85-16, 1985.

[70] Richard Larson. On the double object construction. *Linguistic Inquiry*, 19:335–391, 1988.

[71] David Lewis. Adverbs of quantification. In E. Keenan, editor, *Formal Semantics of Natural Language*, pages 3–15. Cambridge University Press, 1975.

[72] Anoop Mahajan. *The A/A-bar distinction and movement theory.* PhD thesis, MIT, Cambridge, Mass., 1990.

[73] Joan Maling. An asymmetry with respect to wh-islands. *Linguistic Inquiry*, 9:75–89, 1978.

[74] Robert May. *The Grammar of Quantification.* PhD thesis, MIT, 1977.

[75] Robert May. *Logical Form: Its Structure and Derivation.* MIT Press, Cambridge, Mass., 1985.

[76] Roland Meyer. Superiority effects in Russian, Polish and Czech: Judgments and grammar. *Cahiers linguistiques d'Ottawa*, 32:44–65, 2004.

[77] Gary L. Milsark. *Existential sentences in English.* Garland, New York & London, 1979. Published version of MIT PhD thesis, 1974.

[78] Richard Montague. The proper treatment of quantification in ordinary English. In H. Thomason R. editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 247–270. Yale University Press, 1974 (original version of the paper: 1970).

[79] Michael Moortgat. Categorial type logics. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, chapter 2, pages 93–177. Elsevier/MIT Press, 1997.

[80] Gereon Müller. Constraints on displacement: A phase-based approach. Ms., University of Leipzig.

[81] Gereon Müller. *A-bar syntax: A study in movement types*. Mouton, 1995.

[82] Gereon Müller. *Incomplete Category Fronting*. Kluwer, 1998.

[83] Ad Neeleman and Hans van de Koot. A local encoding of syntactic dependencies and its consequences for the theory of movement. Ms., University College London, 2007.

[84] William O'Grady. *Syntactic Carpentry: An Emergentist Approach to Syntax*. Erlbaum, Mahwah, NJ, 2005.

[85] Jong Cheol Park. Quantifier scope, lexical semantics, and surface structure constituency. Technical report, IRCS Report 96-28, 1996.

[86] David Pesetsky. Language-particular processes and the Earliness Principle. Ms., MIT, Cambridge, Mass., 1989.

[87] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

[88] Tanya Reinhart. *The Syntactic Domain of Anaphora*. PhD thesis, MIT, 1976.

[89] Tanya Reinhart. Quantifier scope: How labor is divided between QR and choice functions. *Linguistics and Philosophy*, 20:335–397, 1997.

[90] Tanya Reinhart. *Interface strategies: Optimal and costly computation*. Linguistic Inquiry Monographs. MIT Press, Cambridge, Mass., 2006.

[91] Chris H. Reintges, Philip LeSourd, and Sandra Chung. Movement, wh-agreement, and apparent wh-in-situ. In Lisa Lai-Shen Cheng and Norbert Corver, editors, *Wh-Movement: Moving On*, pages 165–194. The MIT Press, 2006.

[92] Uwe Reyle. Dealing with ambiguities by underspecification. *Journal of Semantics*, 10:123–179, 1993.

[93] Norvin Richards. Subjacency forever. In *Proceedings of WECOL 1996*. Department of Linguistics, California State University, Fresno, 1998.

[94] Norvin Richards. *Movement in Language: Interactions and Architectures*. Oxford University Press, 2001.

[95] Luigi Rizzi. *Relativized Minimality*. Number 16 in Linguistic Inquiry Monographs. MIT Press, Cambridge, 1990.

[96] John Robert Ross. *Constraints on variables in syntax*. PhD thesis, MIT, Cambridge, Mass., 1967.

[97] Catherine Rudin. On multiple questions and multiple wh-fronting. *Natural Language and Linguistic Theory*, 6:445–501, 1988.

[98] Eddy Ruys. *The Scope of Indefinites*. PhD thesis, OTS, Utrecht, 1992.

[99] Chung-chieh Shan. Delimited continuations in natural language: quantification and polar sensitivity. Continuation Workshop 2004, Venice.

[100] Chung-chieh Shan. Quantifier strengths predict scopal possibilities of Mandarin Chinese wh-indefinites. Draft manuscript, Harvard University; `http://www.eecs.harvard.edu/~ccshan/mandarin/`.

[101] Chung-chieh Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming, Snowbird, Utah*, 2004.

[102] Chung-chieh Shan. *Linguistic Side Effects*. PhD thesis, Harvard University, 2005.

[103] Chung-chieh Shan and Chris Barker. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy*, 29(1):91–134, 2006.

[104] Natalia Slioussar. *Grammar and information structure: a study with reference to Russian*. PhD thesis, Utrecht Institute of Linguistics OTS, LOT Dissertation Series 161, 2007.

[105] Edward Stabler. Computing quantifier scope. In Anna Szaboicsi, editor, *Ways of Scope-Taking*, pages 155–182. Kluwer, Dordrecht, 1997.

[106] Edward Stabler. Derivational minimalism. In C. Retoré, editor, *Logical Aspects of Computational Linguistics*, pages 68–95. Springer, 1997.

[107] Edward Stabler and Edward Keenan. Structural similarity. *Theoretical Computer Science*, 293:345–363, 2003.

[108] Michal Starke. *Move dissolves into Merge: A theory of locality*. PhD thesis, University of Geneva, 2001.

[109] Mark Steedman. *The syntactic process*. MIT Press, 2000.

[110] Arthur Stepanov. The end of CED? Minimalism and extraction domains. *Syntax*, 10(1):80–126(47), 2007.

[111] Thomas S. Stroik. *Locality in Minimalist Syntax*. Linguistic Inquiry Monograph 51. The MIT Press, 2009.

[112] Anna Szabolcsi. *Ways of Scope Taking*. Kluwer, Dordrecht, 1997.

[113] Anna Szabolcsi. The syntax of scope. In M. Baltin and C. Collins, editors, *Handbook of Contemporary Syntactic Theory*, pages 607–634. Blackwell, 2000.

[114] Anna Szabolcsi and Marcel den Dikken. Islands. *GLOT International 4/6*, 1999. reprinted in Lisa Cheng and Rint Sybesma, eds., The Second GLOT State-of-the-Article Book, Mouton de Gruyter (2002).

[115] Sonoko Takahashi. *The interrogative marker KA in Japanese*. PhD thesis, The Ohio State University, 1995.

[116] Hidekazu Tanaka. LF wh-islands and the Minimal Scope Principle. *Natural Language and Linguistic Theory*, 17:371–402, 1999.

[117] Wei-Tien Dylan Tsai. *On economizing the theory of A-bar dependencies*. PhD thesis, MIT, 1994.

[118] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, to appear.

[119] Willemijn Vermaat. *The logic of variation. A cross-linguistic account of wh-question formation*. PhD thesis, UiL OTS, Utrecht, 2005.

[120] Željko Bošković. On certain violations of the Superiority Condition, AgrOP, and economy of derivation. *Journal of Linguistics*, 33:227–254, 1997.

[121] Željko Bošković. On multiple feature checking: Multiple wh-fronting and multiple head movement. In S. Epstein & N. Hornstein, editor, *Working Minimalism*, Current Studies in Linguistics 32, pages 159–187. MIT Press, 1999.

[122] Željko Bošković. Sometimes in SpecCP, sometimes in-situ. In R. Martin, D. Michaels, and J. Uriagereka, editors, *Step by step: Essays on minimalism in honor of Howard Lasnik*, pages 53–87. MIT Press, Cambridge, Mass., 2000.

[123] Željko Bošković. On multiple wh-fronting. *Linguistic Inquiry*, 33(3):351–383, 2002.

[124] Željko Bošković. On wh-islands and obligatory wh-movement contexts in South Slavic. In C. Boeckx and K. Grohmann, editors, *In Multiple wh-fronting*, pages 27–50. John Benjamins, 2003.

[125] Željko Bošković. On the selective wh-island effect, 2008.

[126] Akira Watanabe. Subjacency and s-structure movement of wh-in-situ. *Journal of East Asian Linguistics*, 1:255–291, 1992.

[127] Akira Watanabe. Wh-in-situ languages. In M. Baltin and C. Collins, editors, *The Handbook of Contemporary Syntactic Theory*, pages 203–225. Blackwell, Oxford, 2001.

Index

Samenvatting in het Nederlands

Deze dissertatie gaat over de syntaxis en semantiek van niet-lokale afhankelijkheden. Het concentreert zich op wh-verplaatsing en het bereik van operatoren, en kijkt naar de uitdagingen die deze fenomenen vormen voor de theoretische taalkunde: hoe zijn vorm en betekenis gerelateerd en in hoeverre opereren syntaxis en semantiek parallel aan elkaar?

In formeel grammaticale theorieën is het gangbaar er vanuit te gaan dat syntactische verplaatsing en bereiksconstructie hand in hand gaan, en daarmee wordt aangenomen dat een strikte correspondentie bestaat tussen syntaxis en semantiek. Deze aanpak wordt echter ter discussie gesteld door een aanzienlijk aantal empirische gevallen waarin de syntactische positie van een operatorexpressie niet samenvalt met de semantische bereikspositie. In deze dissertatie is het uitgangspunt omgekeerd: er wordt beargumenteerd dat syntactische verplaatsing en semantische bereikseffecten elkaar helemaal niet beïnvloeden.

De aanpak die in deze dissertatie wordt ontwikkeld heeft als centrale aanname dat grammatica uit twee delen bestaat. Allereerst is er een kern waarin de lokale afhankelijkheden worden bepaald, en waarin syntaxis en semantiek parallel aan elkaar opereren. Vervolgens zijn er uitbreidingen van deze kern waarin de niet-lokale afhankelijkheden worden bepaald, en waarin syntaxis en semantiek onafhankelijk van elkaar opereren. Het is juist in het uitbreidingsdeel waar syntactische verplaatsing en bereiksconstructie worden behandeld. Syntactische verplaatsing wordt gezien als een syntactische procedure die opereert op vormniveau, terwijl bereiksconstructie wordt beschouwd als een semantische procedure die alleen op betekenisniveau opereert.

In de uiteenzetting van deze procedures wordt duidelijk dat de belangrijke data – zoals de formulering van wh-clusters, behoud van volgorde met multipele syntactische verplaatsing, condities op lokaliteit, overblijfselverplaatsing (*remnant movement*) en *Freezing*, zowel als afwijkende gevallen van semantisch

bereik – op een bevredigende manier geanalyseerd kunnen worden met behulp van een syntaxis en semantiek die onafhankelijk van elkaar opereren.

Curriculum vitae

Christina Unger was born on the 5th of May 1982 in Leipzig, German Democratic Republic. She got her *Abitur* graduating from the Neue Nikolaischule in Leipzig in 2000. In the same year, she started her studies in Logic & Philosophy of Science, Mathematics and Linguistics at the University of Leipzig, from which she received a Magister Artium degree ("passed with distinction") in 2006, majoring in Linguistics and Logic & Philosophy of Science. In 2006, she was enrolled in the International PhD program at the Utrecht Institute of Linguistics OTS in the Netherlands. This thesis is the result of the work she carried out there. In October 2009, she started work as a research assistant in the Semantic Computing Group of the Cognitive Interaction Technology Center of Excellence (CITEC) at the University of Bielefeld, Germany.