Natural Language Generation by Abstract Machine for Typed Feature Structures

Evgeniy Gabrilovich

Natural Language Generation by Abstract Machine for Typed Feature Structures

Project Thesis

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science

Evgeniy Gabrilovich

Submitted to the Senate of the Technion — Israel Institute of Technology

Tamuz 5758 Haifa June 1998

The work described herein was supervised by Prof. Nissim Francez under the auspices of the Faculty of Computer Science

'But "glory" doesn't mean "a nice knock-down argument," ' Alice objected.
'When **I** *use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean — neither more nor less.'*
'The question is,' said Alice, 'whether you can *make words mean so many different things.'*
'The question is,' said Humpty Dumpty, 'which is to be master — that's all.'
Alice was too much puzzled to say anything, so after a minute Humpty Dumpty began again. 'They've a temper, some of them — particularly verbs, they're the proudest — adjectives you can do anything with, but not verbs — however, **I** *can manage the whole of them!'*

– Lewis Carroll, "Through the Looking Glass"

Acknowledgments

I am beholden to Prof. Nissim Francez for his counsel and guidance throughout this project. I am deeply grateful to Prof. Francez for teaching me a formal and mathematically precise approach to tackling new problems. I would also like to thank Prof. Francez for the encouragement he accorded me when I needed it.

I wish to express my gratitude to Prof. Alon Itai for replacing Prof. Francez while he was on sabbatical leave. I am thankful to Prof. Itai for his time, valuable help, and advice.

I am indebted to Dr. Shuly Wintner for his comments and the constructive review of my work. I am especially thankful to Dr. Wintner for the inestimable assistance he kindly rendered me when I was working on the software part of this project.

I wish to thank Dr. Christer Samuelsson for the clarification of particularly complex points in his work.

Last but not the least, I owe a debt of gratitude to my parents, Margarita and Solomon, for their tolerance, love, and concern.

The generous financial help of the Technion — Israel Institute of Technology is gratefully acknowledged.

Contents

List of Figures

List of Tables

Abstract

Computational Linguistics is the branch both of Computer Science and Linguistics that deals with the computational aspects of natural language phenomena, where one of its goals is to give computers the ability to properly understand and coherently produce texts. The research in this area aims at describing mathematically precise and (preferably) computationally effective models for representing language communication. State of the art computer systems are capable of handling texts at a substantial level of language coverage. The two main directions in contemporary Computational Linguistics are Natural Language Analysis (leading to Natural Language Understanding) and Generation. The latter (which is the principal object of this work) is enabling computers to produce natural language texts conveying the semantic meaning given as input. The generation ability is important in many computer applications: from verbalizing results of a query, through generating large-scale documentation to (when combined with an analysis module) communicating with humans.

This work applies the *Abstract Machine* approach to the problem of *Natural Language Generation*. Such a machine is an abstraction over an ordinary computer, lying somewhere between regular high-level language and common hardware architecture. Programming an Abstract Machine has proved fruitful in previous research, reaching its recent peak as a highly efficient technique to build Prolog compilers. This approach not only yields efficient computation, it also allows formal verification of the programs written for the Abstract Machine.

To perform generation efficiently, we first ported an existing algorithm for surface generation, proposed for a logic programming framework (Samuelsson, 1995), into a unification-based Typed Feature Structure (TFS) formalism. The algorithm uses two internal grammar transformations of normalization and inversion, to render the given grammar with a form inherently suitable for generation. We then defined a concept of *chart generation*, which is similar to the familiar *chart parsing*. In the case of generation it is the given semantic meaning whose components are consumed in the process. Finally, a generation module was designed for the WAM-like Abstract Machine for Natural Language analysis, previously developed in (Wintner, 1997) having only parsing capabilities. This resulted in an efficient, bidirectional (parsing/generation) system for Natural Language Processing.

List of Abbreviations

AF	Argument-Filling (rule)
ALE	Attribute Logic Engine
AM	Abstract Machine
$\mathcal{A}$MALIA	Abstract MAchine for LInguistic Applications
AVM	Attribute Value Matrix
CF	Context Free
CR	Connective Registry
DCG	Definite Clause Grammar
DP	Designated Path
FI	Functor-Introducing (rule)
GUI	Graphical User Interface
HPSG	Head-Driven Phrase Structure Grammar
LD	Lexicon-Derived (rule)
LHS	Left Hand Side
MRS	Multi-Rooted Structure
NL	Natural Language
NLA	Natural Language Analysis
NLG	Natural Language Generation
NLP	Natural Language Processing
RHS	Right Hand Side
SHDG	Semantic-Head-Driven Generation
SKB	Semantics Knowledge Base
TFS	Typed Feature Structure
WAM	Warren Abstract Machine

Chapter 1

Introduction

1.1 Motivation

Contemporary computational linguistics can hardly be imagined without extensive use of computers, which assist in implementing various models and formalisms for language representation. Considerable progress has been achieved in recent years when a number of relatively exhaustive linguistic theories have been introduced. Many of these formalisms describe significantly large fragments of natural language, striving, whenever possible, to develop language-independent approaches.

The field of Natural Language Processing encompasses two main tasks, namely Natural Language Analysis (NLA) and Natural Language Generation (NLG). The former deals with parsing texts to build their logical form representation according to the grammar supplied, while the latter addresses issues of creating texts by computers. The generation task is a relatively new research area, though it has a constantly growing number of devotees and has a recently established international yearly conference. An input for the generation task is a logical form which represents the desired meaning, and a grammar to govern the generation process. The output consists of one or more phrases in the language of the grammar whose meaning is the given logical form. This duality is illustrated by the sketch in Figure 1.1.

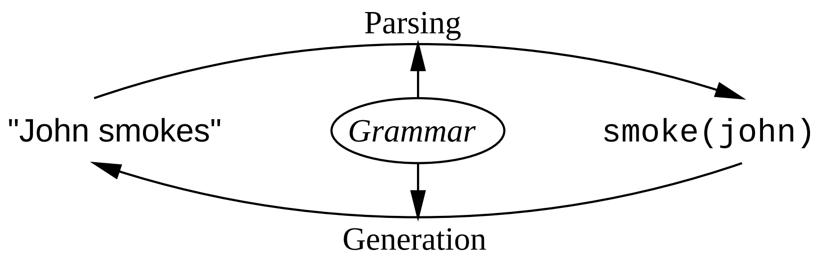


Figure 1.1: The duality of Natural Language Parsing and Generation tasks.

Following recent advances in Natural Language Processing, efforts have been made to formulate a uniform mechanism for effective treatment of both its constituents. In this context, bidirectional grammars, where a single set of rules allows both parsing and generation, are considered an advantage in the literature (Shieber, 1988; Strzalkowski, 1994). Such grammars have been traditionally formulated in the Definite Clause Grammar (DCG) formalism (Pereira and Shieber, 1987), while recent trends in computational linguistics tend to adopt unification-based Typed Feature Structure (TFS) formalisms (e.g. (Carpenter, 1992b)), which, in addition to other benefits, profit from the numerous advantages of the declarative nature of feature-structure unification.

Grammars are usually given in a form oriented towards the analysis of a string and not towards generation from a (usually nested) semantic form. In other words, rules reflect the phrase structure and not the predicate-argument structure. It is therefore desirable to transform the grammar to enable systematic reflection of any given logical form in the productions. To this end we apply an inversion procedure, based upon (Samuelsson, 1995), to the input grammar, to render the rules with the nested predicate-argument structure, corresponding to that of input logical forms. The resultant "inverted" grammar is thus most suitable for performing the generation task. Once the grammar is inverted, the generation process can be directed by the input semantic form; elements of the input are consumed during generation just like words are consumed during parsing.

The work presented here applies an Abstract Machine (AM) approach for Natural Language Generation. Wintner (1997) developed a WAM-like abstract machine (named $\mathcal{A}$MALIA — Abstract MAchine for LInguistic Applications) for Natural Language Parsing. That work proposed using an Abstract Machine within the framework of Typed Feature Structures, thus enabling parsing with unification-based grammar. As defined in (Wintner, 1997), $\mathcal{A}$MALIA performs *chart parsing* — a general parsing method (also known as *tabular parsing*), which stores intermediate parsing results in an auxiliary table ((Aho and Ullman, 1972, pp. 314-330) contains a discussion of several tabular parsing methods). In this work we devise a similar concept of *chart-based generation algorithm*, and use it to extend the machine to also cover the generation direction, by using the same grammar but with different compilers for parsing and for generation. To use $\mathcal{A}$MALIA as an infrastructure for building the generation module, we had to port the original inversion algorithm (Samuelsson, 1995) developed for definite clause grammars, to a feature-structure framework.

Thereby, the extended machine makes dual use of its chart and forms a complete bidirectional NLP system. The extended machine is capable of very efficient processing, since grammars are precompiled directly into abstract machine instructions, which are subsequently executed over and over.

Figure 1.2 delineates an overview of AM-based generation.

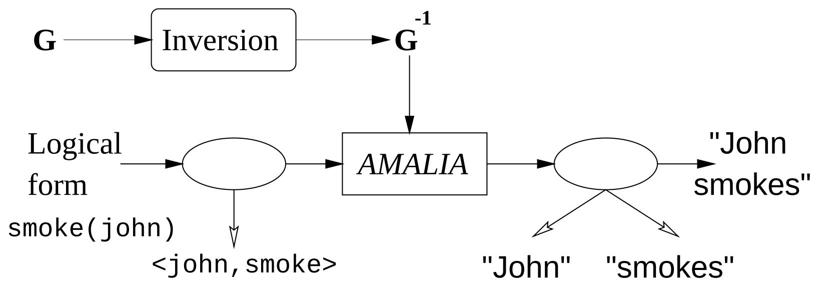


Figure 1.2: An overview of generation with Abstract Machine.

1.2 Literature survey

Historically, early mathematical studies of natural languages introduced context-free grammars as one of the major frameworks for language description. Their reasonable expressiveness, as well as availability of techniques for computationally effective processing, made these grammars a leading tool for defining programming languages and building efficient compilers (see, for example, (Aho, Sethi, and Ullman, 1986)). However, attempts to apply this formalism for describing NL grammars encountered a number of obstacles. On the one hand, there are natural language constructs that cannot possibly be represented at the context-free level[1]. On the other hand, even if a grammar can be described in context-free terms, agreement constraints between phrase constituents usually lead to a combinatorial explosion of the number of rules, as

[1]For example, *cross-serial dependencies* found in Dutch are of the form $a^n b^m c^n d^m$, which is proved to be non-context-free (Hopcroft and Ullman, 1979).

well as to a loss of generalization. In other words, context-free grammars lack the strong generative power[2] featured by the *unification* grammars, as they do not allow the necessary generalizations to be made about the structure of natural languages.

Unification grammars present a generalization over context-free ones, by arranging features of language entities in nested, possibly reentrant descriptions (referred to as *feature structures*, (Shieber, 1986)). The power of the enhanced formalism is equivalent to that of Turing Machines (Johnson, 1988). The basic operation of this approach is *unification*, which combines (non-contradicting) information of its operands (referred to as *unificands*). Carpenter (1992b) further extends the notion of feature structures with types, which are also subject to unification. *Typed Feature Structures (TFSs)* present a generalization over first-order terms and ordinary (typeless) feature structures. Among the advantages of this approach are the strong typedness of the whole definition and the multiple inheritance hierarchy of all the existing types. In light of this fact, many techniques of object-oriented programming such as feature inheritance among the types can be successfully applied. Moreover, handy artificial intelligence techniques for contradiction detection and resolution may be used when traversing the inheritance tree.

Studying recently proposed linguistic theories, we can infer that unification in general has proved to be a powerful tool (Shieber, 1986). This is partly because the unification framework has been thoroughly researched by multiple works in logic programming, so the processing of natural language can benefit from many achievements in that field. Consequently, unification-based formalisms became the leading technique for characterizing natural language grammars in modern computational linguistics. Lexical Functional Grammar (LFG) (Kaplan and Bresnan, 1982) and Head-Driven Phrase Structure Grammar (HPSG) (Pollard and Sag, 1994) are two of the more advanced members of this class. The latter has recently become the most popular linguistic formalism, and currently enjoys a very broad usage, as it covers a wide variety of natural language phenomena in a particularly clean and sound way. The basic functional units of HPSG are Typed Feature Structures introduced by Carpenter (1992b). The formalism defines a set of universal principles, which allegedly hold for all natural languages, and which are extendable with additional language-specific rules. HPSG grammars have been developed for a wide variety of languages from different families; a grammar for a small fragment of Hebrew has recently been developed by Wintner (1997). A number of systems are available that implement unification-based formalisms, and serve environments for grammar design and development. Among those are PATR (Shieber, 1986) and Attribute Logic Engine (ALE) (Carpenter, 1992a). ALE compiles input grammars into Prolog code, which is subsequently invoked under chart parsing strategy. Today, ALE is a widely accepted platform for developing HPSG grammars.

Chart parsing originated in works by Younger (1967) and Earley (1970) as CYK (Cocke-Younger-Kasami) and Earley's algorithms, which make use of an auxiliary table (*chart*) for recording partial processing results. Refer also to (Pereira and Shieber, 1987, Section 6.6) for a comprehensive discussion of tabular parsing in NLP. In this work we present a *chart generation* algorithm, which is conceptually similar to chart parsing in its use of an ancillary table, though instead of the words of the input string (as in the case of parsing), the generation procedure consumes parts of the given logical form.

Reiter (1994) gives an overview of a typical NL generation architecture. According to this work, a complete NLG system combines the following three main[3] modules in a pipeline form:

1. *Content determination* (or *text planning*) decides what information should be conveyed in the text to be created. This module usually accomplishes two tasks: *deep content determination*, which determines what information should be communicated to the hearer, and *rhetorical planning*, which organizes this information in a rhetorically coherent manner. The information chosen at this stage may be expressed in a number of sentences, hence this module is occasionally referred to as *strategic generation*, in contrast to the other two modules which perform *tactical generation* at the single-sentence level.

2. *Sentence planning* maps conceptual structures onto linguistic ones, and arranges information into clauses and sentences. This module also performs *lexical selection*, i.e., choosing among several synonymic words, expressions or idioms to represent each part of the input meaning.

[2]*Strong generative power* is the ability of the grammar to assign "correct" structures to natural language phrases.

[3]Reiter (1994) also mentions two additional modules, namely *morphology* and *formatting*, but those are secondary and far less common in actual NLG systems.

3. *Surface generation* produces surface forms that communicate the desired information in the form of grammatically correct sentences.

(Hovy et al., 1995) poses the problem of *logical form equivalence* as an important theoretical and practical problem for natural language generation. This is the case when several different logical forms correspond to the same surface string.

In this work we mean by "generation" what is sometimes known also as "syntactic generation": given a logical form (and a category)[4], use the grammar to construct an expression of the given category in the language of the grammar, the meaning of which is (up to logical equivalence) the given logical form. Thus, no text planning, speaker intentions and the like are considered here.

A broad variety of generation algorithms can be found in the literature. An early attempt to combine parsing and generation in one system using chart techniques may be found in (Shieber, 1988). This work describes a uniform architecture for both tasks, built around an Earley-style left-to-right chart scheme. Since the nested nature of logical forms (that serve input for generation) is not inherently suitable for chart processing, the system yields generation capability at the expense of efficiency. Semantic-head-driven generation (SHDG) introduced by Shieber et al. (1990) is more efficient, as it is naturally controlled by (parts of) the input semantics. Two examples of works that combine chart-based Earley deduction with semantic-head-driven approaches (mixed top-down and bottom-up strategies) while using the same grammar for both parsing and generation are (Gerdemann, 1991) and (Neumann, 1994). In the latter work parsing and generation operate on the very same *chart* and can *share items* using a *uniform indexing mechanism* for the retrieval of already completed subgoals (called *lemmas* of the Earley deduction). In the case of parsing lemmas are indexed using string information, and in the case of generation they are accessed using semantic information. *Item sharing* enables integration of parsing and generation into an interleaved processing scheme, which may be beneficially used to handle monitoring, revision or generation of paraphrases.

Samuelsson (1995) augmented the SHDG algorithm with a preprocessing procedure, which filters out impossible generation scenarios and reduces spurious search. In his work, Samuelsson (1995) presents a method for normalizing and then inverting the given grammar, so that it can be used by an LR[5] (recursive-descent) generation algorithm. Popowich (1996) describes a chart generator for Shake-and-Bake machine translation, where generation is performed from semantic constituents (called *signs*) stored in a *bag*[6]. Since the bag is not ordered, arbitrary numbering of signs is introduced, and the generator ensures that each sign is used exactly once in the sentence created. The process of generation is governed by the semantic indices of signs, which constrain the way signs may join one another. Kay (1996) describes another chart generation algorithm whose input is flat representation of logical forms, like that used for transfer in Shake-and-Bake translation (Whitelock, 1992). Since constituents of flat semantics are unordered, generating from such input can be seen as parsing a language with absolutely free word order. To prevent combinatorial explosion of time complexity (due to the exponential number of interactions between chart edges), Kay (1996) introduces a notion of *indexing*. The approach distinguishes between *internal* indices, which are unaccessible outside a category, and *external* indices otherwise. The generation algorithm then ensures that each complete edge subsumes all the predicates indexed by the indices internal to this edge. This allows to avoid a lot of spurious search, which results from generating incomplete expressions that leave out parts of the input which cannot be added later. Trujillo (1997) proposes an algorithm to compute the necessary indices automatically, instead of requiring this information to be specified explicitly by the grammar designer. As opposed to the chart generation algorithms outlined above, in our work semantic constituents are ordered (the ordering being induced by recursive processing of the given logical form) and can only join in one way.

As mentioned above, this work uses an adaptation of the surface generation algorithm due to Samuelsson (1995) for a feature-structure framework. To mention other unification-based approaches to generation, Elhadad (1989) presents a feature-based generation framework - Functional Unification Formalism (FUF), based on the functional unification grammar. Calder, Reape, and Zeevat (1989) present an algorithm for generation in unification categorial grammar.

[4]In an extended sense, namely, a feature structure.

[5](Aho, Sethi, and Ullman, 1986, Section 4.7) contains a comprehensive review of LR parsing.

[6]A *bag* is a *multi-set*, i.e., an unordered sequence of items with possible repetitions.

Normally, the generation process starts from a logical form encoding some semantic meaning. It is therefore important to develop a notation for representing semantic objects, that posesses adequate linguistic felicity and expressive power. To this end, we model our semantic interpretation along the guidelines of Montague semantics (Gamut, 1991, Chapter 5)[7], which is based on λ-calculus. β-reduction applied during rule invocation is built into the grammar rules per se, and is thus implicit. The fact that Montague's grammars are purely compositional constitutes an advantage, as it facilitates decomposition of complex logical forms into basic semantic primitives. To encode Montague semantics with feature structures we employ a notation very similar to that of (Nerbonne, 1992).

As mentioned above, our NL generator extends $\mathcal{A}$MALIA — an abstract machine for parsing designed by Wintner (1997). Such machines constitute a trade off between high-level programming languages on one hand, and low-level computer architecture on the other hand. Defined as such, AM benefits from both directions: while it can be conveniently programmed in a familiar high-level style, its code can be easily converted into a real computer assembly. Programming an Abstract Machine has proved fruitful in previous research, especially for implementing programming languages. Starting from a P-Code machine for Pascal, AM techniques were employed for a number of languages belonging to various paradigms. Warren (1983) devised an abstract machine for Prolog, which over the years turned a *de facto* standard for building compilers for the language. The Abstract Machine approach not only yields efficient computation, it also permits formal verification of the programs written for the AM.

1.3 The achievements of the thesis

The work reported herein is aimed at applying the *Abstract Machine* approach to the problem of *Natural Language Generation*. The goal of this project was to enrich and enhance the original $\mathcal{A}$MALIA to work also in the reverse direction. That is, given a TFS-encoded semantic representation to be able to generate a natural language phrase expressing it.

The contribution of the this thesis is threefold:

- First, it ported an existing algorithm for grammar inversion (due to Samuelsson (1995)) to a unification-based Typed Feature Structure framework similar to ALE. Such frameworks are considered by the computational linguistics community more suitable for grammar design than the DCG formalism, in which the algorithm has been formulated originally (Shieber, 1986).

  To prevent combinatorial explosion of generation rules, Samuelsson proposed to introduce a typing of logical forms. To this end we augmented the original algorithm by making use of the "object-oriented" nature of our TFS-based formalism. The type hierarchy is employed to decrease the number of different grammar rules at various processing stages. Thus, for example, a single grammar rule is sufficient for treating transitive verbs, so there is no need to specify a separate rule for each transitive verb in the lexicon.

- Second, the thesis defined a notion of chart generation, operating on the grammars inverted according to the above algorithm. This scheme was defined with $\mathcal{A}$MALIA's parsing functionality in mind, so that the two processing directions could be linked together into a single structure.

- Finally, $\mathcal{A}$MALIA was actually extended so that it became capable of performing generation of natural language constructs. That is, a compiler was written, that given a semantic formula $\mathcal{F}$ translates the input grammar to the abstract machine program $\mathcal{P}$, an invocation of which produces a sentence whose meaning is $\mathcal{F}$. The existing core activity of unification now works for both directions. It turned out that no new commands had to be added to $\mathcal{A}$MALIA's definition, and the augmentation was only in the control.

The combination of analysis and generation components using the *same* unification-based grammar forms a complete and computationally effective system for Natural Language Processing. The Abstract Machine

[7]Although *intensionality* is beyond the scope of the sample grammars considered.

approach also renders the entire system platform-independent, so that it can be easily implemented on any specific computer architecture.

Extending the existing code, we implemented the system in the ANSI-C programming language. The unified AMALIA uses the (YACC-based) input acquisition module of the original one, hence the same subset of ALE grammar specification language is provided. The application was tested on SUN and SILICON GRAPHICS workstations running UNIX operating system, as well as on an IBM PC running WINDOWS'95 and LINUX. Exactly as native AMALIA, the enhanced product supports graphical user interface and batch modes, to facilitate both grammar development and aggregate (analysis/generation) processing.

(Wintner, Gabrilovich, and Francez, 1997a) describes AMALIA as a unified platform for parsing and generation, elaborating more on the way the two directions are integrated into a single system. (Wintner, Gabrilovich, and Francez, 1997b) contains a user manual for the software.

1.4 Document outline

The theory of Typed Feature Structures is reviewed in Chapter 2. This chapter also gives a short survey of AMALIA along the guidelines of (Wintner, 1997), as well as hints on how it can be enhanced with generation capabilities.

Chapter 3 gives an account of grammar inversion, and explains how the algorithm proposed in (Samuelsson, 1995) was ported to a unification formalism. To assist the reader, a simple running example accompanies all the grammar transformations initiated by the algorithm. Chart-based generation with Typed Feature Structures is described in Chapter 4. First, the generation algorithm is presented, and then changes in AMALIA control are described, which were necessary to incorporate the generation module.

The software implementation of the generation extensions to AMALIA is given in Chapter 5. This part of the document details the data and control flow throughout the system, as well as depicts main functions and data structures used. Finally, Chapter 6 outlines conclusions and proposes possible further research directions. Appendix A contains the running example grammar and the results of sample generation with this grammar (including the intermediate inversion results). An additional sample grammar, which incorporates Montague semantics and covers more linguistic phenomena, is shown in Appendix B.

Chapter 2

Processing Unification-Based Grammars with an Abstract Machine

This chapter describes how the existing Abstract Machine for parsing unification-based grammars has been extended with generation capabilities. To this end, we first review the unification formalism in which grammars are encoded (Section 2.1), and how such grammars can be parsed using the $\mathcal{A}$MALIA abstract machine due to Wintner (1997) (Section 2.2). Finally, Section 2.3 explains how the control strategy of $\mathcal{A}$MALIA was augmented to perform generation as well.

2.1 A Typed Feature Structure based formalism for unification grammars

In this work we encode grammars with Typed Feature Structures (TFS). The notion of TFSs was formulated in (Carpenter, 1992b), and is briefly reviewed here. TFSs differ from ordinary feature structures (Shieber, 1986) in that each structure is associated with a *type*. A typed feature structure is a directed, connected, rooted, finite graph, whose vertices are labeled with *types* and edges are labeled with *features*. A bounded complete partial order (referred to as *type hierarchy*) defines how feature structures can be constructed of types and features. According to the type hierarchy, each type has a (possibly empty) list of *appropriate* features, and each feature is assigned the most general type which is *appropriate* as its value. Types may have *subtypes* which necessarily *inherit* their features; *multiple inheritance* is also possible when a single type simultaneously inherits from several other types. For this reason the type hierarchy is occasionally called *inheritance hierarchy.* Two criteria are available to verify whether a feature structure is well-formed according to the given type hierarchy. The first criterion defines *well-typed feature structures*, so that whenever an edge f connects two vertices v_1 and v_2, f is appropriate for $type(v_1)$, and the type appropriate for f in $type(v_1)$ subsumes $type(v_2)$. The second one defines *totally well-typed feature structures* as well-typed FSs, in which whenever a feature f is appropriate for the type of a vertex v, there is an edge starting at v and labeled with f.

Paths in typed feature structures are (possibly empty) finite sequences of feature names, which correspond to sequences of edges in the underlying graph. A path is *full* if it starts from the *root* of the graph, or *partial* if it starts from any other vertex. In general, TFSs may contain *cycles* (i.e., the underlying graph contains directed cyclic paths) and *reentrancies.* The common concepts of *subsumption* and *unification* are naturally extended for TFSs.

To use Typed Feature Structures for encoding Natural Language Grammars, one needs a way to represent grammar rules within the formalism. Wintner (1997) defines *multi-rooted feature structures* as a generalization of TFSs that provides a convenient notation for the rules. A multi-rooted structure (MRS) is a finite, directed, labeled graph with an ordered list of designated vertices called *roots.* The graph is not necessarily connected, but each vertex has to be accessible from at least one of the roots. Apparently, grammar rules can be naturally represented with such structures. Each rule constituent (either the head or a body element) is represented with a feature structure, which are combined into a single multi-rooted structure. The roots of a MRS are those of the constituent FSs, and their ordering is induced by the original rule. The *length* of an MRS is the number of its roots. Feature structures inside an MRS may contain reentrancies, which represent values shared among the constituents of the rule.

Grammars are represented in this formalism as triples $G = \{TH, R, L\}$, where TH is a type hierarchy, R is a set of rules and L is a lexicon. A rule is a multi-rooted structure of positive length; the distinguished first element of the MRS serves the rule *head*, and the rest of the elements comprise the *body* of the rule. ε-rules are also allowed under this definition, and are represented as MRSs of length 1 (rules having only a head and no body). Lexicon items may also be viewed as MRSs of length 1, where the only MRS element corresponds to a feature structure which the lexicon assigns to the given word. In the case of ambiguous words, several lexicon items might be required to describe a single word.

Syntax and semantics are encoded[1] in unification grammars through features of the MRSs which represent the rules. For instance, each rule constituent in the running example grammar has the features *syn* and *sem* defined, which encode the syntactic and semantic properties of this constituent, respectively. The *cat* feature under *syn* encodes categories of the context-free backbone of the grammar. For the sake of briefness we occasionally abbreviate grammar rules and only show their context-free backbone, instead of the entire MRS (e.g., $NP\ VP \Rightarrow S$). It should be observed that in such cases each CF variable (e.g., VP) encodes an entire feature structure rather than the syntactic category per se.

2.2 Parsing by Abstract Machine

In his work, Wintner (1997) devised an abstract machine for parsing unification grammars. The machine, called $\mathcal{A}$MALIA (the acronym stands for "Abstract MAchine for LInguistic Applications"), is specifically designed for executing grammars, encoded in (a subset of) ALE's language (Carpenter, 1992a). $\mathcal{A}$MALIA incorporates a compiler of input grammars into abstract machine instructions, and an interpreter for these instructions. Compilation constitutes a preprocessing phase – once compiled, grammars can be executed (interpreted) very efficiently. The interpreter implements a bottom-up chart parsing; special-purpose AM instructions are available to realize this control strategy. These instructions are responsible for effecting the regular chart operations, as well as for edge management (going over *active* and *complete* edges). Edges are induced by grammar rules and span a subsequence of the input string, assigning it some structure. Each edge contains the MRS representing a grammar rule plus some auxiliary information. In an *active* edge, a *dot* separates the already processed constituents from all the rest. The operation which takes an active edge and a complete edge, and moves the dot one position further to the right in the former (over the constituent which unifies with the complete edge) is called *dot movement*. When the dot reaches the position after the last body constituent, the *completion* operation is invoked to construct the head of the rule and create a *complete* edge. Both kinds of edges are recorded in the chart. The machine also uses a heap-like data structure (referred to as the *heap*), which constitutes the main memory of the machine and hosts all the feature structures built during the parsing process.

Given the input string of words, the parser performs lexical lookup, and each word is associated with a FS (or a set of FSs if the word is ambiguous). In the *scanning* phase, a complete edge is created for each such feature structure, and is then inserted into the appropriate cell on the main diagonal of the chart. The *prediction* phase creates active edges which anticipate in each cell of the main diagonal every possible rule. From here on, the standard chart parsing algorithm is invoked (cf. (Pereira and Shieber, 1987, Section 6.6)) which alternatively applies *dot movement* and *completion* operations. If parsing terminates, it ends up with a (possibly empty) set of feature structures, spanning the entire input.

[1]See Section 3.2 for a discussion of syntax and semantics representation in unification grammars.

2.3 Generation by Abstract Machine

In this work we define the concept of *chart generation* and extend $\mathcal{A}$MALIA (as presented in (Wintner, 1997)) with generation capabilities. The major observation which enables such dual usage of the machine is that the *dot movement* operation can be interpreted differently, depending on the nature of the chart items.

First, to transform grammars (initially designed for parsing) into a form more suitable for generation, we apply an *inversion* procedure based on (Samuelsson, 1995), which renders the rules with the nested predicate-argument structure, corresponding to that of input logical forms. The generation process can then be directed by the input semantic form. The inversion procedure is performed immediately before grammar compilation. The original $\mathcal{A}$MALIA's compiler then produces code for the inverted grammar using exactly the same machine language as for parsing grammars. Thus the same grammar can be compiled into two different object programs (abstract machine instructions) for the two different tasks (parsing and generation).

For generation, the input is a logical form represented as (an ALE description of) a feature structure. We assume such a form to have a nested predicate-argument structure, therefore it can be systematically decomposed into semantic meaning primitives (predicates and arguments) along the lines of this structure. This decomposition also induces some predefined ordering on the resultant sequence of meaning components. The chart is then initialized with complete edges that correspond to elements of this sequence rather than to words (as in the case of parsing). Thus each complete edge in the chart represents some part of the input semantic form.

Now we establish the similarity between parsing with ordinary grammars and generation with inverted ones. In both cases the input is divided into integral subparts, which are systematically consumed during parsing/generation according to the rules of the appropriate grammar. Chart processing algorithm can also be adapted for generation. Items originating from execution of a program created by the parsing compiler have their usual meaning. On the other hand, items generated by the execution of a program due to the generation compiler have a different meaning. They do not span a part of a given string as in the case of parsing; instead, they span a subform of the input semantic form, assigning it a structure that eventually determines a phrase whose meaning is that subform.

When the interpreter operates on a program that was generated from an inverted grammar, it executes the program in precisely the same way it would any other program – only the initialization of the machine's state and the format of the final results differ. Once the chart is initialized, the same processing strategy is applied independently of the task: the compiled program is executed on the input (also referred to as the *query*). The basic operation performed by the object program is *unification*, which is required for both tasks. Unification implements the *dot movement* operation which is essential to both chart parsing and generation. As explained above, dot movement is interpreted in distinct ways for the two tasks, since the (compiled) grammar rules are essentially different. It must be noted however, that the execution of the abstract machine is indifferent to this variation of interpretation – at run-time there is no notion of the particular task (parsing/generation), and the effect of the machine instructions is the same for both tasks. It is the ability to use the same chart core engine for both tasks that constitutes the most important feature of our work.

If generation terminates, the chart algorithm yields a (possibly empty) set of feature structures spanning the entire generation input; that is, the resultant FSs have been built using all the components of the given logical form. According to the grammar inversion algorithm, each such FS has a special-purpose feature (denoted *str* in the sequel) that encodes a list of words comprising the phrase generated. Values of this feature are tree-like structures, which are further processed by an auxiliary procedure (external to the chart algorithm and only applied in the case of generation) in order to obtain the generated phrase per se. When the *str* feature is ultimately used to reconstruct the generated phrase, no further use is made of the grammar.

With the above augmentation, $\mathcal{A}$MALIA[2] allows bidirectional processing of natural language grammars and comprises a unified platform for Natural Language Parsing and Generation. Using the *same* unification-based grammar for both analysis and generation is considered advantageous in the literature. Strzalkowski (1994) lists three options for grammar *reversibility*:

[2]From now on, we use the term "$\mathcal{A}$MALIA" to refer to the *extended* machine version capable of both parsing and generation.

- A grammar is compiled into two separate programs, parser and generator, requiring a different evaluation strategy;
- The parser and the generator are separate programs, executed using the same evaluation strategy;
- The parser and the generator are one program, and the evaluation strategy can run it in either direction.

AMALIA apparently realizes the second option: a single input grammar is compiled into two different (AM object) programs. The two programs are executed using exactly the same mechanism (the interpreter of abstract machine instructions), and therefore employ the same evaluation strategy. While there are indeed two different object programs, it should be observed that they are produced automatically by the compiler. As noted above, the differences between parsing and generation are limited to the initialization of the machine's state before the chart algorithm is invoked, and the interpretation of the final results thereafter.

Chapter 3

Inversion of Unification-Based Grammars for Generation

This chapter explains the process of grammar inversion, which transforms parsing grammars into a form more suitable for efficient generation. Section 3.1 presents motivation for using inversion and gives an overview of the inversion procedure. The way we encode grammars and the constraints imposed on grammar representation by the inversion algorithm are discussed in Section 3.2. Section 3.3 describes the sample grammar which serves as a running example for visualizing inversion steps; encoding of grammar rules in the TFS-based formalism is also shown in this section. Sections 3.4 and 3.5 explicate the two phases of grammar inversion, namely *normalization* and *inversion proper*, respectively.

3.1 Motivation and overview

Parsing grammars are usually given in a form oriented towards the analysis of a (linear) string and not towards generation from a (usually nested) logical form. In other words, grammar rules reflect the surface phrase structure and not the predicate-argument structure. Consider, for example, a phrase U = "John smokes", whose meaning is $f = \mathtt{smoke(john)}$, created with a rule[1] $NP\ VP \Rightarrow S$. The phrase can obviously be analyzed (parsed) using this rule in a straightforward manner. The meaning f of the phrase is obtained during parsing by applying the meaning of the verb ($\lambda x.\mathtt{smoke}(x)$) to that of the noun ($\mathtt{john}$), and performing a β-reduction: $\lambda x.\mathtt{smoke}(x)\,(\mathtt{john}) = \mathtt{smoke(john)}$.

On the other hand, if we want to generate U from its meaning f, it is difficult to predict that this particular rule should be used, as nothing in its body matches the nested structure of f. Examining the structure of f, we can discern the *predicate* `smoke` operating on the *argument* `john`, so that the application of the former to the latter forms a so-called *predicate-argument structure.* If the above rule could be restructured in a way to mirror this predicate-argument structure, for instance $VP(NP) \Rightarrow S$, and if the elements of the predicate-argument structure could be associated with a syntactic verb and noun, the body of the resultant rule would become readily applicable to the logical form f.

Such a transformation procedure was first proposed by Samuelsson (1995) for Definite Clause Grammars. The technique restructures ("inverts") grammar rules, so that their surface appearance resemble the nested structure of logical forms. The *inverted* grammar enables systematic reflection of any given logical form in the productions. Once the grammar is inverted, the generation process can be directed by the input logical form; elements of the input are consumed during generation just like words are consumed during parsing.

The process of grammar inversion is two staged:

1. First, each grammar rule is *normalized* – its constituent feature structures and the entire surface form are reorganized to form a predicate-argument structure. The normalization phase distinguishes

[1]For brevity sake only the context-free backbone of the rules is shown here.

between two types of rules:

(a) **Chain rules**, where the semantics of the rule head is reentrant with that of at most one body constituent, called the *semantic head.*

(b) **Non-chain rules**, otherwise.

Each constituent of an original rule is enhanced with two additional features:

(a) *args*, whose values are lists of arguments, which are passed between logical forms during derivations;

(b) *str*, whose values encode sequences of words spanned by the semantic form of the feature structure.

In chain rules, all the body constituents (apart from the semantic head) are converted into arguments of the semantic head; thus all such rules become *unit* rules. In non-chain rules, a provision is made for argument passing. The normalized versions of chain rules are referred to as *Argument-Filling (AF)*, and those of non-chain rules as *Functor-Introducing (FI)* ones.

Three auxiliary *rearrangement* rules are added to the final normalized grammar, which are ultimately removed in the inverted grammar. These rules are grammar-independent, and their sole purpose is to allow derivations with normalized grammars by handling the arguments passed between sentential forms.

2. The *inverted* grammar is then obtained from the normalized one, by concatenating normalized rules into chains and creating a new (*"inverted"*) rule from the two ends of each such chain. Specifically, the inversion phase examines various sequences (*chains*) of AF rules terminated with FI ones. The aim is to restructure the rules in the manner which allows systematical traversing (*"parsing"*) of logical forms. As a result, whenever the head of an inverted rule represents a logical form (in the predicate-argument notation, i.e., a predicate applied to its arguments), the rule body has a constituent corresponding to each of the head arguments, as well as an additional constituent corresponding to the predicate itself. This allows decomposition of complex logical forms into series of semantic primitives, and facilitates efficient generation by consuming the semantic predicate and then recursively consuming its arguments.

   The inversion phase evaluates *all* the possibilities to link together AF rules in chains ending with FI rules. Therefore, the inverted grammar contains only rules that correspond to the chains potentially useful for generation. It should also be noted that chain combination makes the rules along the chains more specific, due to the unifications performed in the process. Thus the *inverted* rules induced by these chains are at least as specific as their normalized predecessors linked in the chains.

3.2 Grammar representation and its restrictions

This section describes adaptation of the grammar inversion algorithm defined in (Samuelsson, 1995) to our TFS-based formalism. We exemplify all the grammar transformations discussed by performing them on a sample grammar used as a running example. In what follows, G_O is used as a meta-variable ranging over original grammars, and G_N and G_I for their respective normalized and inverted versions. G_O^e, G_N^e and G_I^e refer to the grammars of the running example.

3.2.1 Syntax representation

We encode input grammars with TFSs, where rules are represented using Multi-Rooted Structures (cf. Section 2.1). To ease the presentation we introduce a number of notational conventions. Throughout the document grammar rules are shown so that the rule head is always on the RHS and the body is on the LHS. Grammar rules are usually of the general form ($B_1B_2 \ldots B_m \Rightarrow H$), where H is the rule head and B_i are body constituents. PATR convention (Shieber, 1986) is employed for feature path values, thus $<$ *fs* $Feat_1$ $Feat_2$ $\ldots$ $Feat_n$ $>$ denotes the value located at the end of the feature path $Feat_1$ $Feat_2$ $\ldots$ $Feat_n$, starting from the topmost level of the feature structure *fs*. Whenever an unbound variable appears in a rule, its first occurrence is labeled with the most general type this variable can take. Optional reentrancy tags ($\boxed{\text{i}}$) denote values shared among several feature structures.

We assume that each FS in G_O has the features *syn* and *sem* defined, and that the former has the *cat* feature, whose values are categories of a context-free backbone. The *sem* feature encodes semantics which guides the process of grammar inversion and constitutes the input for generation; the *syn* feature is utilized by the grammar inversion algorithm (see Section 3.5.1). The formalism has no notion of the initial symbol, although one is supplied with G_O^e to demonstrate derivations; the $< \ldots syn\ cat >$ value of the initial symbol is denoted by **s**.

3.2.2 Semantics representation

Predicate-argument structure

We assume that the logical forms specified as meanings by the input grammar (G_O) are given in a so-called *predicate-argument* structure. Such a structure is analogous to the familiar representation of semantic logical forms with first-order terms. This way the semantics is built from basic units (FSs), each having a predicate and (optionally) a number of arguments. This assumption is crucial for both grammar inversion and generation. It allows the generation algorithm to decompose its input logical form into a sequence of meaning primitives (predicates and arguments), to be systematically consumed during generation. Accordingly, the grammar inversion algorithm relies on this assumption to restructure grammars into a form inherently suitable for generation. The predicate feature is denoted *pred* and the arguments (if present) are denoted as *arg1*, *arg2* etc. Predicates may be scalar[2] feature structures or complex ones, having several levels of nested FSs. Arguments may either appear on the same level as the predicate or be deeply embedded in one of the inner levels. The values appropriate for the features *pred* and arg_i are of type [**sem**] (the most general semantics type). During generation, semantic forms are decomposed into meaning primitives along the arg_i pointers of the predicate-argument structure.

We emphasize that the predicate-argument structure reflects the way semantic forms are built from argument primitives and predicates that act upon them. In grammar rules, we assume the semantics of the head to be formed of a *predicate* and (possibly) a number of arguments, while each argument corresponds to some body constituent. In MRS notation, the semantics of an argument is *reentrant* with that of the body constituent to which it corresponds. If there is also a body constituent corresponding to (and reentrant with) the predicate of the rule head, the rule is called a *chain* one; otherwise, the rule is *non-chain*.

Our formalism allows λ-abstractions over predicate-argument constructs. These are represented as FSs of the appropriate type (λ-**bind**), having two features: *var*, encoding the λ-variable, and *rest*, encoding the restriction on this variable. Values of the latter are either another λ-binder or a predicate-argument structure as defined above. Examples of encoding λ-expressions are shown in Section 3.2.4.

Consider, for instance, rule $O_2 = NP\ VP \Rightarrow S$ of the running example grammar (see Section 3.3), which presents an example of a *chain* rule :

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix} \textbf{syn} & \\ cat: & [\textbf{np}] \end{bmatrix} \\ sem: & \boxed{5}\,[\textbf{sem}] \\ str: & \boxed{3}\,[\textbf{list}] \end{bmatrix} \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix} \textbf{syn} & \\ cat: & [\textbf{vp}] \end{bmatrix} \\ sem: & \begin{bmatrix} \lambda\textbf{-bind} & \\ var: & \boxed{5} \\ \underline{rest}: & \boxed{6}\,[\textbf{funct}] \end{bmatrix} \\ str: & \boxed{4}\,[\textbf{list}] \end{bmatrix} \Rightarrow \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix} \textbf{syn} & \\ cat: & [\textbf{s}] \end{bmatrix} \\ sem: & \boxed{6} \\ str: & < \boxed{3}, \boxed{4} > \end{bmatrix}$$

The semantics of VP is given by $\lambda x.P(x)$, where the λ-variable x (designated with the tag $\boxed{5}$) is reentrant with the semantics of NP. This rule is a *chain* rule due to the reentrancy between $< S\ sem >$ and $< VP\ sem\ \underline{rest} >$, which is marked with the tag $\boxed{6}$.

β-reductions (which actually apply λ-expressions to their arguments) are *built into* grammar rules. During derivation, the predicate P and the λ-variable x in the above example are instantiated to specific semantic

[2]We refer to one-level (not nested) feature structures as *scalar* ones.

forms, and the resultant expression $P(x)$ becomes the semantics of S. This creates an apparent anomaly: instead of a λ-variable we have an instantiated expression, which is reentrant with the semantics of NP. This irregularity is eliminated when the body of the rule is replaced by its head.

In the MRS representing the rule, the semantics of the head $(P(x))$ is reentrant with that of the second body constituent modulo the λ-abstraction over the latter. The λ-binder disappears after the β-reduction incorporated into the rule, therefore we ignore it when checking for reentrancy between the head and body constituents.

The formalism also allows systematical encoding of second- and higher order functions, in addition to first-order ones. A sample encoding of a second-order expression is given in Section 3.2.4 (item 4).

Because of the predicate-argument structure of logical forms it is possible to analyze them as follows. Each such form is represented by a *semantic core* that contains the predicate and its arguments, possibly embedded inside an *envelope* of nested λ-binders abstracting over the core. In the above example, $P(x)$ is the semantic core of the λ-expression $\lambda x.P(x)$ (see also examples in Section 3.2.4 below).

Several steps of the normalization algorithm require direct access to the semantic core, bypassing the outer envelope. To this end we define the notion of a *designated path (DP)* which connects the semantic core to the topmost level of the feature structure. The designated path is comprised of the feature *sem* and a (possibly empty) contiguous sequence of *designated*[3] features ($\underline{rest}$): for a given feature structure f, its semantic core is $< f\ sem\ \underline{rest}^* >$, where $\underline{rest}^*$ denotes the sequence of designated features. Whenever we wish to emphasize the *semantic core* of an example FS, we mark it with double brackets: [[...]].

The notion of the *designated path* implies the test to distinguish between *chain* and *non-chain* rules. In the light of the above discussion, we need to check reentrancy between the designated paths of the head and of the body constituents of a rule. Observe that according to the definition, two different paths π_1 and π_2 are **reentrant** in a MRS A (denoted **reentrant**(π_1, π_2)) if they share the same value. For example, rule $O_2 = B_1 B_2 \Rightarrow H$ is a chain rule due to the reentrancy of the designated paths: **reentrant**$(< H\ sem >, < B_2\ sem\ \underline{rest} >)$. On the other hand, in rule O_3 the designated path of the head $(< H\ sem\ \underline{rest} >)$ is not reentrant with that of any body constituent, therefore the rule is non-chain.

To end this description of coding the semantics, we outline special treatment for quantified variables. Ultimately, the chart generation algorithm needs to be able to distinguish quantified variables, so that they are not entered into the chart during initialization. The reason is that since they do not correspond to any actual semantic primitive (see below), they cannot be used in scanning to match a body constituent of some rule. Hence we require that quantifiers explicitly introduce the variables they bind, by means of a *var* feature in the FS of the quantifier. The scope of a quantified variable defined this way is all the nested levels beneath the one in which it is introduced.

Semantic primitives

Logical forms, which represent meanings of sentences derivable by a given grammar, are encoded with a set of semantic primitives. Such a set is fixed for each grammar and can be deduced from the grammar specification. The primitives may be of one of two kinds: most of them represent meanings of lexical constants (e.g., `john`, `smoke`, `today`), while the rest serve as *connectives*[4] which aid in building complex meaning forms (e.g., `mod`, which modifies its first argument with the second one). Primitives of the latter kind are not directly related to any lexicon entry, therefore they need not have explicit representation in the generated string. Thus, such connectives demonstrate that there doesn't have to be a one-to-one correspondence between logical forms and surface structure. The role of connectives is to connect other meaning components together.

We therefore require that apart from the lexicon, a *Connective Registry* (CR) be supplied[5] with the grammar which encompasses the possible uses of the connectives (for instance, some may be applicable to constants, others to predicates etc.). A CR item is actually a connective *usage signature*, similar to the type signatures like $(e, (e, t))$ or $((e, t), t)$ used in Montague semantics (Gamut, 1991, Chapter 5). The definition

[3]The underlined font denotes the *designated* semantic feature ($\underline{rest}$) in AVMs of the running example.
[4]Not necessarily restricted to boolean connectives.
[5]In principle, the Connective Registry may be automatically deduced from the grammar specification. To simplify things, we currently supply it as a part of the input grammar.

of the *designated path* is naturally extended to also cover the CR items. Each CR entry is a feature structure giving the full representation of the primitive being defined.

An example of connective usage as well as a corresponding entry of the Connective Registry is given in Section 3.2.4. The specification of the sample grammar in Appendix A contains a Connective Registry as its last section.

Auxiliary (grammar-independent) features

Two additional features are required for the inversion algorithm to work:

1. To allow propagation of arguments between logical forms, the *args* feature encodes lists[6] of arguments which are collected along the chains of AF rules and passed to their respective functors (supplied by FI rules terminating the chains).

2. Since the rules of G_O are restructured during normalization (in particular, some body elements of the original rules are converted to arguments of the others), a mechanism is necessary to trace the constituent order of the original rule. It is this order that controls how words are combined to form phrases during generation. The *str* feature is used to achieve this aim by encoding ordered sequences of semantic primitives, each corresponding to some lexical item. A phrase spanned by a FS can be produced by analyzing the *str* elements, retrieving the respective words from the lexicon and concatenating them. Values of the *str* feature are of type [**list**].

Both auxiliary features are to be further explained in Section 3.4.

From a practical point of view, the grammar inversion module could automatically extend the type hierarchy of any given grammar with these features and their appropriate types. Nevertheless, in order to ensure consistency and *total well-typedness*[7] we require the auxiliary features and types to be already present in input grammars (see Section 3.2.3).

Feature coloring

For inversion to work correctly, it needs the ability to distinguish between various classes of features (i.e., syntactic, semantic etc.). To this end we introduce *"coloring"* of features that stem from the topmost level, so that all the features in the same class (e.g., *syn*, *sem* etc.) get the same color[8]. The classes need not be pairwise disjoint, and possible reentrancy among them results in *"multi-colored"* features. On the other hand, we do require that all the distinct classes present in a particular feature structure originate at the topmost level. For example, it is not allowed for semantic features to originate from features rooted at *syn*. It should be emphasized that both feature coloring and the choice of "designated" features have to be provided by the grammar designer. To demonstrate feature coloring, we fix an order in which features appear at the topmost level of all the FSs involved (in our example the order is *syn*, *sem*, *str*, *args*). The first feature in this order (*syn*) is assigned $color_1$, the second - $color_2$ and so on.

Presentation notes

Throughout the paper we utilize different fonts as follows:

- The Sans serif font is used for natural language phrases, e.g., "John smokes".
- **Boldface** is used for type names (e.g., **args**), and lowercase *italics* is used for feature names (e.g., *str*). In addition to that, the name of the designated feature is underlined (<u>*rest*</u>).
- Names of feature structures (e.g., f or α) and categories of the context-free backbone (e.g., VP or $AdvP$) are typeset in math mode.

[6]Actually, values of the *args* feature are *stacks*, since those are easier to implement within a TFS-based formalism.
[7]Cf. Section 2.1.
[8]In the running example, the features under *syn*, *sem*, *str* and *args* get colored in four different colors, groupwise.

- The `typewriter` font denotes logical forms (e.g., `smoke(john)`).
- Finally, the SMALL CAPS font is used to typeset product names (e.g., AMALIA or ALE).

3.2.3 Minimum required type hierarchy

The above assumptions induce some constraints on the type hierarchy of input grammars. It is therefore possible to formulate a fragment of the type hierarchy that every grammar must include in order to be invertible. The obligatory minimum type hierarchy in ALE notation is shown in Figure 3.1, and is depicted as an inheritance graph in Figure 3.2.

```
bot sub [sign, syn, syn_term, sem, args, list].
  sign sub [phrase].
    phrase sub [word] intro [syn:syn, sem:sem, args:args, str:list].
      word sub [].
  syn sub [] intro [cat:syn_term].
  syn_term sub [...].
    ...
  sem sub [const, funct].
    const sub [...].
      ...
    funct sub [atomic, quant].
      atomic sub[arg_1] intro [pred:sem].
        arg_1 sub [arg_2] intro [arg1:sem].
          arg_2 sub [arg_3] intro [arg2:sem].
            arg_3 sub [] intro [arg3:sem].
      quant sub [l_bind] intro [var:sem].
        l_bind sub [] intro [rest:sem].
  args sub [] intro [larg:list].
  list sub [ne_list, e_list].
    ne_list sub [] intro [hd:bot, tl:list].
    e_list sub [].
```

Figure 3.1: Minimum required type hierarchy in ALE notation.

Pursuant to ALE's convention, `bot` serves the root of the type hierarchy. Each type has a (possibly empty) set of subtypes listed after the `sub` keyword following the type name. Furthermore, each type may introduce a list of features which appear after the `intro` keyword, while each feature is followed by an appropriate type.

The type hierarchy shown has provisions for up to three arguments per predicate (arg_i), though it can easily be extended to incorporate more, should this become necessary for some particularly sophisticated grammar. The `list` part of the type hierarchy allows for encoding lists and other ordered sequences used in *args* and *str* features.

The ellipsis (...) denotes values not stipulated by the minimum required type hierarchy, since inversion and generation do not assume anything about them. For example, specific syntactic categories (values of the *cat* feature) may vary in actual grammars, and therefore are not shown. On the other hand, the inversion algorithm assumes that constituents of input grammar rules have the feature *syn*, whose values in turn has the feature *cat*. Hence both features are required in the type hierarchy, while the particular values of the latter are not.

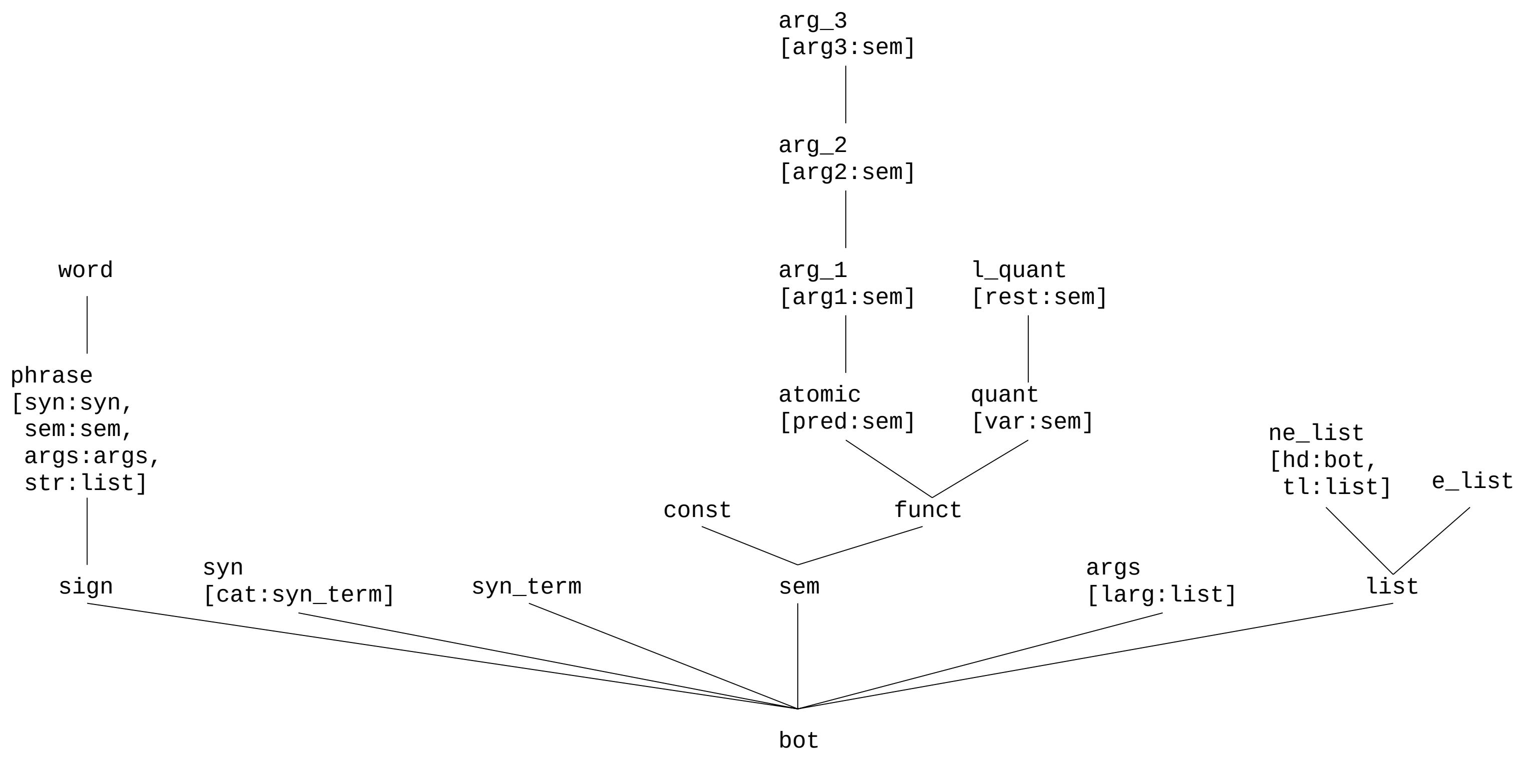


Figure 3.2: Minimum required type hierarchy depicted as a graph.

Currently, the name of the designated feature (*rest*) as well as other distinguished features and types (e.g., *syn*, *sem*, *pred*) are "hard-coded" into the type hierarchy. This way the compiler and the interpreter assume such features and types to be associated with known names. Should the need arise, these symbols can be turned into parametric, to be explicitly supplied with the input grammar.

3.2.4 Encoding examples

This section shows several examples of encoding linguistic information in our formalism.

Some of the feature structures in the examples below contain the *str* feature. To lessen the burden on the reader, the setting of this feature is explained later (see Section 3.4.6), and in the meanwhile the feature is shown only for the sake of completeness.

1. The following feature structure represents a lexical entry of the word "smokes", having the meaning $\lambda x.\texttt{smoke}(x)$:

$$fs_1^e = \begin{bmatrix} \textbf{word} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{vi}]\end{bmatrix} \\ sem: & \begin{bmatrix}\lambda\textbf{-bind} & \\ var: & \boxed{5}\,[\textbf{sem}] \\ \underline{rest}: & \boxed{1}\left[\begin{bmatrix}\textbf{arg\_1} & \\ pred: & [\textbf{smoke}]\\ arg1: & \boxed{5}\end{bmatrix}\right]\end{bmatrix} \\ str: & \left\langle \boxed{1}\right\rangle\end{bmatrix}$$

   The semantic constant [**smoke**] serves here as a *predicate* and the value of the feature *arg1* as its *argument.* The (nested) FS enclosed in double brackets is the *semantic core*, residing inside an envelope of a λ-abstraction. The feature *rest* is the *designated* one, and the *designated path* in this case is $< fs_1^e$ *sem* *rest* $>$.

2. Now compare the previous example with an AVM for the expression `smoke(john)` (which encodes the meaning of the sentence "John smokes"):

$$fs_2^e = \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix} \\ sem: & \boxed{1}\left[\begin{bmatrix}\textbf{arg\_1} & \\ pred: & [\textbf{smoke}]\\ arg1: & \boxed{5}\,[\textbf{john}]\end{bmatrix}\right] \\ str: & \left\langle \boxed{5}, \boxed{1}\right\rangle\end{bmatrix}$$

   Here there is no designated path as the entire *sem* value consists solely of the semantic core. The constant (actually, 0-ary predicate) [**john**] is the argument of the predicate [**smoke**]. To produce the phrase encoded by fs_2^e it is necessary to search the lexicon for words whose meanings are $\boxed{5}$ and $\boxed{1}$ respectively, and then to concatenate the resultant words in this order.

3. Next we consider the usage of connectives (cf. Section 3.2.2). Following below is a Connective Registry entry for the **mod** connective, which modifies an unsaturated predicate (having at least one λ-binder over it) with another meaning primitive. Note that the *syn* feature is defined to be as general as possible (merely for the sake of total well-typedness), since purely semantic connectives have no syntax.

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & \begin{bmatrix}\textbf{syn\_term}\end{bmatrix}\end{bmatrix} \\ sem: & \begin{bmatrix} \lambda\textbf{-bind} & \\ var: & \begin{bmatrix}\textbf{sem}\end{bmatrix} \\ \underline{rest}: & \begin{bmatrix}\begin{bmatrix}\textbf{arg\_2} & \\ pred: & \begin{bmatrix}\textbf{mod}\end{bmatrix} \\ arg1: & \boxed{6}\begin{bmatrix}\textbf{funct}\end{bmatrix} \\ arg2: & \boxed{7}\begin{bmatrix}\textbf{sem}\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Our next example uses the above connective to modify[9] the verb "smokes" with the adverb "today". According to rule O_3 of the running example grammar (see Section 3.3 below), this results in a phrase "smokes today". If the meaning of the verb is $\lambda x.\mathtt{smoke}(x)$ and that of the adverb is `today`, the meaning of the combined phrase is $\lambda x.\mathtt{mod(smoke}(x)\mathtt{,\ today)}$.

$$fs_3^e = \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & \begin{bmatrix}\textbf{vp}\end{bmatrix}\end{bmatrix} \\ sem: & \begin{bmatrix} \lambda\textbf{-bind} & \\ var: & \boxed{5}\begin{bmatrix}\textbf{sem}\end{bmatrix} \\ \underline{rest}: & \begin{bmatrix}\begin{bmatrix}\textbf{arg\_2} & \\ pred: & \begin{bmatrix}\textbf{mod}\end{bmatrix} \\ arg1: & \boxed{6}\begin{bmatrix}\textbf{arg\_1} & \\ pred: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ arg1: & \boxed{5}\end{bmatrix} \\ arg2: & \boxed{7}\begin{bmatrix}\textbf{today}\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix} \\ str: & \left\langle \boxed{6}, \boxed{7} \right\rangle \end{bmatrix}$$

4. The following two examples show feature structures created using the grammar of Appendix B.

   We demonstrate here how a phrase "good man" is obtained from the words "good" and "man", and how the meaning of the phrase is constructed in the process. Montague semantics for "man" is $\lambda x.\mathtt{man}(x)$, i.e. `man` is a first-order predicate. Prenominal adjectives are viewed by Montague approach as *second-order* functions, i.e., expressions which can be attached to a noun to form another noun. Thus, the compositionality principle requires the combined expression good(man) to be a first-order predicate as well, similarly to `man`. This fact should obviously be reflected by the semantics of the phrase "good man".

   Therefore, the meaning of "good man" should be $\lambda x.\,(\mathsf{good(man)})\,(x)$, where the meaning of "good" is applied to that of "man", and their combined meaning is applied to x, stating the fact that x is a "good man". In the light of the above, the phrase "good man" is composed of words good and man using the rule $T17 = (Adj\ CN \Rightarrow CN^{10})$ (see Appendix B).

   Observe, that arg_i pointers of semantic forms only reflect the predicate-argument structure, mirroring the way semantic forms are built from predicates (e.g., good) and arguments (e.g., `man`). In addition to that, it should be specified that x is a parameter of the complex expression $\lambda x.\,(\mathsf{good(man)})\,(x)$. Since `man` is the only argument of good, x cannot be made an additional argument. Otherwise, good would have two arguments and would become different in usage with `man`, which has only one argument (x).

   To overcome this apparent problem, the sample grammar introduces an additional feature $param_i$, which is employed to denote parameters of complex expressions. It should be emphasized that this

[9]We adopted the **mod** connective from (Samuelsson, 1995), so that there is a non-chain rule in the running example grammar (in addition to lexicon entries, which are trivially non-chain). Thus, verb modification is defined using the **mod** connective for expository purposes only, and does not reflect the accepted linguistic analysis. The Montague sample grammar (Appendix B) defines verb modification in the conventional way, by representing adverbs as second-order predicates which operate on verbs (first-order predicates).

[10]Here CN stands for *common noun*.

feature is neither assumed nor required by the inversion and generation algorithms. It is simply an ordinary feature used at the discretion of the grammar designer.

Now, a FS representing $\lambda x. (\texttt{good(man)})\,(x)$ has a feature $param_1$ pointing to x, *in addition to* the feature arg_1 pointing to `man`. To ensure uniformity of representation, a FS for $\texttt{man}(x)$ should also have two such features. Namely, arg_1 points to x denoting that the predicate `man` is applied to the argument x, while $param_1$ also points to x (actually, $param_1$ is reentrant with arg_1) to denote that x is the parameter of the expression $\lambda x.\texttt{man}(x)$.

Here is the feature structure representing the phrase "good man":

$$fs^e_4 = \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{cn}]\end{bmatrix} \\ sem: & \begin{bmatrix} \lambda\textbf{-bind} & \\ var: & \boxed{2}\,[\textbf{sem}] \\ \underline{rest}: & \left[\begin{bmatrix}\textbf{atomic\_1\_1} & \\ pred: & \boxed{1}\,[\textbf{good}] \\ arg1: & \boxed{7}\begin{bmatrix}\textbf{atomic\_1\_1} & \\ pred: & [\textbf{man}]\\ arg1: & \boxed{2}\\ param1: & \boxed{2}\end{bmatrix} \\ param1: & \boxed{2}\end{bmatrix}\right]\end{bmatrix} \\ str: & \left\langle \boxed{1},\boxed{7}\right\rangle \end{bmatrix}$$

where *cn* stands for "common noun".

The purpose of $arg1$ and $param1$ pointers in the semantic core is as follows. The composite expression $(\texttt{good(man)})\,(x)$ was obtained by applying the meaning of `good` to that of `man`, hence $<fs^e_4\ sem\ \underline{rest}\ arg1>$ points to the latter. On the other hand, x is the parameter of the entire expression $\lambda x. (\texttt{good(man)})\,(x)$, therefore $<fs^e_4\ sem\ \underline{rest}\ param1> = x$). The $param_1$ feature in this case denotes that the combined meaning of `good(man)` is applied to x; specifically, it is *not* the case that the meaning of `good` is applied to the combined meaning of $\texttt{man}(x)$).

We can also analyze the resulting expression as follows. The adjective "good" is a predicate-modifier (second-order function), and its Montague type is $((e,t),(e,t))$. When it is combined with the common noun "man" of type (e,t) (observe that the noun is substituted for the value of arg_1), the outcome is a first-order predicate "good man" of type (e,t). The $param_1$ feature points to the parameter x (of type e) of the result.

5. The last example encodes the meaning $\forall x(\texttt{man}(x) \rightarrow \texttt{smoke}(x))$ of the sentence "Every man smokes":

$$fs^e_5 = \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix} \\ sem: & \boxed{6}\left[\begin{bmatrix}\textbf{arg\_2} & \\ pred: & \begin{bmatrix}\forall\textbf{-quant} & \\ var: & \boxed{2}\,[\textbf{sem}] \\ scope: & \begin{bmatrix}\textbf{if} & \\ wff1: & \boxed{3}\begin{bmatrix}\textbf{atomic\_1\_1} & \\ pred: & [\textbf{man}] \\ arg1: & \boxed{2} \\ param1: & \boxed{2}\end{bmatrix} \\ wff2: & \boxed{7}\begin{bmatrix}\textbf{atomic\_1\_1} & \\ pred: & [\textbf{smoke}] \\ arg1: & \boxed{2} \\ param1: & \boxed{2}\end{bmatrix}\end{bmatrix}\end{bmatrix} \\ arg1: & \boxed{3} \\ arg2: & \boxed{7}\end{bmatrix}\right] \\ str: & \left\langle \boxed{6},\boxed{3},\boxed{7}\right\rangle \end{bmatrix}$$

where *wff* stands for "well-formed formula".

The logical form of this example consists of three integral parts, namely, the (schematic) predicate $\forall x(P(x) \to Q(x))$, and two arguments – $\mathtt{man}(x)$ and $\mathtt{smoke}(x)$. The latter two reside deep inside the FS representing the predicate, but are pointed at by the pointers $arg1$ and $arg2$ from the top predicate level. The expressions $\mathtt{man}(x)$ and $\mathtt{smoke}(x)$ in turn are also given in predicate-argument structure, whose sole argument is the universally quantified variable x. For the reasons explained in the previous example, both expressions have the $param_1$ feature pointing to their parameter (x).

The three constituent parts of the above logical form correspond to lexicon entries of the grammar, namely, to the entries for the words "every", "man", "smoke". The meaning of "every" is $\lambda P.\lambda Q.\forall x(P(x) \to Q(x))$, and during derivation P and Q are instantiated to the actual meaning predicates. The meanings of the other two words are $\lambda x.\mathtt{man}(x)$ and $\lambda x.\mathtt{smoke}(x)$, respectively.

Parsing the sentence "Every man smokes" proceeds as follows. First, the lexicon entries for the words "every", "man", "smoke" are retrieved. Rule $T3' = (Det\ N \Rightarrow NP)$ is then invoked, to apply the meaning of the determiner ("every") to that of the noun ("man"). A β-reduction built into the rule is performed, and P gets instantiated to $\mathtt{man}$. The resulting expression $\lambda Q.\forall x(\mathtt{man}(x) \to Q(x))$ gives the semantics of NP; it lacks a verb predicate to become a meaning of a complete sentence. Finally, rule $T2 = (NP\ VP \Rightarrow S)$ applies the semantics of NP to that of the verb. Consequently, Q is instantiated to $\mathtt{smoke}$, and the resultant feature structure is fs_5^e shown above.

The *str* feature of fs_5^e traces the word order of the original sentence. As explained above (Section 3.2.2), the *str* feature encodes series of semantics primitives, which originate in the semantics of lexicon entries. Therefore, $< fs_5^e\ str >$ lists three FSs, each corresponding to the semantic core of one of the lexicon entries involved.

3.3 The sample grammar

To visualize the steps of the inversion algorithm we apply it to a sample grammar G_O^e. Our running example is basically a small subset[11] of the grammar used in (Samuelsson, 1995), but represented in a TFS-based formalism. Appendix A lists the entire ALE-style (Carpenter, 1992a) specification of the grammar. This section comments on various design considerations in encoding the grammar with feature structures, and depicts MRSs for the grammar rules and lexicon items.

3.3.1 Sample grammar type hierarchy

The type hierarchy of the sample grammar is built around the minimum required type hierarchy shown in Section 3.2.3. Since all the auxiliary features and types required by the inversion algorithm are already included (e.g., the *str* feature necessary for tracing NL word order, and the *args* feature whose values hold lists of arguments passed between logical forms), the type hierarchy remains invariant throughout all grammar transformations (normalized and inverted).

3.3.2 Sample original grammar definition

The normalization procedure sets the *str* feature of G_O^e rules to record the original phrase structure. For expository purposes this setting is already incorporated in G_O^e rules shown below. The motivation and the exact way this setting is performed are described later, in Section 3.4.6.

Initial symbol

The *initial symbol* is only supplied here for explanatory purposes, to demonstrate derivations; there is no such notion in the formalism.

[11]We nevertheless keep the original rule numbers to make the reading easier for those familiar with (Samuelsson, 1995).

$$\left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{s}\right]\end{array}\right] \\ \textit{sem}: & \left[\textbf{sem}\right]\end{array}\right]$$

Grammar rules

O_2 This rule applies the semantics of VP to that of NP. β-reduction built into the rule MRS is performed, and the result becomes the semantics of S.

$$\left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{np}\right]\end{array}\right] \\ \textit{sem}: & \boxed{5}\left[\textbf{sem}\right] \\ \textit{str}: & \boxed{3}\left[\textbf{list}\right]\end{array}\right] \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{vp}\right]\end{array}\right] \\ \textit{sem}: & \left[\begin{array}{ll}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5} \\ \underline{\textit{rest}}: & \boxed{6}\left[\textbf{funct}\right]\end{array}\right] \\ \textit{str}: & \boxed{4}\left[\textbf{list}\right]\end{array}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{s}\right]\end{array}\right] \\ \textit{sem}: & \boxed{6} \\ \textit{str}: & <\boxed{3},\boxed{4}>\end{array}\right]$$

O_3 This rule modifies the semantics of VP with that of an adverb (AdvP). The **mod** connective combines the former with the latter, to form the semantics of the rule head.

$$\left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{vp}\right]\end{array}\right] \\ \textit{sem}: & \left[\begin{array}{ll}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5}\left[\textbf{sem}\right] \\ \underline{\textit{rest}}: & \boxed{6}\left[\textbf{funct}\right]\end{array}\right] \\ \textit{str}: & \boxed{3}\left[\textbf{list}\right]\end{array}\right] \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{advp}\right]\end{array}\right] \\ \textit{sem}: & \boxed{7}\left[\textbf{sem}\right] \\ \textit{str}: & \boxed{4}\left[\textbf{list}\right]\end{array}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{vp}\right]\end{array}\right] \\ \textit{sem}: & \left[\begin{array}{ll}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5} \\ \underline{\textit{rest}}: & \left[\begin{array}{ll}\textbf{arg\_2} & \\ \textit{pred}: & \left[\textbf{mod}\right] \\ \textit{arg1}: & \boxed{6} \\ \textit{arg2}: & \boxed{7}\end{array}\right]\end{array}\right] \\ \textit{str}: & <\boxed{3},\boxed{4}>\end{array}\right]$$

Lexicon

Although true for G^e_O, in general case there need not be a one-to-one correspondence between strings (words) and semantic primitives, for example in case of common lexical ambiguity.

O_8 The proper noun "John".

$$\text{“John”} \longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{np}\right]\end{array}\right] \\ \textit{sem}: & \boxed{1}\left[\left[\textbf{john}\right]\right] \\ \textit{str}: & <\boxed{1}>\end{array}\right]$$

O_{11} The intransitive verb "smokes".

$$\text{“smokes”} \longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \textit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \textit{cat}: & \left[\textbf{vi}\right]\end{array}\right] \\ \textit{sem}: & \left[\begin{array}{ll}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5}\left[\textbf{sem}\right] \\ \underline{\textit{rest}}: & \boxed{1}\left[\left[\begin{array}{ll}\textbf{arg\_1} & \\ \textit{pred}: & \left[\textbf{smoke}\right] \\ \textit{arg1}: & \boxed{5}\end{array}\right]\right]\end{array}\right] \\ \textit{str}: & <\boxed{1}>\end{array}\right]$$

O_{14} The adverb "today".

$$\text{“today”} \longrightarrow \begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix} \textbf{advp} \end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \boxed{1} \left[\!\left[\textbf{today} \right]\!\right] \\ \textit{str}: & < \boxed{1} > \end{bmatrix}$$

Connective Registry

- The **mod** connective.

$$\begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} \end{bmatrix} \\ \textit{sem}: & \begin{bmatrix} \lambda\textbf{-bind} & \\ \textit{var}: & \begin{bmatrix} \textbf{sem} \end{bmatrix} \\ \underline{\textit{rest}}: & \left[\begin{bmatrix} \textbf{arg\_2} & \\ \textit{pred}: & \begin{bmatrix} \textbf{mod} \end{bmatrix} \\ \textit{arg1}: & \begin{bmatrix} \textbf{funct} \end{bmatrix} \\ \textit{arg2}: & \begin{bmatrix} \textbf{sem} \end{bmatrix} \end{bmatrix}\right] \end{bmatrix} \\ \textit{str}: & \begin{bmatrix} \textbf{list} \end{bmatrix} \end{bmatrix}$$

3.3.3 Sample derivation with the original grammar

We exemplify below derivation of the sentence "John smokes today" with G^e_O.

In what follows we use the derivation relation as defined in (Wintner, 1997). All the feature structures along the derivation are as specific as needed. There may be reentrancy among FSs within any individual sentential form (denoted by usage of identical tags), but there is absolutely no reentrancy among FSs of *different* sentential forms.

$$\begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix} \textbf{s} \end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \begin{bmatrix} \textbf{arg\_2} & \\ \textit{pred}: & \begin{bmatrix} \textbf{mod} \end{bmatrix} \\ \textit{arg1}: & \boxed{6} \begin{bmatrix} \textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix} \textbf{smoke} \end{bmatrix} \\ \textit{arg1}: & \boxed{5} \begin{bmatrix} \textbf{john} \end{bmatrix} \end{bmatrix} \\ \textit{arg2}: & \boxed{7} \begin{bmatrix} \textbf{today} \end{bmatrix} \end{bmatrix} \\ \textit{str}: & < \boxed{5}, \boxed{6}, \boxed{7} > \end{bmatrix} \xrightarrow{(O_2)}$$

$$\begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix} \textbf{np} \end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \boxed{51} \begin{bmatrix} \textbf{john} \end{bmatrix} \\ \textit{str}: & < \boxed{51} > \end{bmatrix} \begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix} \textbf{vp} \end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \begin{bmatrix} \lambda\textbf{-bind} & \\ \textit{var}: & \boxed{51} \\ \underline{\textit{rest}}: & \begin{bmatrix} \textbf{arg\_2} & \\ \textit{pred}: & \begin{bmatrix} \textbf{mod} \end{bmatrix} \\ \textit{arg1} & \boxed{61} \begin{bmatrix} \textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix} \textbf{smoke} \end{bmatrix} \\ \textit{arg1} & \boxed{51} \end{bmatrix} \\ \textit{arg2} & \boxed{71} \begin{bmatrix} \textbf{today} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \textit{str}: & < \boxed{61}, \boxed{71} > \end{bmatrix} \xrightarrow{(O_3)}$$

$$\begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{np}]\end{bmatrix}\\ sem: & \boxed{52}\,[\textbf{john}]\\ str: & <\boxed{52}>\end{bmatrix}
\begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{bmatrix}\\ sem: & \begin{bmatrix}\lambda\textbf{-bind} & \\ var: & \boxed{52}\\ \underline{rest}: & \boxed{62}\begin{bmatrix}\textbf{arg\_1} & \\ pred: & [\textbf{smoke}]\\ arg1 & \boxed{52}\end{bmatrix}\end{bmatrix}\\ str: & <\boxed{62}>\end{bmatrix}
\begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{advp}]\end{bmatrix}\\ sem: & \boxed{72}\,[\textbf{today}]\\ str: & <\boxed{72}>\end{bmatrix}
\xrightarrow{(lex)}$$

"John smokes today"

3.4 Grammar normalization

Normalization is an auxiliary step towards inversion, which restructures the surface shape of grammar rules to match the predicate-argument structure of logical forms. Before G_O can be normalized, its rules have to be partitioned into *chain* and *non-chain* ones, as the two groups are treated differently by the normalization procedure.

3.4.1 Motivation

Since the inverted grammar is to be ultimately used for generation, let us consider the generation procedure in a nutshell.

The generation algorithm operates similarly to the SHDG procedure devised by Shieber et al. (1990). Non-chain rules serve as nodes in the generation tree[12], and get connected to other nodes by series of (zero or more) chain rules. The idea is to collect the arguments introduced by the chain rules, and to route them to their respective predicates. Whenever a non-chain rule terminates a sequence of chain rules, its body elements match the arguments collected along this sequence. It is this mechanism that allows recursive processing of logical forms during generation: application of a non-chain rule corresponds to consuming the predicate, then its arguments "rise" to the surface and are realized in the body elements. Each such element is then connected to lower nodes via series of chain rules etc.

In the general case, not all the arguments collected in the head of a non-chain rule are matched by the body elements. Let $B_1 B_2 \ldots B_l \Rightarrow H$ be a non-chain rule, such that the semantics of H is of the form $< H\ sem >= P(< B_1\ sem >, \ldots, < B_l\ sem >)$, where P is a predicate. The argument list $< H\ args >$ may contain more than l elements — some of them are to be matched against the constituents of the rule body, while others are to be passed further downwards. Each element of $< H\ args >$ whose semantics serves an argument of P is matched with the corresponding body constituent. The remaining elements of the $< H\ args >$ list need to be passed further downward to their respective predicates. Thus, non-chain rules may either "consume" (match) arguments accumulated in the head, or pass them further on. Specifically, non-chain rules never introduce any new arguments of their own.

3.4.2 Grammar normal form

The generation algorithm outlined above induces the inversion procedure, which transforms the grammar into a form suitable for generation. For the inversion algorithm to work correctly, the input grammar has to be brought to a so-called *"normal form"*, which facilitates the above scenario of rule invocation. To this end, *chain* rules are transformed into *unit* rules: the semantic head is left as the only body constituent, while all the rest are moved to its argument list. This allows to concatenate the resultant unit rules into *chains*, which collect the arguments for the *non-chain* rules to consume. The arguments passed to the non-chain rules are

[12]Like a parsing tree, the *generation tree* is a convenient way to graphically illustrate generation process; it is built by depicting the rules applied to the form being generated from.

consumed by matching them with the rule body constituents. To simplify the matching, bodies of non-chain rule are rearranged, so that the order of constituents mirror the argument order in the head semantics. Upon this transformation the role of each constituent is defined by its position. Then each constituent can be straightforwardly matched with its counterpart from the head argument list, since both correspond to the same argument of the head logical form. Lexicon entries of G_O can be viewed as non-chain rules (since they have no body constituents whose semantics is reentrant with that of the head), and are therefore normalized along with other non-chain rules.

3.4.3 Normalization of G_O

In what follows we describe normalization of different kinds of G_O rules. In order not to inflate G_N unnecessarily, whenever a new normalized rule is about to be created, it is compared with the existing ones. For each such *candidate* rule, the algorithm checks whether a more general rule (from the unification point of view) has already been created. If so, the new rule is discarded, otherwise it either replaces the existing rule (if the latter is more specific) or is added as a new one (if it is neither subsumed by nor subsumes any existing rule). This functionality is realized in the normalization algorithm below by an auxiliary procedure **add_rule**. Since this routine is used by the inversion algorithm as well, it receives the grammar as an explicit parameter, in addition to the rule to be checked.

Normalization of chain rules

According to the usual definition (e.g., (Shieber et al., 1990, p.34)), a rule is a chain rule if the logical form of its head is reentrant with that of one of the body elements, called the *semantic head*[13]. To facilitate further explanation let $B_1 B_2 \ldots B_m \Rightarrow H$ be a generic chain rule, where $< H\ sem >$ is reentrant with $< B_k\ sem >$ such that $1 \leq k \leq m$, and hence B_k is the semantic head. Let us consider the predicate-argument structure of $< H\ sem >$. According to the definition of semantic head, $< B_k\ sem >$ serves the predicate which acts on the semantics of the rest of body constituents as arguments: $< H\ sem > = < B_k\ sem > (< B_1\ sem >, \ldots, < B_{k-1}\ sem >, < B_{k+1}\ sem >, \ldots, < B_m\ sem >)$. The normalization changes the surface form of the rule accordingly, to match the nested structure of $< H\ sem >$. Chain rules are modified to only leave the semantic head as a single constituent in the body, while the rest of the body elements are added to the list[14] of its arguments. Thus, chain rules are restructured to have a single FS in the body, so that the inversion phase may combine the resultant *unit* rules (called *Argument-Filling rules*) into longer chains.

Normalization of non-chain rules

Non-chain rules are also modified for argument handling. To enable correct argument flow, the normalization algorithm needs to determine for each rule the body constituent that receives any "excessive" arguments passed from the head. We call such a constituent the *argument carrier*[15]. The argument carrier is determined manually[16], to ensure[17] that the arguments are eventually passed to their respective predicates. The normalized versions of non-chain rules are called *Functor-Introducing rules*, since they "introduce" functors (cf. predicate P in the above example) which consume arguments collected along the chains of AF rules. The latter "fill" the functors with their arguments, hence the name of *Argument-Filling* rules.

Further modifications to non-chain rules include restructuring their body to match the argument order in the head. That is, the new rule body is $B_{i_1} \ldots B_{i_l}$, where B_{i_j} is reentrant with j-th argument of $< H\ sem >$. This is performed to establish the order in which the arguments of the head are matched in the body

[13]Here we rely on the assumption made in Section 3.1 that chain rules may have no more than one semantic head.

[14]Actually, the arguments added by the chain rule are *pushed onto* the argument stack of the semantic head.

[15]We currently assume that there may be only one argument carrier in each non-chain rule. This assumption is essentially equivalent to the analogous assumption that chain rules may have only one semantic head.

[16]Each non-chain rule in the input grammar is accompanied by a marker that indicates the argument carrier for this rule. These markers are implemented as comments of ALE specification language, and are therefore transparent to other applications that may use the same grammar.

[17]In Section 3.5.1 we explain why it is important to correctly determine the argument carrier.

elements. Also, a new body constituent is created after B_{i_l}, which is reenrant with the entire $< H\ sem >$; we denote this last body element as $[H]$ and call it *the semantics constituent.* Thus the normalized FI rule is $B_{i_1} \ldots B_{i_k}(A) \ldots B_{i_l}\ [H] \Rightarrow H(A)$, where A is a generic argument list and B_{i_k} is the argument carrier.

To understand the motivation for this transformation, consider bottom-up generation from a logical form $f = P(a_1, \ldots, a_l)$. As mentioned above, our generation algorithm is chart-based. Therefore, rules are applied during generation using the chart operations of *dot movement* and *completion* (see Section 4.1). In particular, the former operation matches the body of the rule against the contents of the chart, and the latter creates a new chart item representing the rule head if the rule application is successful.

The generator flattens[18] the given logical form f and produces a sequence of items $s = [a_1, \ldots, a_l, P]$. These items are then used to initialize the chart. Recall that inverted rules are made from chains of AF rules terminated with a FI one, thus the body of an inverted rule is due to some Functor-Introducing rule in G_N. To apply such a rule using dot movement, it is necessary to match its body constituents against the elements of s. This is when the apparently strange restructuring of non-chain rules begins to work. Since both the reordered rule body *and* the generation input (the sequence s of meaning components) have *exactly the same order*, they can be matched in a straightforward manner. The last body constituent is necessary to match the last element of s, namely the predicate itself. This constituent allows for propagation of predicate semantic forms through inverted grammar rules, since the semantics of the predicate is shared by all the AF rules along the chain.

To enable proper treatment of arguments as described above, the feature *args* is employed. Its values (of type **args**) have the only feature – *larg* – which holds a list of arguments.

We shall presently outline the normalization of lexicon entries, but first we consider an example emphasizing the above principles.

A normalization example

Let us consider an example of normalization. We use parenthesized notation to abbreviate feature structures enclosed as arguments. For instance, rule $O_2 = NP\ VP \Rightarrow S$ of G_O^e is normalized to the Argument-Filling rule $N_2 = VP(NP) \Rightarrow S$. The meaning of the resultant rule is that the FS representing the VP has a one-element argument list, which contains a FS representing the NP. Similarly, a mere (NP) would mean a FS for an NP wrapped in an envelope of type **args**. Rule $O_3 = VP\ AdvP \Rightarrow VP$ induces the Functor-Introducing rule $N_3 = VP(A)\ AdvP\ [VP] \Rightarrow VP(A)$. Three additional FI rules are created from the lexicon: $N_8 = A\ [NP] \Rightarrow NP(A)$, $N_{11} = A\ [VP] \Rightarrow VP(A)$ and $N_{14} = A\ [AdvP] \Rightarrow AdvP(A)$.

Derivation of the sentence "John smokes today" from a FS with the logical form $f =$ `mod(smoke(john), today)` may pursue the following scenario. Using rule N_2, the initial symbol S with semantics f goes to $VP(NP)$. Rule N_3 is then applied to create $VP(NP)\ AdvP$ [**mod**]; observe[19] how the NP argument is routed to the VP constituent of the N_3 body, which acts as an argument carrier. Rule N_{11} is invoked on the resultant sentential form to consume the VP predicate, thus leaving (NP) [**smoke**] $AdvP$ [**mod**]. An auxiliary rearrangement rule R_2 (see the definition in Section 3.4.5 below) extracts the NP from its envelope of type **args** and leaves NP [**smoke**] $AdvP$ [**mod**]. At last, the remaining FSs for NP and $AdvP$ are exhausted by rules N_8 and N_{14}, resulting in the sentential form [**john**] [**smoke**] [**today**] [**mod**]. Observe that derivation with G_N terminates with a sequence of meaning primitives, which together comprise the input logical form.

Normalization of the lexicon

Lexicon entries can be viewed as non-chain rules with an empty body, and their semantics is treated as a (possibly 0-ary) functor with arguments to be filled. Thus, lexicon items induce a set of additional Functor-Introducing rules, also called *Lexicon-Derived (LD) rules.* If "word" $\rightarrow H$ is a lexicon entry, the corresponding FI rule is of the form $A\ [H] \Rightarrow H(A)$ so that $< H\ args >$ is reentrant with A. That is, the entire argument

[18]Flattening of logical forms is performed in *postorder*, placing first the arguments, then the predicate.

[19]Although not obvious here due to the briefness of notation, rule N_3 also consumes the **mod** predicate – see Section 3.4.9 for a full derivation example on feature structures.

list A collected in the head forms the body of the new rule. The body of Lexicon-Derived rules is a single feature structure of type **args** encoding a list of arguments; it is flattened to actually form a list of elements using *rearrangement rules* (see Section 3.4.5 below). Thus, Lexicon-Derived rules constitute the "ultimate" destination of arguments accumulated along the chains, as they can be passed no further.

In order to minimize the number of Lexicon-Derived rules and reduce the number of specific semantic forms present in such rules, the lexicon is normalized in a slightly more sophisticated way than that just described above. The normalization procedure analyzes lexicon entries by abstracting over particular semantic constants. If the semantic core of a given lexical item is either scalar or is a predicate-argument construct whose predicate is scalar, that scalar constant is replaced by the lowermost (non-terminal) supertype from the type hierarchy. The outcome becomes a *candidate* for a new FI rule. The idea here is to produce one rule to treat all nouns, another to treat all intransitive verbs, etc. This allows to greatly reduce the number of lexicon-derived Functor-Introducing rules, compared with the straightforward normalization[20] outlined in (Samuelsson, 1995).

For instance, consider the lexicon entry O_8. To form a new rule, the semantic constant **john** in O_8 is replaced with its supertype **pn** (which stands for "proper noun"). In item O_{11}, the type **v_intrans** is substituted for the semantic predicate **smoke**, and the normalized rule N_{11} is then created.

In addition to initiating a set of lexicon-derived rules (as described above), the lexicon of the original grammar also stays in both G_N and G_I, to supply the information about association of NL words and their semantic meanings.

3.4.4 Observations

We now further explain and refine some of the normalization steps. Let us review normalization of chain rules into Argument-Filling rules. As described in Section 3.2, feature structures in the rules of G_O have the feature *cat*, which denotes categories of the context-free backbone of the grammar. When multiple body constituents become arguments of the semantic head and are pushed onto its argument stack, the body of the normalized AF rule is a feature structure having *several cat* values (at various depths and via different paths). For example, observe rule N_2 below, where the NP became an argument of the VP. The single body element contains two *cat* values, one for each original constituent: $< B_1$ *args larg hd syn cat* $>=$ **np** and $< B_1$ *syn cat* $>=$ **vp**, where B_1 denotes the feature structure in the rule body.

The initial symbol persists unchanged through all grammar transformations.

3.4.5 Rearrangement rules

The grammar G_N is endowed with a set of G_O-independent auxiliary rearrangement rules, which are required for proper argument processing in derivations under G_N. These rules flatten argument lists as described below.

R_1 This rule serves to extract information from inside an "envelope" of type **args**; it treats argument lists of length more than 1.

$$\begin{bmatrix}\textbf{args}\\ larg: \boxed{1}\begin{bmatrix}\textbf{ne\_list}\\ hd: \begin{bmatrix}\textbf{phrase}\end{bmatrix}\\ tl: \begin{bmatrix}\textbf{list}\end{bmatrix}\end{bmatrix}\end{bmatrix} \boxed{2}\begin{bmatrix}\textbf{phrase}\end{bmatrix} \Longrightarrow \begin{bmatrix}\textbf{args}\\ larg: \begin{bmatrix}\textbf{ne\_list}\\ hd: \boxed{2}\\ tl: \boxed{1}\end{bmatrix}\end{bmatrix}$$

R_2 This is the terminal case of information extraction from an **args** envelope. This rule treats lists of length 1, i.e., those having only a non-empty head and an empty tail.

[20] In his article, Samuelsson (1995) briefly mentions that a similar approach for lexicon processing is possible for his LR grammar compilation for generation. Since we work in a unification-based formalism, using the type hierarchy information to make grammar rules as general as possible is a natural thing to do.

$$\boxed{1}\begin{bmatrix}\textbf{phrase}\end{bmatrix} \Longrightarrow \begin{bmatrix}\textbf{args} & \\ larg: & \begin{bmatrix}\textbf{ne\_list} & \\ hd: & \boxed{1} \\ tl: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

R_3 This rule discards empty argument envelopes.

$$\lambda \Longrightarrow \begin{bmatrix}\textbf{args} & \\ larg: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix}\end{bmatrix}$$

where λ is the empty feature structure.

3.4.6 Grammar preprocessing

The *str* feature

An additional transformation performed by the normalization algorithm involves setting the *str* feature, which serves for recording the original phrase structure of the G_O rules. This is performed as a preprocessing step, since the normalization changes the surface order of rule constituents.

The intuition for *str* usage is that tracing its values ultimately enables us to recreate strings spanned by various logical forms. The feature is used differently for regular rules and lexical items: in grammar rules it relates to a substring a particular FS eventually derives, while in lexical items such substrings are available immediately. The *str* values of various FSs may be concatenated on various stages of grammar processing, and thus they are of type **list**. In order to do without an `append` operation, the *str* values are represented as tree-like structures. Thus, to concatenate s_1 and s_2, a new list element s is created of type **ne_list**, so that $< s\ hd >= s_1$ and $< s\ tl >= s_2$:

$$\begin{bmatrix}\textbf{ne\_list} & \\ hd: & s_1 \\ tl: & s_2\end{bmatrix}$$

where in the general case s_1 and s_2 are *tree-like* graphs whose edges are labeled with features *hd* and *tl*. Branches of such trees are terminated with an [**e_list**] construct. For instance, a feature structure encoding a one-element list is depicted below:

$$\begin{bmatrix}\textbf{ne\_list} & \\ hd: & s_3 \\ tl: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix}\end{bmatrix}$$

This structure enables flattening trees into *linear* lists by recursively flattening the elements of s_1 and then the elements of s_2. To keep the feature structures in this document compact, we abbreviate the tree-like structures with their flattened forms (though actual FSs encode the *str* values as trees, as explained above). For instance, $< s_1^1, \ldots, s_1^m, s_2^1, \ldots, s_2^n >$ might encode the elements of s_1 concatenated with those of s_2. Whenever we want to outline the true structure of an *str* value, we use the notation $<< s_1^1, \ldots, s_1^m >, < s_2^1, \ldots, s_2^n >>$ to emphasize which elements actually belong to s_1 and which to s_2.

- In grammar rules the *str* feature of each body constituent is marked with a tag (e.g., $\boxed{3}$, $\boxed{4}$ etc.). The *str* value of the rule's head is then set to the concatenation of body's tagged values, in the order in which their respective constituents appear in the rule's body. In other words, the purpose of the *str* feature in the head of grammar rules is to store the surface linear order of the body constituents. For example, in rule O_2 of the running example grammar (see Section 3.3), the value of the head *str* feature is $< \boxed{3}, \boxed{4} >$, which is a concatenation of the value $\boxed{3}$ of the left body element and $\boxed{4}$ of the right body element. The *str* feature becomes crucial in G_N and G_I. For instance, normalization of rules may change their structure so that some body constituents become hidden inside the semantic head. Thus, the surface representation of the body changes, while its original form is still stored in the head *str* value. For illustration, consider the rules of G_N^e in Section 3.4.8.

- In lexical items the head of the *str* list points to the *core* of the semantics expression (which resides at the end of the designated path starting at the topmost level). This makes logical forms be the primitive elements of *str* lists. Given such a list, logical forms may be mapped onto words, resulting in the string derived/generated.

3.4.7 Normalization algorithm

In what follows we describe transformations applied to each type of G_O rules. During the conversion process each normalized rule is marked as an AF or FI one, for the inversion phase.

0. AUXILIARY DEFINITIONS

   / Procedure* **add_rule** *receives a grammar G and a "candidate" rule r, and checks whether G already contains a rule more general than r. If so, the "candidate" rule r is discarded, otherwise it either replaces the existing rule (if the latter is more specific) or is added as a new one (if it is neither subsumed by nor subsumes any existing rule). */*

   procedure add_rule(G, r)

   (a) If $\exists r' \in G : r' \sqcup r \neq \top$ then
      i. If $r \sqsubseteq r'$ then replace r' with r: $G = (G \setminus \{r'\}) \cup \{r\}$
      ii. If $r' \sqsubseteq r$ then discard r.

   (b) Otherwise, add r to G: $G = G \cup \{r\}$.

   end /* procedure add_rule */

1. PREPROCESSING STEP

   For every $r = B_1 \ldots B_m \Rightarrow H$ in G_O:

   (a) Set $< H\ str > = < B_1\ str > \cdot < B_2\ str > \cdot \ldots \cdot < B_m\ str >$,
   where $\cdot$ denotes concatenation.
   Comment: Record the original phrase structure.

   (b) If $\exists k, 1 \leq k \leq m$ such that **reentrant**$(< H\ sem\ \underline{rest}^* >, < B_k\ sem\ \underline{rest}^* >)$,
   mark r as *chain*, otherwise mark r as *non-chain*. In case of a chain rule, mark B_k as the *semantic head* of r.
   Comments:
   - According to the definition of chain rule in Section 3.1, no more than one such k as above exists in each rule.
   - This induces a partition on G_O :
     $G_{O_{chain}} \cup G_{O_{nonchain}} = G_O$ (obviously, $G_{o_{chain}} \cap G_{O_{nonchain}} = \emptyset$).

   (c) $G_N := \emptyset$

2. NORMALIZATION OF NON-CHAIN RULES.

   For every r in $G_{O_{nonchain}}$:

   (a) Let $SC = < H\ sem\ \underline{rest}^* >$ be the semantic core of H.
   Then $r_N = B_{i_1} \ldots B_{i_l}(A) \ldots B_{i_m}\ [H] \Rightarrow H(A)$,
   where $[H] = < H\ sem >$ (the semantics constituent);
   $\forall k, 1 \leq k \leq m :$ **reentrant**$(< B_{i_k}\ sem\ \underline{rest}^* >, < SC\ arg_k >)$;
   B_{i_l} is the manually designated argument carrier, therefore set $< B_{i_l}\ \ args\ \ larg > = < H\ args\ larg >$ (the generic argument list $A = < H\ args\ larg >$ denotes this reentrancy) and $\forall j, 1 \leq j \leq m, j \neq l : < B_{i_j}\ args\ larg >= [$**e_list**$]$.
   Comment: Restructure the rule body to match the head argument order.

(b) Mark r_N as Functor-Introducing and attempt to add it to the grammar: **add_rule**(G_N, r_N).

3. Normalization of chain rules.

   For every r in $G_{O_{chain}}$:

   (a) Let B_k be the semantic head: **reentrant**$(< H\ sem\ \underline{rest}^* >, < B_k\ sem\ \underline{rest}^* >)$.

   - If $m = 1$, then let $r_N = r$ and set $< B_1\ args > = < H\ args >$.
   - Otherwise, $r_N = B_k(B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_m, A) \Rightarrow H(A)$,
     where $< B_k\ args\ larg\ hd > = B_1$ and
     $< B_k\ args\ larg\ tl > = (B_2 \cdot \ldots \cdot B_{k-1} \cdot B_{k+1} \cdot \ldots \cdot B_m \cdot A)$
     ($A = < H\ args\ larg >$ denotes a generic argument list and $\cdot$ denotes concatenation).
     $\forall j, 1 \leq j \leq m, j \neq k$: set $< B_j\ args\ larg > = [\mathbf{e\_list}]$.

   Comment: Convert the body elements into arguments of the semantic head.

   (b) Mark r_N as Argument-Filling and attempt to add it to the grammar: **add_rule**(G_N, r_N).

4. Normalization of lexicon entries.

   For every lexicon entry $l =$ ("word" $\rightarrow H$):

   (a) Let $r_N = A\ [H'] \Rightarrow H'$,
   where $A = < H'\ args >$ is a generic argument list,
   $[H'] = < H'\ sem >$ (the semantics constituent, created as in the case of FI rules),
   and H' is created from H as follows:

   - If the semantic core of H is either scalar or is a predicate-argument structure whose *pred* value is scalar, this scalar constant is replaced with its lowermost (non-terminal) supertype from the type hierarchy.
   - Otherwise, $H' = H$.

   Comment: Create only "representative" Lexicon-Derived rules, which correspond to several semantic primitives.

   (b) Set $< H'\ str > = < H'\ sem\ \underline{rest}^* >$.

   Comment: The *str* feature value in the head of a Lexicon-Derived rule is associated with the semantic core of the head.

   (c) Mark r_N as Functor-Introducing and attempt to add it to the grammar: **add_rule**(G_N, r_N).

5. $G_N = G_N \cup \{R_1, R_2, R_3\}$

   Comment: Add the rearrangement rules to G_N.

The algorithm ends up with a normalized grammar G_N, which consists of three pairwise disjoint sets of rules $G_N = G_{FI} \cup G_{AF} \cup G_R$, where G_{FI} is a set of Functor-Introducing rules, G_{AF} is a set of Argument-Filling rules and G_R is a set of auxiliary rearrangement rules.

According to the normalization algorithm, each rule $r_N \in G_N \setminus G_R$ corresponds to a rule (or a lexicon item) $r \in G_O$ that licensed its creation. In such a case, r is referred to as the **prototype** of r_N, and r_N is the **image** of r.

3.4.8 Normalized sample grammar

The normalized grammar of the running example looks as follows. The numbering of G_N^e rules corresponds to that of their prototypes in G_O^e. Whenever appropriate, G_N^e rules are accompanied with a comment on how they have been obtained from the corresponding original rules.

Initial symbol

Exactly the same as in G^e_O.

Functor-Introducing rules

N_3 The semantics of the rule head is not reentrant with that of any body constituent, hence the prototype rule is a *non-chain* one. An additional body constituent (the so-called *semantics constituent*) is created in the last position, which is made reentrant with the head semantics. The first body constituent is the manually designated *argument carrier*, therefore $< B_1\ args\ larg >=< H\ args\ larg >$.

$$\left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{array}\right] \\ sem: & \left[\begin{array}{ll}\lambda\textbf{-bind} & \\ var: & \boxed{5}\,[\textbf{sem}] \\ \underline{rest}: & \boxed{6}\,[\textbf{funct}]\end{array}\right] \\ str: & \boxed{3}\,[\textbf{list}] \\ args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & \boxed{9}\,[\textbf{list}]\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{advp}]\end{array}\right] \\ sem: & \boxed{7}\,[\textbf{sem}] \\ str: & \boxed{4}\,[\textbf{list}] \\ args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{array}\right]\end{array}\right] \boxed{17} \left[\begin{array}{ll}\lambda\textbf{-bind} & \\ var: & \boxed{5} \\ \underline{rest}: & \left[\begin{array}{ll}\textbf{arg\_2} & \\ pred: & [\textbf{mod}] \\ arg1: & \boxed{6} \\ arg2: & \boxed{7}\end{array}\right]\end{array}\right] \Longrightarrow$$

$$\left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{array}\right] \\ sem: & \boxed{17} \\ str: & << \boxed{3} >, < \boxed{4} >> \\ args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & \boxed{9}\,[\textbf{list}]\end{array}\right]\end{array}\right]$$

Comment: Rules N_8, N_{11} and N_{14} below are regular grammar rules (though lexicon-derived in their nature). In addition to these, the lexicon items O_8, O_{11} and O_{14} constitute the lexicon of the normalized grammar (see below).

N_8 Lexicon items are normalized into Functor-Introducing rules. The *semantics constituent* is created as the second body element, and is made reentrant with the head semantics.
This lexicon-derived rule covers the case of *proper nouns*. The semantic constant **john** in the original lexicon item has been replaced with **pn**, which is the greatest type subsuming the former. No other FI rule created thus far unifies with this one, hence a new FI rule (N_8) is created.

$$\boxed{5}\left[\begin{array}{ll}\textbf{args} & \\ larg: & [\textbf{list}]\end{array}\right] \boxed{1}\,[\textbf{pn}] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{np}]\end{array}\right] \\ sem: & \boxed{1} \\ str: & < \boxed{1} > \\ args: & \boxed{5}\end{array}\right]$$

N_{11} This lexicon-derived rule covers the case of *intransitive verbs* (**v_intrans**).

$$\boxed{6}\left[\begin{array}{ll}\textbf{args} & \\ larg: & [\textbf{list}]\end{array}\right] \boxed{17} \left[\begin{array}{ll}\lambda\textbf{-bind} & \\ var: & \boxed{5}\,[\textbf{sem}] \\ \underline{rest}: & \boxed{1}\left[\begin{array}{ll}\textbf{arg\_1} & \\ pred: & [\textbf{v\_intrans}] \\ arg1 & \boxed{5}\end{array}\right]\end{array}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{vi}]\end{array}\right] \\ sem: & \boxed{17} \\ str: & < \boxed{1} > \\ args: & \boxed{6}\end{array}\right]$$

N_{14} This lexicon-derived rule covers the case of *adverbs* (**adv**).

$$\boxed{5}\begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{list}]\end{bmatrix}\ \boxed{1}\,[\textbf{adv}] \Longrightarrow \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{advp}]\end{bmatrix}\\ sem: & \boxed{1}\\ str: & <\boxed{1}>\\ args: & \boxed{5}\end{bmatrix}$$

Argument-Filling rules

N_2 The semantics of the head in rule O_2 is reentrant with that of the second body constituent (the *semantic head*), hence the prototype rule is a *chain* one. The first body constituent is thus moved to the argument list of the semantics head (more precisely, it is pushed upon the generic list of arguments, denoted with $\boxed{7}$).

$$\begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{bmatrix}\\ sem: & \begin{bmatrix}\lambda\textbf{-bind} & \\ var: & \boxed{5}\,[\textbf{sem}]\\ \underline{rest}: & \boxed{6}\,[\textbf{funct}]\end{bmatrix}\\ str: & \boxed{4}\,[\textbf{list}]\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & \begin{bmatrix}\textbf{ne\_list} & \\ hd: & \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{np}]\end{bmatrix}\\ sem: & \boxed{5}\\ str: & \boxed{3}\,[\textbf{list}]\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix}\\ tl: & \boxed{7}\,[\textbf{list}]\end{bmatrix}\end{bmatrix}\end{bmatrix} \Longrightarrow \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix}\\ sem: & \boxed{6}\\ str: & <<\boxed{3}>,<\boxed{4}>>\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & \boxed{7}\end{bmatrix}\end{bmatrix}$$

Lexicon

The lexicon (rules O_8, O_{11} and O_{14}) of G^e_O.

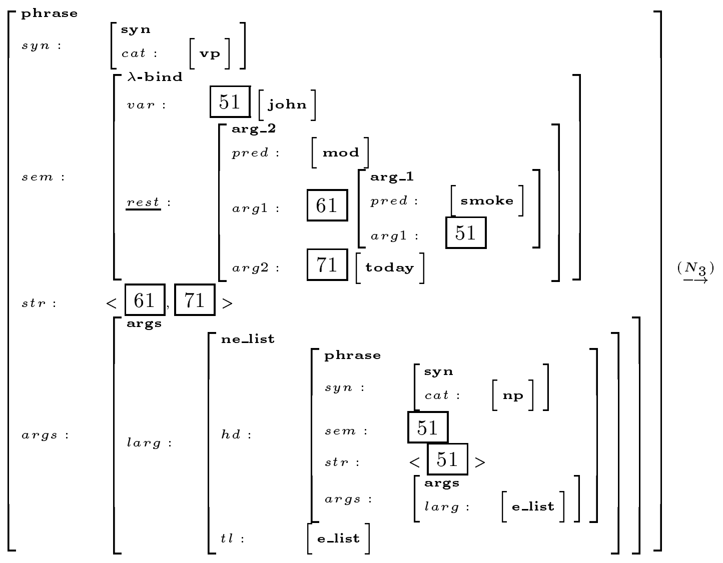
Rearrangement rules

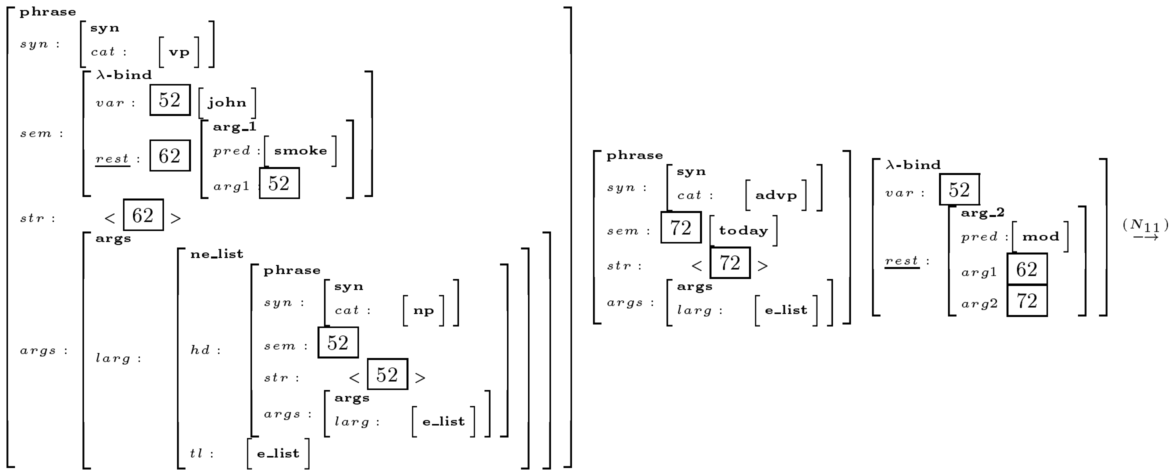
Rules $R_1 - R_3$ added by the normalization algorithm.

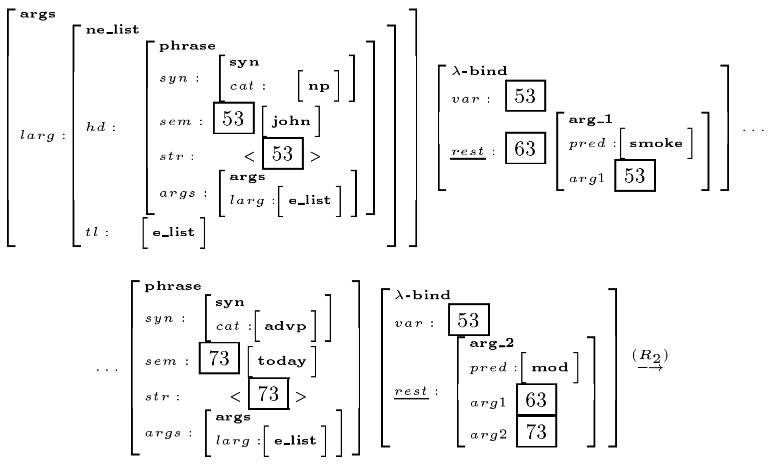
3.4.9 Sample derivation with the normalized grammar

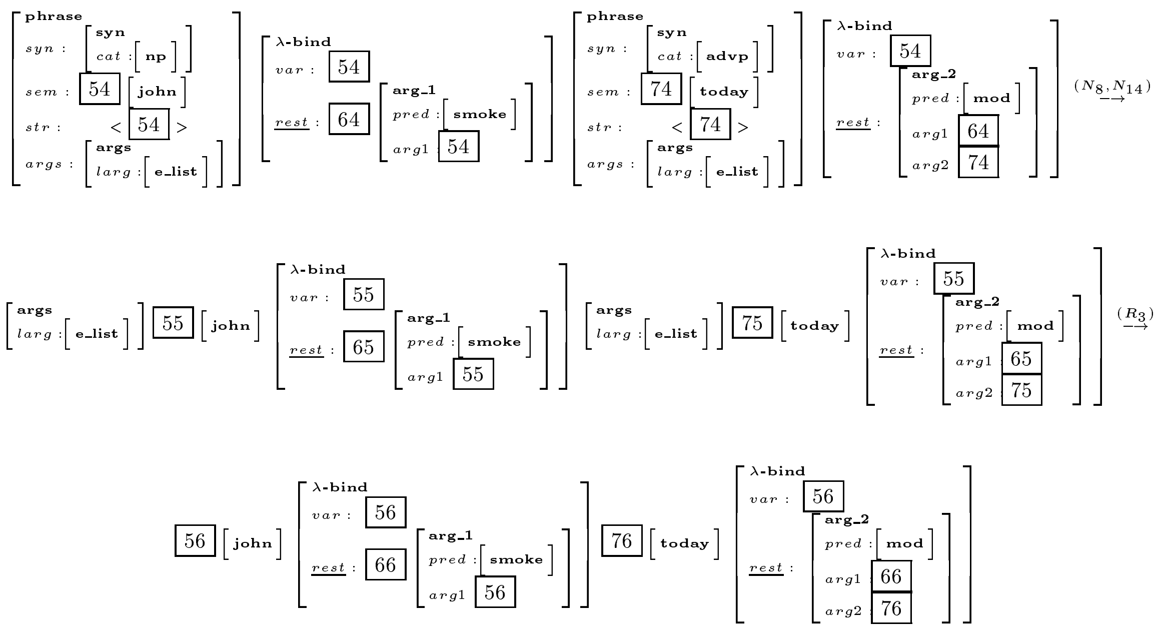
Here we show derivation of the sentence "John smokes today" with G^e_N.

$$\begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix}\\ sem: & \begin{bmatrix}\textbf{arg\_2} & \\ pred: & [\textbf{mod}]\\ arg1: & \boxed{6}\begin{bmatrix}\textbf{arg\_1} & \\ pred: & [\textbf{smoke}]\\ arg1: & \boxed{5}\,[\textbf{john}]\end{bmatrix}\\ arg2: & \boxed{7}\,[\textbf{today}]\end{bmatrix}\\ str: & <\boxed{5},\boxed{6},\boxed{7}>\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix} \xrightarrow{(N_2)}$$


$\xrightarrow{(N_3)}$


$\xrightarrow{(N_{11})}$


$\xrightarrow{(R_2)}$



As mentioned above, derivation with G_N terminates with a sequence of meaning primitives, which together comprise the input logical form.

3.5 Grammar inversion

3.5.1 The nature of inversion

As explained above, the inversion transforms a normalized grammar into a form suitable for generation.

In what follows we refer to the rules of the inverted grammar as *"inverted"* rules. The inverted grammar is obtained from the the normalized one: the inversion algorithm examines various chains of Argument-Filling rules terminated with Functor-Introducing rules. As we explained in Section 3.4, chains of AF rules collect arguments and "route" them to their respective functors provided by the FI rules. A Functor-Introducing rule at the end of a chain can be viewed as "consuming" the predicate and raising its arguments to the surface, to be processed recursively in the same manner. The inversion algorithm concatenates normalized rules into chains and creates a new (inverted) rule from the two ends of each such chain. Thus, applying an inverted rule is equivalent to dealing with the predicate and simultaneously producing all of its arguments, to be processed next.

The aim of inversion is to create rules that allow systematical *"parsing"* (consumption) of logical forms. Upon inversion, the grammar attains the nested predicate-argument structure similar to that of meaning logical forms. Now both the semantic form and grammar rules may be "traversed" in parallel.

As a result, whenever the head of an inverted rule represents a logical form (in the predicate-argument notation, i.e., a predicate applied to its arguments), the rule body has a constituent corresponding to each of the head arguments, as well as an additional constituent corresponding to the predicate itself. This allows straightforward decomposition of complex logical forms into semantic primitives, by consuming the semantic predicate and then recursively consuming its arguments.

For example, let us consider rule I_6 of the inverted grammar (see Section 3.5.3 below). The semantics of the rule head is formed by applying semantics of an intransitive verb to that of a noun phrase (thus, the former acts as a predicate and the latter as an argument). The first body constituent corresponds to the argument of the head semantics ($\boxed{5}$), while the second constituent represents the predicate ($\boxed{1}$).

Originally, the generation algorithm presented in (Samuelsson, 1995) was designed to improve the efficiency of the Semantic-Head-Driven Generation (devised by Shieber et al. (1990)). The SHDG algorithm operates by picking a non-chain rule and connecting it to an upper node in the derivation tree which has the same semantic form, while the connection is made through a series of chain rules. In this scenario many different rule combinations are attempted, vastly increasing the amount of spurious search. Samuelsson's algorithm remedies this situation by concurrently processing all chains leading to a particular logical form, before any rule is actually applied.

The inversion phase examines all the possible chains of AF rules, and attempts to terminate each such chain with a suitable FI rule; the arguments collected along the chain are then consumed by the trailing FI link. Combining AF rules into chains is performed in such a way that the body of one rule unifies with the head of another, while the body of the penultimate rule unifies with the head of the FI rule. Here we can clearly see the necessity of the normalization phase to restructure the chain rules into unit rules, thus allowing to combine them into chains. Moreover, normalization sets the stage for proper argument passing in chains.

When a chain has been maximally expanded (i.e., no other AF rule may further extend it) and properly terminated, it originates a series of inverted rules. First, the entire chain makes a new rule — the head of the first rule in the chain constitutes the head, and the body of the trailing FI rule constitutes the body of the new rule. Furthermore, sub-chains that begin at AF rules with a non-preterminal syntactic category in the head also induce inverted rules, in the same way as the complete chain (see Section 3.5.4 for the examples when such rules can be used).

To minimize the number of rules, every new rule is compared against the already existing ones, for possible subsumption. The algorithm performs subsumption check between the MRS of the new rule and MRSs of the rules created so far. If any existing MRS is more general than the new one, than it is replaced with the latter, and if it is more specific – the latter is discarded. Otherwise, the new rule is added to the inverted grammar.

If a FI rule cannot terminate any chain of AF rules, it makes an inverted rule on its own (as if preceded by a zero-length chain). In any case, after a chain is combined and terminated with a FI rule, the head of the first rule in the chain is modified to set $< H\ args\ larg >= [\mathbf{e\_list}]$. The reason is that all the possible arguments have already been collected along the chain (or there are no arguments, if the chain is of zero length), hence there is no need to expect any more.

Processing lexicon-derived rules

Let us consider what happens during inversion to lexicon-derived FI rules. According to Step 4 of the normalization algorithm, such rules are of the form $A\ [H] \Rightarrow H$, where $A =< H\ args >$ is a generic lists of arguments, and $[H] =< H\ sem >$ – the semantics constituent. When such a rule terminates a chain of AF rules, A is instantiated to the list of arguments collected along the chain and passed to the functor introduced by the lexicon-derived rule. The FS representing A is of type **args**, and its *larg* feature encodes a linked list of arguments using the standard list types (**ne_list** and **e_list**). The inversion algorithm then flattens such a list to a sequence of FSs, and thus creates the body of the new rule.

This step is equivalent to applying the rearrangement rules $R_1 - R_3$ of G_N, until the list is flattened entirely. In the normalized grammar, the particular values of the argument lists are unknown at the time the grammar is created, hence the rules $R_1 - R_3$ are required to process constructs of type **args**, when such appear during actual derivations. On the contrary, in the inverted grammar argument lists are fully instantiated, due to the chain combination which collects the arguments and passes them to the FI rule at the end of the chain. Therefore, the flattening of argument lists may be performed at the time of grammar inversion, i.e., at compilation time instead of at run-time. Consequently, rearrangement rules are not necessary in G_I (since their effect is already *built into* the rules).

After the list of arguments is flattened, the FSs of the resulting sequence are reordered to reflect the actual argument order of the head semantic core. This is performed in a similar manner as bodies of FI rules are reordered in Step 2 of the normalization algorithm). Again, this particular transformation could not have been performed during normalization, as the body of the lexicon-derived rules only gets instantiated during inversion, due to chain combination.

The inversion algorithm also includes a special provision for handling semantic primitives that can serve both predicates and arguments. This case is special, since a feature structure whose semantics serves a predicate necessarily has a non-empty argument list, while in a feature structure representing an argument this list may be empty. Since this case goes beyond the scope of our sample grammar, we state the necessary provision in Step 1(d)iii of the Inversion algorithm and give the detailed explanation later in Section 3.5.5.

Termination

Since the inversion algorithm inspects all the possible chains of AF rules, in order for the algorithm to terminate the input grammar (G_N) must not have purely AF cycles. Hence an analogous condition needs to be assumed for G_O, namely, it must not have purely chain-rule cycles. In other words, we require G_O to satisfy the off-line parsability criterion, as formulated in (Kaplan and Bresnan, 1982, pp.264-266).

To ensure termination of the inversion phase, rule chains are actually combined "from the end", i.e., starting from the trailing FI rule. This way the semantics of the FI rule prevents infinite chain expansion.

Derivation with inverted grammars

According to the description above, rules of inverted grammars reflect the predicate-argument structure of their head's logical form. Thus, the body of an inverted rule has one constituent corresponding to each argument of the head's semantics, as well as another constituent corresponding to the semantics predicate of the head logical form. Moreover, the order of the body elements mirrors a *postorder* traversal of the head semantics, i.e., the last body constituent corresponds to the predicate, while the rest of the constituents appear in the order of their respective arguments.

A G_I derivation terminates (if at all) when no more inverted rules can be applied. If the derivation terminates successfully, then all the FSs in the resultant sentential form are of types which are subtypes of **sem**. Such feature structures result from the *semantics constituents*, which are the last constituents of FI rules created during normalization. According to the normalization algorithm (Steps 2 and 4) those are actually semantic primitives that comprise logical forms. Therefore, when inverted rules are applied one after another during derivations, such postorder sequences of semantic primitives accumulate, while each sequence corresponds to some part of the input logical form.

Thus, when a derivation with G_I terminates successfully (if at all), it ends up with a sequence of meaning primitives, which correspond to the *postorder* decomposition (flattening) of the entire input logical form. This fact is crucial in the proposed generation algorithm (see Chapter 4). Given a logical form with the desired meaning, the generation algorithm flattens it into a postorder sequence of arguments and predicates. This sequence initializes the chart and the *bottom-up* generation starts. Since the bodies of inverted rules also reflect the predicate-argument structure of the head semantic form, the *dot movement* operation matches them against the contents of the chart in a straightforward manner. During generation, application of the inverted rules proceeds until the entire semantics input (all the chart initialization entries) has been consumed.

In the light of the above description, we can now explain the remark in Section 3.4.3 about the selection of *argument carriers* in Functor-Introducing rules. If arguments are not eventually passed to their respective functors, the sequence of semantic primitives which results from a derivation with G_I will not be in a correct *postorder* format (arguments followed by their predicates). In fact, this sequence will lack a systematical structure our generation algorithm relies on, therefore obstructing systematical "consumption" of input semantic forms. In some cases, a derivation with G_I might even get stuck because of incorrect argument transfer. Consider for example, that in rule O_3 the second body constituent were designated as the argument carrier, instead of the first one. Then the normalized version of this rule (N_3) would be of the form $VP\ AdvP(A)\ [VP] \Rightarrow VP(A)$, and the inverted rule I_2 due to concatenation of N_2 and N_3 would be $VP\ AdvP(NP)\ [VP] \Rightarrow S$. Since there are no Argument-Filling rules where $AdvP$ acts as a predicate, the only inverted rule handling $AdvP$ is I_{14}, which does not expect any arguments. In the example discussed, the feature structure representing $AdvP(NP)$ has a non-empty argument list (namely, there is an NP argument), therefore it cannot unify with the head of I_{14}. Consequently, the derivation can proceed no further. Thus it is the designated argument carriers of grammar rules that are responsible for correct routing of arguments. We require argument carriers to be specified manually, since it is not immediately clear how to perform this task automatically in the general case.

Auxiliary definitions

Prior to formally stating the inversion algorithm we define the *rule chain* and the operation of *rule combination*. We use "$\downarrow$" to denote that a partial operation is *defined* on its arguments and "$\uparrow$" otherwise.

Rule combination and rule chains

1. If $R = (B_1 \dots B_l \Rightarrow C) \in G_{FI}$, then R is a **rule chain**.
2. If $R = (B_1 \dots B_l \Rightarrow D_m \Rightarrow \dots \Rightarrow D_2 \Rightarrow D_1 \Rightarrow C)$ is a **rule chain**, $Q = (D_0 \Rightarrow H) \in G_{AF}$ and **unifiable**(C, D_0), then $R' = (B'_1 \dots B'_l \Rightarrow D'_m \Rightarrow \dots \Rightarrow D'_2 \Rightarrow D'_1 \Rightarrow D'_0 \Rightarrow H')$ is a **rule chain**, where $D'_0 = C \sqcup D_0$ and $B'_1, \dots, B'_l, D'_m, \dots, D'_2, D'_1, H'$ are $B_1, \dots, B_l, D_m, \dots, D_2, D_1, H$ respectively as changed upon the unification of C and D_0.

   The **combination** of R and Q into R' is denoted by $R' = R \circ Q$. If $\neg$**unifiable**(C, D_0) then $(R \circ Q) \uparrow$.
3. There are no other rule chains.
4. The constituents $B_1 \dots B_l$ in items (1) and (2) above are collectively referred to as the **body** of a rule chain.

It should be emphasized that a rule chain is *not* a grammar rule but rather an intermediate conversion result, therefore it may contain multiple production symbols ("$\Rightarrow$"). Observe that the body of the last rule in the chain (the FI rule) may have more than one constituent.

Let us consider an example of rule combination. We start "growing" a chain with Functor-Introducing rule $N_{11} = A\ [V_i] \Rightarrow V_i(A)$. According to item (1) of the above definition, this rule constitutes a rule chain by itself. We then *combine* this rule chain with Argument-Filling rule $N_2 = VP(NP) \Rightarrow S$. The combination is possible since the head $V_i(A)$ of N_{11} unifies with the body $VP(NP)$ of N_2. The unification is performed *in the context* of the two rules, and both rules become more specific. Namely, the generic argument list A is instantiated to NP, and VP becomes V_i since the latter is more specific. Thus, $N_{11} \circ N_2 = NP\ [V_i] \Rightarrow V_i(NP) \Rightarrow S$. According to item (4) of the above definition, the constituents NP and $[V_i]$ together comprise the body of the chain $N_{11} \circ N_2$. Since there is no Argument-Filling rule whose body unifies with S, the example chain cannot be extended any more.

Preterminal and argument categories For the inversion algorithm to operate correctly, we distinguish two special kinds of syntactic categories:

Preterminal categories A syntactic category c is a **preterminal category** if

1. there exists a lexicon entry ("word" $\rightarrow H) \in \mathit{Lexicon}(G_O)$, s.t. $< H\ syn\ cat >= c$, **and**
2. there **does not** exist a grammar rule $(B_1 \dots B_m \Rightarrow H) \in G_O$ s.t. $< H\ syn\ cat >= c$.

In what follows, **ptc**(c) denotes the fact that c is a preterminal category. The set of all the preterminal categories for the grammar is denoted by $PTC(G_O)$.

- The path $< H\ syn\ cat >$ used in the above definition necessarily exists according to the assumptions on the minimal required type hierarchy of the input grammars, see Section 3.2.3.
- Note that $PTC(G_O)$ is easily computable from $\mathit{Lexicon}(G_O)$ in $O(|\mathit{Lexicon}(G)|)$ time. In our implementation, $PTC(G_O)$ is computed during lexicon normalization. In the grammar of the running example, $PTC(G^e_O) = \{NP, V_i, AdvP\}$, while $S, VP \notin PTC(G^e_O)$.

Argument categories If $B_0(B_1 \ldots B_m) \Rightarrow H$ is an AF rule produced by the Normalization algorithm, then $< B_1$ *syn cat* $>, \ldots, < B_m$ *syn cat* $>$ are **argument categories**. The set of all the argument categories for the grammar is denoted by $ARGC(G_N)$.

3.5.2 Inversion algorithm

1. CHAIN COMBINATION.

   Calculate the set $RC(G_N)$ of **maximal rule chains** over G_N:

   $RC(G_N) := \emptyset$

   For every $R = (B_1 \ldots B_l\ [H] \Rightarrow H)$ in G_{FI} :

   (a) Let $R' = R$;

   (b) **While** there is $Q = (D \Rightarrow C)$ in G_{AF}
   such that $(R' \circ Q) \downarrow$ **and** $\neg(\mathbf{ptc}(< D\ \mathit{syn\ cat} >) \wedge \neg\mathbf{ptc}(< H'\ \mathit{syn\ cat} >))$
   do $R' = R' \circ Q$;
   Comment: A rule (in the beginning of the chain R') with a non-preterminal category in the head (H') cannot be concatenated with an AF rule Q whose body has a preterminal category. The latter can only expand some lexicon-derived rule whose head has a preterminal category.

   (c) Set $< H'\ \mathit{args\ larg} >= [\mathbf{e\_list}]$;

   (d) If R is a *lexicon-derived* rule:

      i. Flatten the body of R' (of type **args**), which encodes a list of arguments collected along the chain, to obtain a shallow sequence of body constituents: $B'_1 \ldots B'_m$.

      ii. Restructure the body of $R' = (B'_1 \ldots B'_m\ [H'] \Rightarrow H')$ to reflect the actual argument order of the H' semantic core:
      Let $SC =< H'\ \mathit{sem}\ \underline{\mathit{rest}}^* >$ be the semantic core of H'.
      Then $R' = (B'_{i_1} \ldots B'_{i_m}\ [H] \Rightarrow H')$,
      where $\forall k, 1 \leq k \leq m : \mathbf{reentrant}(< B_{i_k}\ \mathit{sem}\ \underline{\mathit{rest}}^* >, < SC\ \mathit{arg}_k >)$.

      iii. If any AF rules have been combined with R in Step 1b and $< H\ \mathit{syn\ cat} >\in ARGC(G_N)$, create an additional chain which consists of a single rule $R'' = R$ and set $< H''\ \mathit{args\ larg} >=$ $[\mathbf{e\_list}]$. Add R'' to $RC(G_N)$: $RC(G_N) = RC(G_N) \cup \{R''\}$.
      Comment: This is to allow for the possibility of semantic primitives that can serve both predicates and arguments. This rule has no arguments and hence its body does not need restructuring.

   (e) $RC(G_N) = RC(G_N) \cup \{R'\}$.

   Comment: Step 1 of the algorithm terminates, since by assumption (Section 3.5.1) there are no purely AF cycles.

2. CREATION OF THE INVERTED RULES PROPER.

   Decompose the combined chains into rules:

   (a) $G_I := \emptyset$

   (b) For each $R' = (B'_{j_1} \ldots B'_{j_l} \Rightarrow B'_{j-1} \Rightarrow \ldots \Rightarrow B'_1 \Rightarrow H')$ in $RC(G_N)$:

      i. If $j = 1$ then $\mathbf{add\_rule}(G_I, R')$.
      Comment: a chain of length 1.

      ii. else
      $r_I := (B'_{j_1} \ldots B'_{j_l} \Rightarrow H')$; $\mathbf{add\_rule}(G_I, r_I)$;
      For $n := 1$ to $(j-1)$

Case	Combined AF rules	Terminating FI rule	Chain backbone
1	N_2	N_3	$VP(NP)\ AdvP\ [VP] \Rightarrow VP(NP) \Rightarrow S$
2	N_2	N_{11}	$NP\ [V_i] \Rightarrow V_i(NP) \Rightarrow S$
3	—	N_8	$[NP] \Rightarrow NP$
4	—	N_{14}	$[NP] \Rightarrow AdvP$

Table 3.1: Terminated chains of combined G_N^e rules.

```
    A. if $< B_n\ syn\ cat > \notin PTC(G_O)$
       (where $B_n$ is $B'_n$ before the unifications of Step 1)
       then $r_I := (B'_{j_1} \ldots B'_{j_l} \Rightarrow B'_n)$; **add_rule**$(G_I, r_I)$;
    B. else **break** from the **for** loop
       /* **continue** with another $R' \in RC(G_N)$ (if any) */
```

3. The lexicon of G_O makes the lexicon of G_I.

NOTE: As follows from the Inversion algorithm, the rearrangement rules $R_1, R_2, R_3 \in G_N$ are disposed of and not added to G_I (since their effect is "built into" the rules of G_I in Step 1 of the algorithm).

3.5.3 Inverted sample grammar

Let us now demonstrate the grammar G_I^e which results from applying the inversion algorithm to G_N^e. To better illustrate how the inversion algorithm works, we show in Table 3.1 the intermediate results received in processing G_N^e after Step 1. Namely, we list the elements of $RC(G_N^e)$ and for each element specify which G_N^e rules have licensed its creation. For brevity sake we only display the CF backbones of combined chains.

In what follows we list the rules of G_I^e. Whenever appropriate, comments are provided to explain how the inverted rules result from the normalized ones; case numbers in the comments refer to the lines of Table 3.1.

Initial symbol

Exactly the same as in G_N^e.

Inverted grammar rules

The numbering of the inverted rules below corresponds to that in (Samuelsson, 1995).

I_2 Case 1. The two ends of the chain are taken for a new rule.

$$\left[\begin{array}{ll}
\textbf{phrase} & \\
syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{array}\right] \\
sem: & \left[\begin{array}{ll}\lambda\textbf{-bind} & \\ var: & \boxed{5}\,[\textbf{sem}] \\ \underline{rest}: & \boxed{6}\,[\textbf{funct}]\end{array}\right] \\
str: & \boxed{4}\,[\textbf{list}] \\
args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & \left[\begin{array}{ll}\textbf{ne\_list} & \\ hd: & \left[\begin{array}{ll}\textbf{phrase} & \\ syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{np}]\end{array}\right] \\ sem: & \boxed{5} \\ str: & \boxed{3}\,[\textbf{list}] \\ args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{array}\right]\end{array}\right] \\ tl: & [\textbf{e\_list}]\end{array}\right]\end{array}\right]
\end{array}\right]
\left[\begin{array}{ll}
\textbf{phrase} & \\
syn: & \left[\begin{array}{ll}\textbf{syn} & \\ cat: & [\textbf{advp}]\end{array}\right] \\
sem: & \boxed{7}\,[\textbf{sem}] \\
str: & \boxed{9}\,[\textbf{list}] \\
args: & \left[\begin{array}{ll}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{array}\right]
\end{array}\right]
\left[\begin{array}{ll}
\lambda\textbf{-bind} & \\
var: & \boxed{5} \\
\underline{rest}: & \boxed{17}
\end{array}\right] \Longrightarrow$$

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix} \\ sem: & \boxed{17}\begin{bmatrix}\textbf{arg\_2} & \\ pred: & [\textbf{mod}] \\ arg1: & \boxed{6} \\ arg2: & \boxed{7}\end{bmatrix} \\ str: & << \boxed{3} >, < \boxed{4} >, < \boxed{9} >> \\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix}$$

I_4 Case 1. Since the intermediate VP is not a pre-terminal, the proper sub-chain $VP(NP)\ AdvP\ [VP] \Rightarrow VP(NP)$ also forms an inverted rule.

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{bmatrix} \\ sem: & \begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ var: & \boxed{5}[\textbf{sem}] \\ \underline{rest}: & \boxed{6}[\textbf{funct}]\end{bmatrix} \\ str: & \boxed{4}[\textbf{list}] \\ args: & \boxed{8}\begin{bmatrix}\textbf{args} & \\ larg: & \begin{bmatrix}\textbf{ne\_list} & \\ hd: & \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{np}]\end{bmatrix} \\ sem: & \boxed{5} \\ str: & \boxed{3}[\textbf{list}] \\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix} \\ tl: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix}\end{bmatrix} \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{advp}]\end{bmatrix} \\ sem: & \boxed{7}[\textbf{sem}] \\ str: & \boxed{9}[\textbf{list}] \\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix} \boxed{20} \Longrightarrow$$

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{vp}]\end{bmatrix} \\ sem: & \boxed{20}\begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ var: & \boxed{5} \\ \underline{rest}: & \boxed{17}\begin{bmatrix}\textbf{arg\_2} & \\ pred: & [\textbf{mod}] \\ arg1: & \boxed{6} \\ arg2: & \boxed{7}\end{bmatrix}\end{bmatrix} \\ str: & << \boxed{4} >, < \boxed{9} >> \\ args: & \boxed{8}\end{bmatrix}$$

I_6 Case 2.

$$\begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{np}]\end{bmatrix} \\ sem: & \boxed{5}[\textbf{sem}] \\ str: & \boxed{3}[\textbf{list}] \\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix} \begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ var: & \boxed{5}[\textbf{sem}] \\ \underline{rest}: & \boxed{1}\begin{bmatrix}\textbf{arg\_1} & \\ pred: & [\textbf{v\_intrans}] \\ arg1 & \boxed{5}\end{bmatrix}\end{bmatrix} \Longrightarrow \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & [\textbf{s}]\end{bmatrix} \\ sem: & \boxed{1} \\ str: & << \boxed{3} >, \boxed{1} > \\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix}$$

I_7 Case 2. Since VP is not a pre-terminal, the proper subchain $NP\ [V_i] \Rightarrow V_i(NP)$ also makes an inverted rule.

$$\boxed{9}\left[\begin{array}{ll}\textbf{phrase} & \\ \mathit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \mathit{cat}: & \left[\textbf{np}\right]\end{array}\right] \\ \mathit{sem}: & \boxed{5}\left[\textbf{sem}\right] \\ \mathit{str}: & \boxed{3}\left[\textbf{list}\right] \\ \mathit{args}: & \left[\begin{array}{ll}\textbf{args} & \\ \mathit{larg}: & \left[\textbf{e\_list}\right]\end{array}\right]\end{array}\right] \boxed{17}\left[\begin{array}{ll}\boldsymbol{\lambda}\textbf{-bind} & \\ \mathit{var}: & \boxed{5} \\ \underline{\mathit{rest}}: & \boxed{1}\left[\begin{array}{ll}\textbf{arg\_1} & \\ \mathit{pred}: & \left[\textbf{v\_intrans}\right] \\ \mathit{arg1}: & \boxed{5}\end{array}\right]\end{array}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \mathit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \mathit{cat}: & \left[\textbf{vi}\right]\end{array}\right] \\ \mathit{sem}: & \boxed{17} \\ \mathit{str}: & <\boxed{1}> \\ \mathit{args}: & \left[\begin{array}{ll}\textbf{args} & \\ \mathit{larg}: & \left[\begin{array}{ll}\textbf{ne\_list} & \\ \mathit{hd}: & \boxed{9} \\ \mathit{tl}: & \left[\textbf{e\_list}\right]\end{array}\right]\end{array}\right]\end{array}\right]$$

I_{11} Case 3. Obtained from the FI rule N_8, as it cannot be combined with any AF rule.

$$\boxed{1}\left[\textbf{pn}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \mathit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \mathit{cat}: & \left[\textbf{np}\right]\end{array}\right] \\ \mathit{sem}: & \boxed{1} \\ \mathit{str}: & <\boxed{1}> \\ \mathit{args}: & \left[\begin{array}{ll}\textbf{args} & \\ \mathit{larg}: & \left[\textbf{e\_list}\right]\end{array}\right]\end{array}\right]$$

I_{14} Case 4. Obtained from the FI rule N_{14}, as it cannot be combined with any AF rule.

$$\boxed{1}\left[\textbf{adv}\right] \Longrightarrow \left[\begin{array}{ll}\textbf{phrase} & \\ \mathit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \mathit{cat}: & \left[\textbf{advp}\right]\end{array}\right] \\ \mathit{sem}: & \boxed{1} \\ \mathit{str}: & <\boxed{1}> \\ \mathit{args}: & \left[\begin{array}{ll}\textbf{args} & \\ \mathit{larg}: & \left[\textbf{e\_list}\right]\end{array}\right]\end{array}\right]$$

Lexicon

The lexicon (rules O_8, O_{11} and O_{14}) of G^e_O.

3.5.4 Sample derivation with the inverted grammar

Let us demonstrate a derivation of the sample sentence "John smokes today" with G^e_I.

$$\left[\begin{array}{ll}\textbf{phrase} & \\ \mathit{syn}: & \left[\begin{array}{ll}\textbf{syn} & \\ \mathit{cat}: & \left[\textbf{s}\right]\end{array}\right] \\ \mathit{sem}: & \left[\begin{array}{ll}\textbf{arg\_2} & \\ \mathit{pred}: & \left[\textbf{mod}\right] \\ \mathit{arg1}: & \boxed{6}\left[\begin{array}{ll}\textbf{arg\_1} & \\ \mathit{pred}: & \left[\textbf{smoke}\right] \\ \mathit{arg1}: & \boxed{5}\left[\textbf{john}\right]\end{array}\right] \\ \mathit{arg2}: & \boxed{7}\left[\textbf{today}\right]\end{array}\right] \\ \mathit{str}: & <\boxed{5},\boxed{6},\boxed{7}> \\ \mathit{args}: & \left[\begin{array}{ll}\textbf{args} & \\ \mathit{larg}: & \left[\textbf{e\_list}\right]\end{array}\right]\end{array}\right] \xrightarrow{(I_2)}$$

$$\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{vp}]\end{array}\right]\\ \mathit{sem}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{52}[\textbf{john}]\\ \underline{\mathit{rest}}: & \boxed{62}\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_1}}\\ \mathit{pred}: & [\textbf{smoke}]\\ \mathit{arg1}: & \boxed{52}\end{array}\right]\end{array}\right]\\ \mathit{str}: & <\boxed{62}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{ne\_list}}\\ \mathit{hd}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{np}]\end{array}\right]\\ \mathit{sem}: & \boxed{52}\\ \mathit{str}: & <\boxed{52}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right]\\ \mathit{tl}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{advp}]\end{array}\right]\\ \mathit{sem}: & \boxed{72}[\textbf{today}]\\ \mathit{str}: & <\boxed{72}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{52}\\ \underline{\mathit{rest}}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_2}}\\ \mathit{pred}: & [\textbf{mod}]\\ \mathit{arg1}: & \boxed{62}\\ \mathit{arg2}: & \boxed{72}\end{array}\right]\end{array}\right] \xrightarrow{(I_7)}$$

$$\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{np}]\end{array}\right]\\ \mathit{sem}: & \boxed{54}[\textbf{john}]\\ \mathit{str}: & <\boxed{54}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{54}\\ \underline{\mathit{rest}}: & \boxed{64}\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_1}}\\ \mathit{pred}: & [\textbf{smoke}]\\ \mathit{arg1}: & \boxed{54}\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{advp}]\end{array}\right]\\ \mathit{sem}: & \boxed{74}[\textbf{today}]\\ \mathit{str}: & <\boxed{74}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right] \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{54}\\ \underline{\mathit{rest}}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_2}}\\ \mathit{pred}: & [\textbf{mod}]\\ \mathit{arg1}: & \boxed{64}\\ \mathit{arg2}: & \boxed{74}\end{array}\right]\end{array}\right] \xrightarrow{(I_{11}, I_{14})}$$

$$\boxed{56}[\textbf{john}] \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{56}\\ \underline{\mathit{rest}}: & \boxed{66}\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_1}}\\ \mathit{pred}: & [\textbf{smoke}]\\ \mathit{arg1}: & \boxed{56}\end{array}\right]\end{array}\right] \boxed{76}[\textbf{today}] \left[\begin{array}{ll}\multicolumn{2}{l}{\lambda\textbf{-bind}}\\ \mathit{var}: & \boxed{56}\\ \underline{\mathit{rest}}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_2}}\\ \mathit{pred}: & [\textbf{mod}]\\ \mathit{arg1}: & \boxed{66}\\ \mathit{arg2}: & \boxed{76}\end{array}\right]\end{array}\right]$$

As mentioned above, derivation with G_I terminates with a sequence of meaning primitives, which correspond to the *postorder* decomposition (flattening) of the input logical form.

Explanation of the sample derivation

The nature of the above derivation may appear counterintuitive at the first glance. The familiar syntactic analysis (cf. Section 3.3.3) calls to first apply the rule $NP\ VP \Rightarrow S$ (O_2), and then to decompose the VP using the rule $VP\ AdvP \Rightarrow VP$ (O_3). This order of rule application reflects that the adverb modifies the verb, not the entire sentence, and the noun phrase joins the verb already modified.

As opposed to this, the first inverted rule to be applied is $I_2 = VPAdvP[\mathbf{mod}] \Rightarrow S$, which detaches the adverb from the verb. Only then is rule $I_7 = NP[V_i] \Rightarrow V_i$ applied to decompose the verb phrase into a noun phrase and a verb. Observe that the order in which the inverted rules are applied reflects the predicate-argument structure of the given logical form. The semantics supplied with the initial symbol is `mod(sleep(john, today))`, hence rule I_2 singles out the arguments `sleep(john)` and `today` of the predicate `mod`. Then rule I_7 decomposes the semantics of the verb phrase into the argument (`john`) and the predicate (`sleep`), organized in the postorder arrangement.

Additional examples

In conclusion we briefly discuss the cases when the rest of the inverted rules (not used in the example above) may be employed.

1. Rule I_4 can be used if a sentence contains two adverbs. For example, suppose the sample grammar contained a lexical entry for the word "slowly" similar to entry O_{14} for the word "today". Then the sentence "John smokes slowly today" having the meaning `mod( mod(smoke(john), slowly), today )` could be derived as follows. First, rule I_2 would be applied to the input feature structure (similarly to the example in Section 3.5.4). This would detach the outer `mod` predicate from its two arguments — VP with the semantics `mod(smoke(john), slowly)` and $AdvP$ with the semantics `today`. Then rule I_4 would develop the arguments of the former (i.e., the *inner* `mod` predicate of the given logical form). The application of rule I_4 would create feature structures for the VP and the $AdvP$ (having the semantics `smoke(john)` and `slowly`, respectively), followed by a FS for the `mod` predicate. Finally, rule I_7 would decompose the remaining VP exactly as in the example above.

2. Rule I_6 can be used to derive the simple sentence "John smokes":

$$\begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix}\textbf{s}\end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \boxed{6}\begin{bmatrix} \textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \boxed{5}\begin{bmatrix}\textbf{john}\end{bmatrix} \end{bmatrix} \\ \textit{str}: & <\boxed{5},\boxed{6}> \\ \textit{args}: & \begin{bmatrix} \textbf{args} & \\ \textit{larg}: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix} \end{bmatrix} \end{bmatrix} \xrightarrow{(I_6)}$$

$$\begin{bmatrix} \textbf{phrase} & \\ \textit{syn}: & \begin{bmatrix} \textbf{syn} & \\ \textit{cat}: & \begin{bmatrix}\textbf{np}\end{bmatrix} \end{bmatrix} \\ \textit{sem}: & \boxed{51}\begin{bmatrix}\textbf{john}\end{bmatrix} \\ \textit{str}: & <\boxed{51}> \\ \textit{args}: & \begin{bmatrix} \textbf{args} & \\ \textit{larg}: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix} \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} \lambda\textbf{-bind} & \\ \textit{var}: & \boxed{51} \\ \underline{\textit{rest}}: & \boxed{61}\begin{bmatrix} \textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \boxed{51} \end{bmatrix} \end{bmatrix} \end{bmatrix} \xrightarrow{(I_{11})}$$

$$\boxed{52}\begin{bmatrix}\textbf{john}\end{bmatrix} \begin{bmatrix} \lambda\textbf{-bind} & \\ \textit{var}: & \boxed{52} \\ \underline{\textit{rest}}: & \boxed{62}\begin{bmatrix} \textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \boxed{52} \end{bmatrix} \end{bmatrix}$$

3.5.5 Advanced issues

Semantic primitives that can serve both predicates and arguments

Step 1(d)iii of the Inversion algorithm constitutes a special provision for handling semantic primitives that can serve both predicates and arguments. In what follows we explain why this case is special and how it should be addressed during grammar inversion.

We use here a small subset of the sample grammar of Appendix B. For the sake of briefness, only the CF backbone of the grammar is shown. Let G_O^1 be as follows:

- **Grammar rules**

  O_1^1 $NP\ V_i \Rightarrow S$

  O_2^1 $V_t\ NP \Rightarrow V_i$

- **Lexicon**

  O_3^1 john $\rightarrow NP$

O_4^1 mary $\rightarrow NP$

O_5^1 loves $\rightarrow V_t$

The normalized grammar G_N^1 looks as follows. If both O_1^1 and O_2^1 are chain rules, then during normalization they are transformed into corresponding AF rules:

N_1^1 $NP(V_i) \Rightarrow S$

N_2^1 $V_t(NP) \Rightarrow V_i$

Lexicon entries O_3^1 – O_5^1 induce the following FI rules:

N_3^1 $A\ [NP] \Rightarrow NP(A)$

N_5^1 $A\ [V_t] \Rightarrow V_t(A)$

According to the definition in Section 3.5.1, $ARGC(G_N^1) = \{V_i, NP\}$.

Without Step 1(d)iii, the Inversion algorithm creates G_I^1 as follows:

I_1^1 $V_i\ [NP] \Rightarrow S$

I_2^1 $NP\ [V_t] \Rightarrow V_i$

Now let us try to derive the sentence "John loves Mary" using G_I^1. First, rule I_1^1 is applied $S \xrightarrow{(I_1^1)} V_i\ [NP]$. Then rule I_2^1 produces $V_i\ [NP] \xrightarrow{(I_2^1)} NP\ [V_t]\ [NP]$. Here we run into a problem since there is no inverted rule to handle the first constituent (NP) of the resultant sentential form.

Let us analyze the source of this problem. In rule N_1^1, NP is a predicate whose argument (V_i) appears on its argument list. When this rule is combined with rule N_3^1 during inversion, the generic argument list A is instantiated to V_i. That is, rule I_1^1 is created on the assumption that the argument list of NP is not empty, since such is the only rule combined with N_3^1. Specifically, there is no inverted rule which accomodates an NP with an empty argument list, while this is exactly the case we exemplified above.

This case can easily be predicted by examining the normalized grammar offline, prior to its inversion. Indeed, rule N_2^1 contains NP in an argument role which does not have any arguments of its own (that is why $NP \in ARGC(G_N^1)$). Therefore, an additional inverted rule is necessary to facilitate this case, namely

$$I_3^1 = [NP] \Rightarrow NP$$

(that is, rule N_3^1 in which A is instantiated to an empty list and consequently removed from the body). The necessary precondition to create such a rule is that the respective lexicon-derived rule (N_3^1) has an *argument* syntactic category in the head and is combined with other AF rules (i.e., rules in which NP serves as a predicate and therefore has a non-empty argument list). Formally speaking, an additional inverted rule is created for a lexicon-derived FI rule $r = A\ [H] \Rightarrow H(A)$, if

1. r terminates some chain of AF rules, **and**

2. $< H\ syn\ cat > \in ARGC(G_N)$.

This additional rule is created in exactly the same way as for the lexicon-derived rules which do not terminate any chain — the new rule has no arguments and hence a single (semantic) body constituent (cf. rule I_3^1 above).

Chapter 4

Chart-Based Generation with Typed Feature Structures

This chapter describes a chart-based generation algorithm for unification grammars. Section 4.1 gives an overview of the generation process. The generation algorithm proper is presented in Section 4.2. Section 4.3 shows an example of chart generation with the sample grammar. Finally, Section 4.4 explains how the Abstract Machine for Parsing developed by Wintner (1997) is enhanced to also perform generation.

4.1 Chart generation

Our generation algorithm relies on the assumption that its input grammars are given in the appropriate format. Namely, an input grammar should satisfy the following requirements:

1. The semantics associated with grammar rules is given in predicate-argument structure.
2. The body of each rule mirrors[1] the predicate-argument structure of the head:
    - exactly one body constituent corresponds to each argument of the head predicate,
    - the mutual order of body constituents duplicates the argument order in the head semantics,
    - an additional body constituent (situated in the last position) corresponds to the predicate of the head logical form.
3. The *str* feature in the constituents of grammar rules traces the actual phrase structure of the language defined by the grammar.
4. The lexicon supplied with the grammar allows to map semantic primitives (elements of logical forms) into natural language words.

One possible way to obtain a suitable grammar is to invert a "parsing" grammar using the inversion algorithm of Chapter 3. As we showed in Section 3.5, if a derivation with G_I starting from a feature structure F with semantics f terminates successfully, it ends up with a sequence of meaning primitives, which correspond to the *postorder* traversal of the logical form f. Given such a sequence of semantic elements, the generation algorithm reconstructs the analysis tree under a bottom-up control strategy.

[1]This requirement effectively states that the rule body reflects postorder flattening of the head logical form (to the limited depth of a single level).

4.1.1 Chart generation vs. chart parsing

We delineate the interrelation between parsing and generation by the following scheme. Given an phrase $w \in L(G_O)$, parsing it w.r.t. to G_O yields a feature structure F such that $< F\ sem >= f$, where $f \in LF$ and LF is some description language for logical forms, which conforms to the assumptions of Section 3.2. On the other hand, generating from the input f (i.e., from a FS α such that $< \alpha\ sem >= f$) w.r.t. to G_I (obtained by inverting G_O), yields a feature structure F' such that $< F'\ str >$ encodes a phrase w with meaning f.

The generation algorithm presented here is a *chart-based* one, since it employs a table (*chart*) for storing intermediate analysis results, similarly to chart parsing algortihms. Our algorithm was designed to use the existing Abstract Machine engine for chart processing of unification grammars (Wintner, 1997), with the aim of creating a bidirectional NLP system. This way the generation extension of $\mathcal{A}$MALIA uses the existing implementation of the chart processing routine, with appropriate modifications to actually perform generation instead of parsing.

The nature of chart items is quite different for parsing and generation. As opposed to parsing chart items, which span a sub-sequence of the input string, generation items span a part (sub-form) of the input logical form. In fact, this defines a way to decompose the input in either case into series of components. We also observe that in both cases the analysis of the input (parsing or generation, respectively) is performed by combining the components using the rules of the appropriate grammar (regular grammar in the case of parsing, or *inverted* grammar in the case of generation). Thus, usual chart operations (namely, *dot movement* and *completion*) may be used to apply grammar rules in both cases, regardless of the actual processing direction.

The order of rule application during bottom-up generation is the reverse of that of a *complementary* derivation with the inverted grammar. By *complementary* we mean here a derivation which starts from the same feature structure that constitutes the input for the generation. Section 4.3 (in particular, Subsection 4.3.2) demonstrates a sample bottom-up chart generation, which is the reciprocal of the sample derivation shown in Section 3.5.4.

In the case of generation the input is some meaning encoded as a nested logical form, and a NL grammar. This form is first flattened to produce a linear sequence of feature structures, each containing some part of the input semantics. Next these parts are mapped into NL constructs (words or phrases, also represented with FSs) with the corresponding meaning. During generation these FSs are combined using the rules of the inverted grammar, with semantics guiding the selection of rules to be applied. The *str* feature in the constituents of the inverted rules (see Section 3.2.2) encodes parts of the phrase being generated, by tracing the word order of the original NL grammar. Ultimately, if generation terminates successfully (i.e., with a non-empty set of FSs whose semantics subsumes the entire input), the *str* value of each resultant FS encodes a phrase with the desired meaning. This encoding uses feature structures to represent NL words in the correct order. Further (grammar-independent) processing is required to convert such lists of FSs into regular phrases (sequences of words); we refer to this step as *"verbalization"*.

Since grammars in our formalism do not have an initial symbol, the generator may yield not only "complete" phrases, but "partial" phrases as well. For example, when generating from the input form `mod(smoke(john), today)`, the phrase "smokes today" may be produced in addition to the expected "John smokes today". This happens when all the parts of the input logical form are consumed, but some of them remain on the argument list of the outer predicate. An example of this case is given in Section 4.3.4 below (cf. Section A.3.2). In order to filter out such phrases, it is possible to prohibit non-empty argument lists in the generation results, and during verbalization consider only feature structures whose $< \ldots\ args\ larg >=$ [**e_list**].

4.1.2 Chart operations

Let $L = l_0 \ldots l_{n-1}$ be the chart initialization sequence (input to the chart processing routine). In the general case, chart items are of the form $\gamma = [j, B_1 \ldots B_m, k, status]$, where $B_1 \ldots B_m$ are feature structures spanning a part of L between the positions j and k — $l_j \ldots l_k$, and *status* is either ACTive or COMPlete (designating *active* or *complete* items, respectively). We refer to the second constituent of an item $(B_1 \ldots B_m)$ as its *nucleus*, and denote it as $\hat{\gamma}$.

The generation process performs the same two kinds of chart operations as parsing, namely, *dot movement* and *completion*:

Dot movement: Given an active item $\alpha = [i^\alpha, B_1^\alpha \ldots B_l^\alpha, j^\alpha, \textsc{Act}]$, a complete item $\beta = [i^\beta, B^\beta, j^\beta, \textsc{Comp}]$ s.t. $j^\alpha = i^\beta$, and a rule $r = (A_1 \ldots A_m \Rightarrow A_{m+1})$ s.t. $l < m$, *dot movement* unifies $\hat{\alpha}$ and $\hat{\beta}$ with the corresponding constituents $A_1 \ldots A_l\ A_{l+1}$ of the rule, thus advancing the dot and creating a new item $I := I \cup [i^\alpha, A'_1 \ldots A'_{l+1}, j^\beta, \textsc{Act}]$.

Completion: Given an active item $\alpha = [i^\alpha, B_1^\alpha \ldots B_m^\alpha, j^\alpha, \textsc{Act}]$ and a rule $r = (A_1 \ldots A_m \Rightarrow A_{m+1})$, the *completion* operation matches the rule body with $\hat{\alpha}$, and creates a new item by taking the head A''_{m+1} after all the unifications performed: $I := I \cup [i^\alpha, A''_{m+1}, j^\beta, \textsc{Comp}]$.

These two operations are performed in exactly the same way and by the same chart processing module as for parsing. As mentioned above, this module runs blindly, without the notion of the actual task (parsing or generation) being carried out.

The two major differences between parsing and generation algorithms are in chart initialization (scanning) and interpretation of the final results. These phases are further explained in the sequel.

4.1.3 Initialization

The generation algorithm we propose implements a so-called chart generation, where a table (*chart*) is used to store intermediate results (items built by applying the rules of G_I to (parts of) the given logical form f). The chart is a two-dimensional array I of size $n \times n$, where n is the number of semantic primitives in f.

The given logical form is traversed recursively in a *postorder manner* and decomposed into a sequence $L = l_0 \ldots l_{n-1}$ of predicates and their arguments. Such a decomposition is performed along the features which define the predicate-argument structure, namely, *pred* and *arg*$_i$. Each l_i in the sequence L is a feature structure corresponding to some part of f. The *postorder traversal* of f orders the elements of L and thus numbers positions inside the logical form (vertices of the FS which correspond to roots of predicate and arguments substructures).

The flattening procedure ignores quantified variables which serve as arguments, and does not create l_i items for them, so that they are not entered into the chart during initialization. The reason is that since quantified variables do not correspond to any actual semantic primitive, scanning cannot match them against body constituents of (inverted) grammar rules. Let us consider an example of flattening a logical form which contains quantified variables. Consider the feature structure fs_5^e of Section 3.2.4, whose semantics encodes the logical form $\forall x(\texttt{man}(x) \rightarrow \texttt{smoke}(x))$ containing the (universally) quantified variable x. Observe that this logical form consists of three semantic primitives, namely, the (schematic) predicate $\forall x(P(x) \rightarrow Q(x))$, and two arguments – $\texttt{man}(x)$ and $\texttt{smoke}(x)$. Therefore, it is these three primitives that should result from flattening the semantics of fs_5^e. The postorder flattening discerns that the feature structure $< fs_5^e\ sem >$ has the features *pred*, *arg1* and *arg2*, and recursively proceeds to the first argument (via *arg1*). In its turn, the feature structure $< fs_5^e\ sem\ arg1 >$ is a predicate-argument structure by itself, hence *its arg1* feature should be explored first[2]. This is when the algorithm encounters a quantified variable (note that $< fs_5^e\ sem\ arg1\ arg1 >=< fs_5^e\ sem\ pred\ var >$, which encodes the quantified variable x). As mentioned above, we do not want to produce separate chart items for quantified variables (otherwise, the example logical form yields five distinct semantic primitives, instead of the expected three). Consequently, the flattening algorithm retreats, and only considers the entire expression $\texttt{man}(x)$. The expression $\texttt{smoke}(x)$ is treated similarly, and the algorithm ultimately returns to process the predicate $\forall x(P(x) \rightarrow Q(x))$.

Let us consider the nature of feature structures which may form nuclei of chart items. Recall that the *nucleus* of an item $\gamma = [j, B_1 \ldots B_m, k, status]$ is its second constituent: $\hat{\gamma} = B_1 \ldots B_m$. When the input logical form is decomposed into a postorder sequence of semantic primitives, the resulting FSs are in fact

[2]Pursuant to the postorder traversal policy.

semantic cores and not full semantics constructs. Observe that according to the grammar inversion algorithm (Step 2 of the Normalization algorithm and Step 1(d)ii of the Inversion algorithm) bodies of inverted rules are also built in the postorder manner, reflecting the predicate-argument structure of the head semantic form. The *semantics constituent* of inverted rules' bodies (the last body constituent of FI rules, created according to Step 2 of the Normalization algorithm) represents[3] the entire semantics of the rule head, and not just the semantic core. Therefore, to use the regular *dot movement* operation to match the inverted rule bodies against the chart entries, the latter should be in an appropriate format, namely, full semantics constructs with λ-envelopes around the semantic cores whenever applicable.

As an example, consider the logical form `smoke(john)` corresponding to a NL sentence "John smokes". The feature structure fs_2^e of Section 3.2.4 represents a possible generation input in this case:

$$fs_2^e = \left[\begin{array}{ll} \textbf{phrase} & \\ syn: & \left[\begin{array}{ll} \textbf{syn} & \\ cat: & [\textbf{s}] \end{array}\right] \\ sem: & \boxed{1}\left[\left[\begin{array}{ll} \textbf{arg\_1} & \\ pred: & [\textbf{smoke}] \\ arg1: & \boxed{5}[\textbf{john}] \end{array}\right]\right] \\ str: & \left\langle \boxed{5}, \boxed{1} \right\rangle \end{array}\right]$$

Postorder decomposition of $< fs_2^e\ sem >$ yields a sequence of two semantic primitives, namely, `john` and `smoke`:

$$[\textbf{john}] \left[\begin{array}{ll} \textbf{arg\_1} & \\ pred: & [\textbf{smoke}] \\ arg1: & [\textbf{john}] \end{array}\right]$$

Inverted rules I_{11} and I_6 should apparently be the first applied in a bottom-up generation. The body of I_{11} is readily applicable to the former FS which encodes the semantics of `john`. On the other hand, the body of I_6 is not unifiable with the latter FS, since it has a λ-binder around the semantics of `smoke`, thus encoding the expression $\lambda x.\texttt{smoke}(x)$. Therefore, to make the inverted rules applicable to the generation input, a method is needed to map semantic cores (e.g., `smoke` in the above example) into full semantic expressions (e.g., $\lambda x.\texttt{smoke}(x)$). In the terms of the example, the chart item representing the predicate `smoke` should contain the full semantic construct[4]:

$$\left[\begin{array}{ll} \boldsymbol{\lambda}\textbf{-bind} & \\ var: & \boxed{5}[\textbf{john}] \\ \underline{rest}: & \left[\left[\begin{array}{ll} \textbf{arg\_1} & \\ pred: & [\textbf{smoke}] \\ arg1: & \boxed{5} \end{array}\right]\right] \end{array}\right]$$

Semantics Knowledge Base (SKB)

Semantic primitives which comprise logical forms originate from the logical predicates (possibly 0-ary, as in the case of scalar constants) of lexicon entries. To facilitate generation with inverted grammars, an additional transformation is applied to G_O apart from the inversion. Namely, the lexicon and the Connective Registry of G_O are processed to create a so-called *Semantics Knowledge Base (SKB)*. An SKB item corresponding to a lexicon entry "word" $\rightarrow H$ (or a CR entry H) is a feature structure fs such that $fs =< H\ sem >$. In other words, SKB stores entire semantics of lexicon (Connective Registry) entries. The notion of the *semantic core* is also applicable to SKB items, so that the semantic core of fs is $< fs\ rest^* > = < H\ sem\ rest^* >$.

The SKB is supplied to the generator as a part of the inverted grammar.

[3]The Normalization algorithm makes the *semantics constituent* of a Functor-Introducing rule reentrant with the entire semantics of the rule's head.

[4]The feature structure shown is ostensibly anomalous since its λ-variable is already instantiated to [**john**]. This is due to the unification performed between $< fs_2^e\ sem >$ and the corresponding SKB entry, when retrieving the semantic expression $\lambda x.\texttt{smoke}(x)$ (see below).

Initialization with SKB entries

As explained above, the chart has to be initialized with full semantics constructs rather than with the immediate elements of L (the sequence of semantic primitives resulting from flattening the given logical form). To this end, the generation algorithm consults the SKB and associates each l_i with a set of SKB entries $K_i = \{K_{i_1}, \ldots, K_{i_q}\}$, such that for each K_{i_p} its semantic core (denoted $SC(K_{i_p})$) *is unifiable* with l_i. The case when more than one SKB item corresponds to a particular element l_i (i.e., $q > 1$) occurs when the input grammar contains synonyms (different words having similar semantics).

During initialization, a set of chart items of the form $[i, K'_{i_p}, i+1, \textsc{Comp}]$ is created for each element l_i of L, such that $K_{i_p} \in K_i$ and K'_{i_p} is K_{i_p} as changed upon the unification of $SC(K_{i_p})$ and l_i *in the context of* K_{i_p}. This step actually corresponds to the notion of *scanning* in chart parsing. Note that all the elements of the K_i set correspond to the same part of the input semantics (l_i), therefore they all enter the same chart cell $(i, i+1)$. This way, the scanning phase fills the chart diagonal with the elements of input.

From this point, generation proceeds in exactly the same way as chart parsing. The initialization step creates *prediction* items for grammar rules. Then the algorithm proceeds to the generation phase per se, which invokes regular chart operations of *dot movement* and *completion*. The aim of this phase is to combine items using inverted grammar rules, until a new item is created which spans the entire semantics input, thus producing a phrase corresponding to the given logical form f.

4.1.4 Verbalization

As mentioned in Chapter 3 (see Section 3.2.2 and Section 3.3), feature structures of G_N and G_I rules are endowed with a dedicated feature *str* which traces the order of constituents in corresponding G_O rules. This is actually the order in which NL words are combined to form phrases, and it is this order that should be reconstructed during generation. According to Steps 1a and 4b of the normalization algorithm, the *str* feature encodes ordered sequences of semantic primitives, each corresponding to the semantic core of some lexical item. A phrase spanned by the semantics of a feature structure can be produced by analyzing the value of its *str* feature, retrieving the respective words from the lexicon and concatenating them.

If generation terminates, it ends up with a (possibly empty) set of FSs, such that for each feature structure F' its semantics is *at least as specific* as the input logical form f ($< F' \; sem > \sqsupseteq f$). Since F' has been built using the rules of G_I and according to the Inversion algorithm, $< F' \; str >$ encodes the words of the generated phrase w. For notational convenience, let us denote $< F' \; str > = f'_1 \ldots f'_t$, where f'_i are semantic primitives, which constitute semantic cores of lexicon entries. To map the elements of $< F' \; str >$ into NL words, the generator again consults the SKB. For each f'_i an SKB item is located such that its semantic core subsumes f'_i. Then the word is retrieved from the lexicon entry, which served the prototype for this SKB item. The resultant words are concatenated in the order induced by the elements of $< F' \; str >$. If the lexicon contains synonyms, several SKB items (and, therefore, several words) may correspond to the same f'_i. In such a case, several phrases (actually, *paraphrases*[5]) will be created for F', with different words in the $i-th$ position. It is obvious that there may be more than one position in the resultant phrase where synonymic words exist, thus leading to numerous paraphrases.

The case when generation yields several feature structures represents *structural ambiguity*; in such a case each resultant FS corresponds to a different generation tree.

4.2 The generation algorithm

4.2.1 Terminology

Scalar FS f is a **scalar** FS if it only has a type but no features. The fact that a FS f is scalar is denoted by **scalar**(f).

Empty MRS A MRS having an empty set of nodes is an **empty** MRS, denoted with λ.

[5] *Paraphrases* are different NL expressions with the same meaning.

4.2.2 The algorithm

In what follows we describe a bottom-up chart generation algorithm.

Let G_I be the input (inverted) grammar, and F be the feature structure which encodes the desired meaning.

```
0. AUXILIARY DEFINITIONS
   $I := \emptyset$; /* the chart */
   $L := \varepsilon$; /* a list to store FSs obtained from flattening the given logical form */
   $n := 0$; /* the number of of semantic primitives in the given logical form */
   /* Procedure flatten assumes its parameter to be a feature structure, which represents some logical
   form and conforms to the assumptions of Section 3.2. The procedure decomposes the given FS into a
   sequence of semantic primitives along the predicate-argument structure. */
   procedure flatten(fs)
     (a) If scalar(fs) and fs is not a quanitfied variable, then $L[n] := fs$; $n := n + 1$;
     (b) Else if
           $fs = \begin{bmatrix} \textbf{λ-bind} & \\ var: & \boxed{1}\begin{bmatrix}\textbf{sem}\end{bmatrix} \\ rest: & [C_0] \end{bmatrix}$
         then flatten($C_0$);
     (c) Else if
           $fs = \begin{bmatrix} \textbf{arg\_k} & \\ pred: & [C_0] \\ arg1: & [C_1] \\ \vdots & \\ argk: & [C_k] \end{bmatrix}$
         then
           i. For $i := 1$ to $k$
                flatten($C_i$);
           ii. $L[n] := fs$; $n := n + 1$;
     (d) Otherwise, fs is not a predicate-argument structure; abort.
   end /* procedure flatten */
1. CHART INITIALIZATION
   (a) [Flattening]
       Execute flatten($< F\ sem >$)
   (b) [Scanning]
       For $i := 0$ to $n - 1$
         • For every $K \in SKB$
           s.t. $SC(K)$ is the semantic core of $K$ and $SC(K) \sqcup L[i] \neq \top$
           $I := I \cup [i, K', i + 1, \textsc{Comp}]$;
           where $K'$ is $K$ as changed upon the unification of $SC(K)$ and $L[i]$ in the context of $K$.
   (c) [Prediction]
       For $i := 0$ to $n - 1$
       $I := I \cup [i, \lambda, i, \textsc{Act}]$;
2. CHART GENERATION PROPER
   Perform the following operations in any order, until quiescence (i.e., until no new items can be added
   to $I$ anymore):
```

```
(a) Dot movement
    If
      • $\exists r = (A_1 \dots A_m \Rightarrow A_{m+1}) \in G_I, m > 0$;
      • $\exists l < m$;
      • $\exists \alpha \in I : \alpha = [i^\alpha, B_1^\alpha \dots B_l^\alpha, j^\alpha, \textsc{Act}]$;
      • $\exists \beta \in I : \beta = [i^\beta, B^\beta, j^\beta, \textsc{Comp}], j^\alpha = i^\beta$;
    then
      • $A'_1 \dots A'_l := (A_1 \dots A_l) \sqcup_r (B_1^\alpha \dots B_l^\alpha)$,
        where $\sqcup_r$ designates unification in the context of $r$, resulting in $r' = (A'_1 \dots A'_m \Rightarrow A'_{m+1})$;
      • $A''_{l+1} := A'_{l+1} \sqcup_{r'} B^\beta$;
      • $I := I \cup [i^\alpha, A''_1 \dots A''_{l+1}, j^\beta, \textsc{Act}]$;
(b) Completion
    If
      • $\exists r = (A_1 \dots A_m \Rightarrow A_{m+1}) \in G_I, m > 0$;
      • $\exists \alpha \in I : \alpha = [i^\alpha, B_1^\alpha \dots B_m^\alpha, j^\alpha, \textsc{Act}]$;
    then
      • $A'_1 \dots A'_m := (A_1 \dots A_m) \sqcup_r (B_1^\alpha \dots B_m^\alpha)$,
        where $\sqcup_r$ designates unification in the context of $r$, resulting in $r' = (A'_1 \dots A'_m \Rightarrow A'_{m+1})$;
      • $I := I \cup [i^\alpha, A'_{m+1}, j^\alpha, \textsc{Comp}]$;

3. VERBALIZATION
   For every $F'$ such that $[0, F', n, \textsc{Comp}] \in I$
   /* print paraphrases with the desired meaning */
   (a) Let $< F'\ str >= f'_1 \dots f'_t$
   (b) For $i := 1$ to $t$
         • For every $K \in SKB$
           s.t. $SC(K)$ is the semantic core of $K$, $SC(K) \sqcup f'_i \neq \top$ and
           $lex = (\text{“word”} \rightarrow H)$ is the lexicon entry such that $K =< H\ sem >$:
           /* print (a synonym of) the i-th word of the generated phrase */
           print(“word”);
```

4.3 Sample generation

To exemplify the Generation algorithm we show a sample chart generation with the grammar G_I^e. The input to the generator is the logical form `mod(smoke(john), today)`, encoded in feature structures as

$$F = \begin{bmatrix} \textbf{phrase} & \\ syn: & \begin{bmatrix} \textbf{syn} & \\ cat: & \begin{bmatrix} \textbf{s} \end{bmatrix} \end{bmatrix} \\ sem: & \begin{bmatrix} \textbf{arg\_2} & \\ pred: & \begin{bmatrix} \textbf{mod} \end{bmatrix} \\ arg1: & \begin{bmatrix} \textbf{arg\_1} & \\ pred: & \begin{bmatrix} \textbf{smoke} \end{bmatrix} \\ arg1: & \begin{bmatrix} \textbf{john} \end{bmatrix} \end{bmatrix} \\ arg2: & \begin{bmatrix} \textbf{today} \end{bmatrix} \end{bmatrix} \end{bmatrix} \tag{4.1}$$

For the sake of compactness we only show creation of items that are required for the generation being demonstrated. During the actual generation process a number of unnecessary items are created as a by-product – those are disregarded here. We also do not show *prediction* items as they are numerous and trivial in their structure.

A note is appropriate about the way chart items are displayed. Item nuclei are MRSs and may hence have reentrancies among their parts. It should be noted however that such reentrancies (as well as the corresponding tags) are purely local, and nothing can be shared between the nuclei of two distinct items. Therefore, the use of identical reentrancy tags in different items only serves expository purposes and does not imply shared substructures.

4.3.1 Chart initialization

Flattening

The following sequence of semantic primitives results from flattening $< F\ sem >$ $(n = 4)$:

$$L = \left\langle \begin{bmatrix}\textbf{john}\end{bmatrix}, \begin{bmatrix}\textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \begin{bmatrix}\textbf{john}\end{bmatrix}\end{bmatrix}, \begin{bmatrix}\textbf{today}\end{bmatrix}, \begin{bmatrix}\textbf{arg\_2} & \\ \textit{pred}: & \begin{bmatrix}\textbf{mod}\end{bmatrix} \\ \textit{arg1}: & \begin{bmatrix}\textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \begin{bmatrix}\textbf{john}\end{bmatrix}\end{bmatrix} \\ \textit{arg2}: & \begin{bmatrix}\textbf{today}\end{bmatrix}\end{bmatrix} \right\rangle \tag{4.2}$$

Scanning

During the scanning phase each element l_i of L (4.2) is matched by an SKB entry whose semantic core is unifiable with it. The unifications are then performed, and the results are used to create the following chart initialization items:

$$[0, \begin{bmatrix}\textbf{john}\end{bmatrix}, 1, \textsc{Comp}] \tag{4.3}$$

$$[1, \begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5}\begin{bmatrix}\textbf{john}\end{bmatrix} \\ \underline{\textit{rest}}: & \begin{bmatrix}\begin{bmatrix}\textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \boxed{5}\end{bmatrix}\end{bmatrix}\end{bmatrix}, 2, \textsc{Comp}] \tag{4.4}$$

$$[2, \begin{bmatrix}\textbf{today}\end{bmatrix}, 3, \textsc{Comp}] \tag{4.5}$$

$$[3, \begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ \textit{var}: & \boxed{5}\begin{bmatrix}\textbf{john}\end{bmatrix} \\ \underline{\textit{rest}}: & \begin{bmatrix}\begin{bmatrix}\textbf{arg\_2} & \\ \textit{pred}: & \begin{bmatrix}\textbf{mod}\end{bmatrix} \\ \textit{arg1}: & \begin{bmatrix}\textbf{arg\_1} & \\ \textit{pred}: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ \textit{arg1}: & \boxed{5}\end{bmatrix} \\ \textit{arg2}: & \begin{bmatrix}\textbf{today}\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}, 4, \textsc{Comp}] \tag{4.6}$$

4.3.2 Chart generation

The chart generation shown here can be seen as the reverse of the derivation starting from the same input. Compare the generation below with the sample derivation of Section 3.5.4, which also starts from the feature structure F given by (4.1) above.

Dot movement expands the prediction item $[0, \lambda, 0, \textsc{Act}]$ with initialization item (4.3) using rule I_{11}:

$$[0, \begin{bmatrix}\textbf{john}\end{bmatrix}, 1, \textsc{Act}] \tag{4.7}$$

Completion uses rule I_{11} to convert item (4.7) into

$$[0, \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{np}]\end{array}\right]\\ \mathit{sem}: & \boxed{1}[\textbf{john}]\\ \mathit{str}: & <\boxed{1}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right], 1, \textsc{Comp}] \tag{4.8}$$

Dot movement and *completion* are applied in a similar way to item (4.5) using rule I_{14}, resulting in the item

$$[2, \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{advp}]\end{array}\right]\\ \mathit{sem}: & \boxed{1}[\textbf{today}]\\ \mathit{str}: & <\boxed{1}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right], 3, \textsc{Comp}] \tag{4.9}$$

Rule I_7 and the prediction item $[0, \lambda, 0, \textsc{Act}]$ are used to convert the complete item (4.8) to an active one. Then *dot movement* matches the resulting item with the first body constituent of the rule. Another *dot movement* matches item (4.4) with the second body constituent (denoted by the tag $\boxed{17}$ in the rule), instantiating the semantics of the rule head at the same time. A subsequent *completion* operation creates the following item:

$$[0, \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{vi}]\end{array}\right]\\ \mathit{sem}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\boldsymbol{\lambda}\textbf{-bind}}\\ \mathit{var}: & \boxed{5}[\textbf{john}]\\ \underline{\mathit{rest}}: & \boxed{6}\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_1}}\\ \mathit{pred}: & [\textbf{smoke}]\\ \mathit{arg1}: & \boxed{5}\end{array}\right]\end{array}\right]\\ \mathit{str}: & <\boxed{6}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{ne\_list}}\\ \mathit{hd}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{np}]\end{array}\right]\\ \mathit{sem}: & \boxed{5}\\ \mathit{str}: & <\boxed{5}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right]\\ \mathit{tl}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right]\end{array}\right], 2, \textsc{Comp}] \tag{4.10}$$

The prediction item $[0, \lambda, 0, \textsc{Act}]$ is used again, now in conjunction with rule I_2, to convert the complete item (4.10) to an active one. The resultant item is matched against the first body constituent of the rule, while the values of the tags $\boxed{5}$ and $\boxed{6}$ in the rule are unified with their namesakes in chart item. Two additional *dot movements* unify items (4.9) and (4.6) with the remaining body constituents. Finally, a *completion* operation creates the following item:

$$[0, \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{phrase}}\\ \mathit{syn}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{syn}}\\ \mathit{cat}: & [\textbf{s}]\end{array}\right]\\ \mathit{sem}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_2}}\\ \mathit{pred}: & [\textbf{mod}]\\ \mathit{arg1}: & \boxed{6}\left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{arg\_1}}\\ \mathit{pred}: & [\textbf{smoke}]\\ \mathit{arg1}: & \boxed{5}[\textbf{john}]\end{array}\right]\\ \mathit{arg2}: & \boxed{7}[\textbf{today}]\end{array}\right]\\ \mathit{str}: & <\boxed{5},\boxed{6},\boxed{7}>\\ \mathit{args}: & \left[\begin{array}{ll}\multicolumn{2}{l}{\textbf{args}}\\ \mathit{larg}: & [\textbf{e\_list}]\end{array}\right]\end{array}\right], 4, \textsc{Comp}] \tag{4.11}$$

The semantics of item (4.11) subsumes the entire input, therefore the sample generation terminates successfully.

4.3.3 Verbalization

The *str* feature of the generation result (the nucleus of item (4.11)) lists the following three semantic primitives:

$$\left\langle \begin{bmatrix}\textbf{john}\end{bmatrix}, \begin{bmatrix}\textbf{arg\_1} & \\ pred: & \begin{bmatrix}\textbf{smoke}\end{bmatrix} \\ arg1: & \begin{bmatrix}\textbf{john}\end{bmatrix}\end{bmatrix}, \begin{bmatrix}\textbf{today}\end{bmatrix}\right\rangle$$

Consulting the SKB shows that these semantic primitives correspond to lexicon entries O_8, O_{11} and O_{14}, respectively. The words defined by these entries have no synonyms, therefore the generation yields a single phrase "John smokes today".

Clearly, if the generated sentence is parsed w.r.t. G^e_O resulting in a feature structure F', then the value $< F' \ sem >$ is exactly the logical form $< F \ sem >$ which served the input for the sample generation above.

4.3.4 Generation of partial phrases

Let us consider a slightly different generation scenario from the same input (4.1). At the very end, instead of applying rule I_2 to create item (4.11), the new scenario applies rule I_4 to items (4.10), (4.9) and (4.6), and produces the following item:

$$[0, \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & \begin{bmatrix}\textbf{vp}\end{bmatrix}\end{bmatrix}\\ sem: & \begin{bmatrix}\boldsymbol{\lambda}\textbf{-bind} & \\ var: & \boxed{5}\begin{bmatrix}\textbf{john}\end{bmatrix} \\ \underline{rest}: & \begin{bmatrix}\textbf{arg\_2} & \\ pred: & \begin{bmatrix}\textbf{mod}\end{bmatrix}\\ arg1: & \boxed{6}\begin{bmatrix}\textbf{arg\_1} & \\ pred: & \begin{bmatrix}\textbf{smoke}\end{bmatrix}\\ arg1: & \boxed{5}\end{bmatrix}\\ arg2: & \boxed{7}\begin{bmatrix}\textbf{today}\end{bmatrix}\end{bmatrix}\end{bmatrix}\\ str: & <\boxed{6},\boxed{7}>\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & \begin{bmatrix}\textbf{ne\_list} & \\ hd: & \begin{bmatrix}\textbf{phrase} & \\ syn: & \begin{bmatrix}\textbf{syn} & \\ cat: & \begin{bmatrix}\textbf{np}\end{bmatrix}\end{bmatrix}\\ sem: & \boxed{5}\\ str: & <\boxed{5}>\\ args: & \begin{bmatrix}\textbf{args} & \\ larg: & [\textbf{e\_list}]\end{bmatrix}\end{bmatrix}\\ tl: & \begin{bmatrix}\textbf{e\_list}\end{bmatrix}\end{bmatrix}\end{bmatrix}\end{bmatrix}, 4, \textsc{Comp}] \tag{4.12}$$

The resultant item (4.12) obviously covers the entire input (4.1), and therefore constitutes a legitimate generation result. On the other hand its *str* feature encodes only the partial phrase "smokes today", since the NP representing "John" still remains on the argument list and does not contribute to the generated phrase. Observe also that the syntactic category of (4.12) is VP, which is different from that of the generation input (4.1) — $< F \ syn \ cat >= S$.

4.4 An abstract machine for chart generation

This section describes an Abstract Machine which implements the chart generation algorithm of Section 4.2. In what follows we only discuss the extensions to $\mathcal{A}$MALIA which were necessary to facilitate generation.

As mentioned above (Section 4.1.1), the main differences between parsing and generation in $\mathcal{A}$MALIA lie in chart initialization and interpretation of the final results. To this end, the control module of $\mathcal{A}$MALIA was augmented to perform the correct action depending on the specific task (parsing/generation) at hand.

4.4.1 Augmentation of the Control Module

In the case of generation, the AM *program* is obtained by *inverting* and then *compiling* the given grammar. The input logical form which gives the desired meaning makes the *query*, which is also compiled to the Abstract Machine instructions. Similar to parsing, the instructions resulting from query processing are the first to be executed on the chart (this phase in fact performs chart initialization). Then the program is executed, to perform chart generation per se. Finally, a grammar-independent routine is invoked to actually build the generated phrase, which is encoded in chart items (one or more, if any) spanning the entire input.

Grammar compilation

The supplied grammar is normalized and inverted according to the respective algorithms of Sections 3.4.7 and 3.5.2. The grammar is then compiled into an Abstract Machine program using exactly the same compilation scheme as for parsing. The lexicon and the Connective Registry are then converted into the Semantic Knowledge Base (SKB), by extracting the $< H\ sem >$ substructure of each lexicon entry of the form "word" $\rightarrow H$ or a CR entry H (see Section 4.1.3).

The compiler passes to the generator some of the symbol tables it has built. In particular, it passes type and feature lists, an encoding of the type hierarchy, and type unification information.

Chart initialization (scanning)

To perform chart initialization, the semantics f of the given query α ($f =< \alpha\ sem >$) is first flattened into a sequence L of semantic primitives (see procedure **flatten** of the Generation algorithm, Section 4.2). Then for each element l_i of L the SKB entries $K_i = \{K_{i_1}, \ldots, K_{i_q}\}$ are collected whose semantic core is unifiable with l_i. Unifications are performed, and the resultant set of feature structures $K'_i = \{K'_{i_1}, \ldots, K'_{i_q}\}$ are used to initialize the chart.

It is *not* sufficient to only check subsumption between the elements of L and the SKB entries for the following reason. The given logical form f may contain additional information compared with that found in the original grammar lexicon (and, consequently, in the SKB). If parts of the query are not unified with the SKB, then such information is irretrievably lost, and cannot affect (constrain) the generation process despite the fact it was supplied in the input. Unification with parts of the query adds to the SKB entries any information not found in the respective lexicon entries but present in the query. It should be emphasized that the unification in this phase is performed on feature structures, and no use is made of the chart whatsoever.

To initialize the chart, the feature structures resulting from the unification above have to be compiled into the abstract machine code. This necessitates to perform compilation during generation, using the symbol tables prepared while compiling the grammar. The outcome of the compilation are functions in the AM instruction language, which need to be executed on the chart to initialize it. For each l_i, the functions corresponding to $K'_i = \{K'_{i_1}, \ldots, K'_{i_q}\}$ are invoked on the same chart cell (namely, the cell $(i, i+1)$ which represents l_i).

At the end of this phase, the chart is initialized for the entire query, and the generator proceeds to executing the program.

Chart processing

Chart generation is performed by executing the program (i.e., the inverted grammar compiled into AM instructions) on the chart initialized with the query. The control module of the abstract machine implements the chart processing algorithm (which realizes the *dot movement* and *completion* operations) and invokes the program in exactly the same way as for parsing. During this phase there is no notion of the specific task (parsing or generation) being carried out.

Interpreting the results of chart generation (verbalization)

A successful generation ends up with a non-empty set of feature structures, whose *str* feature encodes the generated phrase. What remains to be done is to convert this encoding into sequences of NL words.

According to the grammar inversion procedure (Chapter 3), the *str* feature encodes lists of semantic primitives, represented with feature structures. It is therefore necessary to decompose such lists into sequences of elements, and to map the resulting elements into words. Should several synonymic words correspond to the same semantic primitive, all of them are produced in the output in the respective position. Currently, $\mathcal{A}$MALIA uses braces to denote such alternative words, e.g., "John {loves, likes} Mary passionately".

To facilitate further processing, each resultant feature structure is converted from its AM heap encoding to a regular representation used during grammar compilation. This is performed so that subsumption tests with SKB entries can be performed without the use of the chart.

For each feature structure, the value of its *str* feature is retrieved, and then flattened[6] into a linear sequence of semantic primitives. Then the SKB is consulted for the second time during generation, and for each semantic primitive all the SKB entries are found whose semantic core subsumes (i.e., is more general than) this primitive. This way the generated phrase may be expressed with natural language words, since each SKB entry corresponds to a lexicon item of the original grammar.

4.4.2 Application

The above extensions to the Abstract Machine for Parsing have been implemented, and merged into a single application with the original $\mathcal{A}$MALIA. Both processing directions share the grammar compilation and chart processing modules, as well as the common graphical user interface. When invoked in the parsing mode, the program compiles the given grammar and parses queries (NL phrases) with respect to it. When invoked in the generation mode, the extended control module inverts the input grammar prior to compilation, and then performs generation along the above guidelines. In the latter case queries are supplied in an auxiliary file, encoded in ALE notation for feature structures.

The software implementation of the generation extensions to $\mathcal{A}$MALIA is outlined in the next chapter.

[6]Observe that according to Section 3.4.6, values of the *str* feature are *tree-like* structures encoded in the usual linked list notation.

Chapter 5

Software Implementation

This chapter describes the implementation of $\mathcal{A}$MALIA. Since the implementation of the basic $\mathcal{A}$MALIA architecture as well as the chart mechanism are detailed elsewhere (Wintner, 1997; Wintner, Gabrilovich, and Francez, 1997a), we mainly focus the discussion below on the generation extensions. The overview of $\mathcal{A}$MALIA and its functionality is given only briefly, for the sake of completeness of the presentation.

$\mathcal{A}$MALIA is implemented in the C programming language, complying to the ANSI-C requirements (Kernighan and Ritchie, 1988). The *lex* and *yacc* tools (Aho, Sethi, and Ullman, 1986) were used to implement the input acquisition module, and the *Tcl/Tk* toolkit (Ousterhout, 1994) was used to build the graphical user interface. The application is compatible with a variety of platforms, such as SUN and SILICON GRAPHICS workstations running UNIX operating system, as well as IBM PC running WINDOWS'95 and LINUX. There are two versions of $\mathcal{A}$MALIA : an interactive, user-friendly program with a GUI, and a non-interactive but more efficient version which can be used for batch processing. The former program is ideally suited for interactive grammar design and development, as it allows the user to debug grammars by tracing rule applications step-by-step, and by examining the internal machine state at any step. Since $\mathcal{A}$MALIA facilitates both parsing and generation, the interactive version can be viewed as an environment for developing *reversible grammars.*

Both versions of $\mathcal{A}$MALIA can operate in two modes: **parsing** and **generation**.

1. In the case of **parsing**, an input consists of a natural language grammar G (encoded in a subset of ALE specification language (Carpenter, 1992a)) and a phrase w to be parsed (given as a sequence of natural language words). The grammar is *compiled* into a *program* which consists of AM instructions. Then lexical lookup is performed on the given phrase by associating the input words with (one or more) lexicon entries of the grammar. The resultant sequence of feature structures is also compiled to the machine code, thus forming the *query.*

   Then the control module of the abstract machine takes over. Its algorithm realizes chart parsing by repeatedly applying the *parsing step operator* $T_{G,w}$ as defined in (Wintner, 1997, p. 32), until its least fix-point is reached (if the process terminates at all). To this end the *query* is first executed to initialize the chart, performing the *scanning* step of chart parsing. The control module makes use of the program to create *prediction* items and handle the case of *ε-rules*, and then to effect the chart operations of *dot movement* and *completion.* If the computation is terminating, then in case of a *successful* parsing the chart contains one or more items which span the entire input — these items contain feature structures which represent the analysis results. If the parsing is *unsuccessful* the chart contains no such items.

2. In the case of **generation**, an input consists of a NL grammar G and a feature structure α (both encoded in the subset of ALE specification language). The semantics of the given feature structure (i.e., $f =< \alpha\ sem >$) encodes a logical form which defines a meaning to generate. The generator then has to create one or more phrases in the language of G so that they possess the desired meaning given as input.

In the light of the previous discussion (see Chapters 3 and 4) the given grammar is *inverted* into a form G_I which is more suitable for effective generation. The inverted grammar is then compiled into machine code in exactly the same way as for parsing. The logical form f is decomposed into a linear sequence of semantic primitives, which upon an appropriate lexical lookup (consulting the SKB) becomes the generation *query*.

From now on the control module of AMALIA executes the same chart algorithm as in the case of parsing (this can be viewed as *"parsing"* the given logical form according to the *inverted* grammar). If the computation terminates successfully, the chart contains one or more feature structures which span (subsume) the entire semantic input. What remains to be done in order to present the generation results in a familiar form, is to perform the *verbalization* step. This step extracts the value of the *str* feature of each resulting feature structure, and performs a variant of lexical lookup to convert it to natural language words.

AMALIA's generation module operates in accordance with the above guidelines. In what follows, Section 5.1.1 outlines the process of generation and discusses several features of the application. Sections 5.1.2 and 5.1.3 cover more technical aspects of the implementation of grammar compilation for generation and chart generation, respectively.

5.1 Generation additions to AMALIA

5.1.1 Overview

As follows from the above description, generation extensions are introduced in AMALIA at two levels, namely in the grammar **compiler** and the **interpreter** of AM instructions. Whenever possible, the regular data flow of parsing is obeyed during generation as well, while in the rest of the cases operations specific to generation are performed either in addition to or instead of those specific to parsing.

Grammar compilation for generation

During compilation for generation, an additional transformation is performed on the input grammar G, immediately upon grammar acquisition but prior to compilation. The input grammar is *normalized* and then *inverted* using the respective algorithms of Chapter 3. For the grammar to be *invertible* it should comply with the restrictions listed in Section 3.2. Since AMALIA automatically performs type inference during input acquisition, the auxiliary features *str* and *args* only have to be included in the type hierarchy, and does not need to be explicitly used in the grammar rules or lexicon entries. After partial descriptions are expanded, these features may be used by the normalization and inversion procedures.

The inverted grammar is represented using Multi-Rooted Structures in exactly the same way as the original grammar. Therefore it can be immediately compiled into AM instructions, using the compilation procedure for parsing grammars. Thus both original (parsing) grammars and inverted grammars are compiled using the same AM instruction set, to be then executed by the same interpreter.

As in the case of parsing, the compiler passes to the interpreter a number of symbol tables. Namely, type and feature lists, an encoding of the type hierarchy and type unification information are passed.

Lexicon processing The lexicon of the given grammar is also processed differently for parsing and generation. If G is compiled for parsing, the lexicon is compiled into AM instructions together with the grammar. For each word in the lexicon the compiler creates a function, which can be executed on the chart during the *scanning* phase, to initialize it appropriately if this word occurs in input. In addition to that, a special *"words"* file is created, which associates each word with the label of the corresponding function in the program code.

If G is compiled for generation, the lexicon needs to be *disambiguated* prior to grammar compilation. To this end the lexicon is searched for *"ambiguous"* entries, i.e., entries representing *homonyms* (or, more precisely, *homographs*). Such entries associate a single spelling with several disjunctive feature structures which represent different meanings. Since generation is *"meaning-oriented"*, such sets of FSs have to be decomposed into several simple lexicon entries, where each spelling is associated with a single feature structure.

During grammar inversion the lexicon induces a set of (lexicon-derived) Functor-Introducing rules (see Step 4 of the Normalization algorithm in Section 3.4.7). After the grammar is inverted, the lexicon and the Connective Registry (if any is supplied with the grammar) are transformed into the Semantic Knowledge Base. To this end, each lexicon entry of the form "word" $\rightarrow H$ is converted into an SKB item "word" $\rightarrow fs$, where $fs =< H\ sem >$, and each CR entry H is converted into $< H\ sem >$ (cf. Section 4.1.3). It should be noted that for the reasons explained in Section 4.4.1, in the generation mode the lexicon is not compiled along with the grammar. Instead, individual SKB entries which correspond to parts of the given query are compiled immediately prior to the generation, and are then executed to initialize the chart according to the input.

The generation cycle

The generation cycle performs generation from a *program* and a *query* it receives as input. The cycle starts with initialization of the abstract machine by building its main data structures: the *heap*, the *chart*, *general purpose* and *special purpose registers*, the *trail* and the *code area*. Then the symbol tables prepared by the compiler (including the SKB file) are read, followed by the acquisition of the program, when the program is loaded into the code area.

Chart initialization The query has to be executed before the program, hence the next step is to process the query. First, the query file is parsed in order to build the query representation as a feature structure. According to Step 1a of the Generation algorithm (Section 4.2) the semantics of the query is flattened into a sequence of semantic primitives $L = l_0 \ldots l_{n-1}$. Then for each l_i, all the SKB entries $K_i = \{K_{i_1}, \ldots, K_{i_q}\}$ whose sematic core subsumes l_i are collected. For each K_{i_j}, its sematic core is then unified with l_i, and the outcome of the unification (denoted K'_{i_j}) induces a chart initialization item. These items are compiled similarly to the way lexicon entries are compiled for parsing, using the symbol tables built during grammar compilation. The compiled code for all K'_{i_j} is executed on the chart cell which corresponds to l_i. This way all the SKB entries that represent words whose meaning subsumes that of l_i enter the corresponding chart cell, in accordance with Step 1b of the Generation algorithm.

Compilation of the SKB Compilation of SKB items for generation is slightly different from the corresponding compilation of lexicon entries for parsing. In the case of parsing, the lexicon associates each word with one or more feature structures which together form a (disjunctive) lexicon entry. Thus during scanning all the feature structures corresponding to a given word are available immediately. Therefore the chart cell corresponding to a certain word in the input is initialized with those and only those feature structures which comprose the respective lexicon entry. When the compiled code of a lexicon entry is executed on a chart cell, the chart control module may proceed to the next cell.

In contrast, in the case of generation each semantic primitive may correspond to a number of SKB entries whose semantic cores subsume it. These do not necessarily come together, and in the general case may be scattered all over the SKB (for example, SKB entries induced by ambiguous words in the original lexicon). Consequently, it is unclear how many SKB items will be used to initialize a chart cell until all the SKB is scanned. To this end, each individual SKB item is compiled with a provision that another one will enter the same chart cell. This is realized by compiling each SKB item with `'same_word'` as the last instruction. When all the SKB items corresponding to a given semantic primitives have been collected and compiled, the `'proceed'` instructions is created explicitly. This way when initialization of a chart cell is complete, the control module automatically proceeds to the next cell.

Chart generation After the chart is initialized, the chart engine assumes control and performs chart generation per se, thus effecting first Step 1c, and then Step 2 of the Generation algorithm. To this end the program (which is already loaded to the code area) is invoked, by exactly the same mechanism and in the same way as for parsing.

Verbalization If the program terminates, what remains to be done is to interpret the chart results (if any) according to the meaning in the generation mode, and output the phrase(s) generated. During the chart generation all the feature structures are built in the main memory of $\mathcal{A}$MALIA (namely, on the *heap*). From the practical point of view, it is more convenient to perform verbalization on feature structures in their regular representation (used during grammar compilation), instead of on their heap representation. The reason for this is that verbalization needs to perform subsumption check and list processing operations, and the corresponding functions have already been used in the compiler.

To this end the final state of the heap is examined, collecting the items which contain feature structures that span the entire input. Each such FS is then converted to a convenient representation, and the value of its *str* feature is extracted. The *str* list is then decomposed from its tree-like structure into a linear sequence of semantic primitives. For each component of the *str* list, the SKB is scanned for entries whose semantic core subsumes this component, and the corresponding words are produced. If there are several SKB entries which match the same component, the corresponding words are pooled and are denoted with braces in the output.

5.1.2 Functional breakdown of grammar compilation for generation

Data structures

The main data structures used during compilation are those representing grammars and machine instructions. Each grammar rule is represented as a Multi-Rooted Structure whose distinguished first root corresponds to the rule head. Lexicon entries are also embedded into MRSs, with the only distinction that all roots have the same status, and correspond to the individual feature structures of the disjunctive lexicon entry. Connective Registry entries are represented as MRS of length 1, each introducing some semantic primitive. In general, there is no notion of a “stand-alone” feature structure which does not reside inside some “host” MRS.

Grammars are compiled into instructions of the AM language. Each instruction has an operation code which identifies this instruction, an optional label, up to three arguments and a string comment, which makes it easier for humans to read the AM code.

Main functions and data flow

1. First, the grammar specification file (in ALE format) is parsed using a YACC-based parser (function `'yyparse'`). Various grammar-dependent symbol tables, such as type and feature lists, type unification table and the type hierarchy are built during grammar acquisition. While building the type hierarchy the parser searches for the types and features prescribed by the minimum required type hierarchy (see Section 3.2.3). These types and features are used by numerous functions throughout the system, and for that reason their identifiers are made available globally (for example, `'sem_type'`, `'args_type'`, `'atomic_type'`; `'cat_feat'`, `'str_feat'`, `'pred_feat'`).
2. ε-rules are removed from the grammar using function `'eliminate_empty_cats'`.
3. After grammar acquisition is complete, function `'separate_lex_ambiguity'` is invoked to decompose ambiguous lexical entries into several MRSs of length 1.
4. The grammar is then inverted using function `invert_grammar`.
    (a) Function `'append_str'` effects Step 1a of the Normalization algorithm and sets the *str* feature values in grammar rules.
    (b) Function `'create_lex_str'` performs a similar operation for lexicon entries, to the effect of Step 4b ibid.
    (c) Function `'normalize'` performs grammar normalization, using (self-explanatory) functions `'normalize_grammar_rules'` and `'normalize_lexicon_entries'`.
        i. This routine uses function `'chain_rule'` to obtain a boolean value which is **true** for *chain* rules and **false** otherwise (Step 1b).

ii. According to the classification results, functions `'norm_FI_rule'`, `'norm_AF_rule'` and `'norm_LEX_rule'` are invoked, to the effect of Steps 2, 3 and 4, respectively.

iii. The following auxiliary functions are used during normalization:

- Function `'set_non_sem_head_argument_flow'` sets the argument flow in all the body constituents of FI or AF rule, except the semantic head (Steps 2a and 3a).
- Function`'set_rule_head_argument_flow'` sets the argument flow in the head of FI rules. (Step 2a).
- Function `'convert_body_sem_head_args'` converts the body constituents of AF rules into arguments of the semantic head (Step 3a).
- Function `'get_sem_core'` returns a pointer to the semantic core of a given FS.
- A boolean function `'dp_reentrant'` checks whether the designated paths in two given feature structures (inside a specified MRS) are reentrant.
- Function `'restructure_rule_body'` restructures bodies of FI rules to match the order of arguments in the head logical form (Step 2a).
- Whenever a new rule is created (FI, AF or lexicon-derived), function `'add_rule'` is invoked to check if it shoulb be added to the grammar (this function is also used during grammar inversion when new inverted rules are created). The function adds the rule to the existing set of rules, if there is no rule more general than the new one. If there exists a rule more specific than the new one, the former is replaced by the latter. If a more general rule exists, the new rule is discarded.

(d) Function `'invert'` performs grammar inversion:

i. Function `'compute_rule_chains'` computes maximal chains of normalized rules, according to Step 1 of the Inversion algorithm.

A. Function `'grow_chains'` implements Step 1b and computes all the chains which terminate with the given FI rule. The function uses an auxiliary routine `'expand_chain'` which expands a given chain with an additional AF rule. If the syntactic category of the (single) body constituent of this AF rule is not preterminal, the corresponding link in the chain is marked accordingly. In such a case the chain is later decomposed at this point as well (i.e., an inverted rule is created from the part of the chain between this point and the FI rule terminating the chain).

B. Then all the created chains are analyzed. If a chain is terminated with a lexicon-derived FI rule, function `'flatten_inv_args'` is used to flatten its body (Step 1(d)i). The flattened body is subsequently restructured (Step 1(d)ii) with function `'restructure_rule_body'` (which was used during normalization to restructure bodies of the FI rules which are not lexicon-derived). If the LD rule has an "argument" syntactic category and was combined with other AF rules, a new chain is created which duplicates this rule (Step 1(d)iii). To this end an auxiliary boolean function `'arg_cat'` is used to determine whether a given category is an "argument" category.

ii. Function `'decompose_chains'` breaks down the chains into separate inverted rules (Step 2). An auxiliary function `'create_new_rule'` is used to build a new rule by extracting its head and body from the MRS representing the chain. New inverted rules are added to the inverted grammar using function `'add_rule'`.

(e) For the interpreter to operate correctly the grammar should be arranged so that unit rules (if any) precede any other rules. To this end boolean function `'unit_rule'` is used to determine whether a given rule is a unit rule.

5. Before the SKB is created by the compiler, the lexicon and the Connective Registry are processed to replace each feature structure with the value of its *sem* feature (cf. Section 4.1.3).

6. Finally, the lexicon and the Connective Registry are merged, and the SKB is created using function `'create_skb'`. This function uses an auxiliary function `'print_spec_fs'` which prints a given feature structure in ALE specification language. The SKB is recorded in a file to be used by the interpreter.

5.1.3 Functional breakdown of the generation cycle

Data structures

The main data structures used during the generation cycle are those used by the $\mathcal{A}$MALIA interpreter (see (Wintner, 1997, Chapter 3)); these data structures were not affected by the generation additions to the machine. In addition to that, the operations of SKB lookup and verbalization[1] perform unification and subsumption checks on MRSs in the same representation as used during compilation (cf. Section 5.1.2).

Main functions and data flow

1. The abstract machine is initialized using function `'initialize_machine'`, which builds the main data structures of the AM: the heap, the chart, general and special purpose registers, the trail and the code area.
2. Function `'new_program'` reads the symbols tables prepared by the compiler (`'read_sym_tables'`), and then loads the program supplied in the specified input file into the code area (`'load_program'`).
3. The query is supplied by the user in an additional file (besides the one that contains the program), encoded in ALE specification language. Function `'process_query'` decomposes the query and prepares chart initialization items. These items are then compiled and executed on the chart.
   (a) First, the query file is parsed using the same parser that inputs grammar specifications during compilation — function `'yyparse'`). The outcome of this process is a MRS whose single root contains a FS that represents the query.
   (b) The query is scanned for vertices which represent quantified variables, and those found are marked accordingly (see Step 0a of the procedure **flatten** in the Generation algorithm, Section 4.2). This step is performed by the function `'find_quant_vars'`.
   (c) The query is then flattened (`'flatten_query'`) into a sequence of semantic primitives (Step 1a of the Generation algorithm). The function creates new roots in the MRS that contains the query, with each root corresponding to some semantic primitive found during flattening. The mutual order of the roots corresponds to the postorder traversal of the query.
   (d) Function `'query_init_chart'` performs the SKB lookup for the components of the query, compiles the chart initialization items and executes them:
      i. For each component of the query (represented by a root in the host MRS), all SKB entries whose semantic core subsumes this component are collected. In the general case there may be more than one SKB entry per component. The semantic core of each such SKB entry is unified with this component (`'fs_unify'`) *in the context* of the entire entry. The unification results are collected in the form of a (new) MRS, where each root corresponds to some SKB entry as changed upon the unification. All the MRSs corresponding to the components of the query are assembled in an array (`'query_skb'`).
         - In the above scheme unification is performed each time between a component of the query and the semantic core of a SKB entry. Observe that function `'fs_unify'` is defined to unify two feature structures which reside in the same host MRS. To this end a dummy MRS is created which contains two nodes corresponding to the two unificands. This is achieved by function `'add_one_fs_to_mrs'` which adds (copies) a single feature structure of one MRS as a new root to another MRS.

[1]See Section 5.1.1.

ii. What remains to be done is to compile all the SKB entries which correspond to each component of the query, ans execute them to initialize the chart. This is realized by function `'compile_and_execute_query'`.

A. This is performed by the same functions which compile the lexicon for parsing. Function `'gen_lex_equations'` converts each SKB entry into a set of equations, and function `'gen_instructions'` generates the AM instructions.

B. Function `'add_program_instruction'` loads the newly created machine functions into the code area, and then function `'execute_function'` executes them.

C. After the query is executed, the program counter needs to be restored to the beginning of the program using function `'set_program_counter'`.

(e) This completes the *scanning* step of the chart generation (Step 1b of the Generation algorithm).

4. After initialization, the chart processing algorithm is invoked by function `'run_program'`, which executes the program created for the inverted grammar. This effects the process of chart generation by first creating *prediction* items for the grammar rules (Step 1c), and then performing the operations of *dot movement* and *completion* (Step 2).

5. Function `'print_final_results'` displays the generation results (if any).

   (a) Each complete edge which covers the entire input is retrieved from the chart, and the feature structure it defines is built from the heap representation using function `'heap_to_fs'`.

   (b) Function `'gen_result_to_words'` creates a string of words spanned by the *str* feature of the given FS:

      i. The value of the *str* feature of the result encodes a sequence of semantic primitives in the form of a tree-like structure. Function `'flatten_tree'` flattens this tree, and creates a linear list of semantic primitives (Step 3 of the Generation algorithm). To this end the tree nodes retrieved during flattening are organized as new roots of the host MRS.

      ii. For each element of the flattened list all the SKB entries are collected whose semantic core subsumes this element (the subsumption check is performed by function `'mrs_less_than'`). The words corresponding to the selected SKB entries are retrieved, and together they comprise the generated phrase (Step 3b). Whenever several SKB entries subsume a singe semantic primitive, the corresponding words are pooled together and delimited with braces in the output.

Chapter 6

Conclusions

The main purpose of this work was to apply the Abstract Machine approach to the problem of Natural Language Generation. Given the $\mathcal{A}$MALIA abstract machine for parsing unification grammars (Wintner, 1997), we enhanced it with generation capabilities.

To this end we defined a concept of chart generation, similar to the familiar chart parsing. This technique allows the generator to systematically consume parts of the given logical form, and build natural language phrases with corresponding meaning. Unlike previous works (Kay, 1996; Trujillo, 1997), we do not require flat semantics representation; instead, the input to the generator may comprise nested logical forms. In order to process such complex forms and allow their systematical decomposition into meaning primitives, we made use of an existing algorithm for grammar inversion (Samuelsson, 1995). The algorithm discerns the predicate-argument structure of logical forms, and rebuilds parsing grammars to reflect this structure, making them inherently suitable for generation. Since $\mathcal{A}$MALIA works with grammars encoded in a TFS-based formalism, we ported the grammar inversion algorithm (originally formulated for the DCG formalism) accordingly.

We then extended the control structures of $\mathcal{A}$MALIA to facilitate chart generation. It should be observed that the additions to $\mathcal{A}$MALIA were limited only to the control, while the instruction set of the machine remained unaffected. The unified $\mathcal{A}$MALIA performs parsing and generation in a largely uniform way, and the chart processing core is totally independent of the actual processing direction. Extending $\mathcal{A}$MALIA with the ability to perform generation resulted in an efficient bidirectional system for Natural Language Processing.

For a grammar to be invertible (and thus suitable for generation with $\mathcal{A}$MALIA) it must satisfy a number of requirements. Among the stricter ones is the compositionality criterion, which requires that in every grammar rule, the semantics of the head must be a predicate-argument structure of the semantics of (all and only) the elements in the body. There are natural language constructs (e.g., expletives) which do not satisfy the compositionality requirement, therefore a possible extension to this work would be to relax this condition. Another assumption which is a potential candidate for relaxation is that currently chain rules may have no more than one semantic head (and, analogously, that the non-chain rules may have only one argument carrier).

There are a number of other limitations inherent to the grammar inversion procedure, and further research on the topic may address these issues as well. For example, the way inverted rules are created from chains of normalized rules (by taking the head of the first and the body of the last rule in a chain) does not necessarily preserve *goals* (if any were specified for the original rules comprising the chain). This occurs if goals are associated with rules in the middle of the chain, since constituents of such rules are not necessarily explicitly represented in the resultant inverted rules.

Also, it is unclear in the meanwhile how to adapt the proposed technique to generate partial phrases rather than complete sentences. For instance, it is natural to expect the generator to produce the phrase "loves Mary" (or "to love Mary") given the input $\lambda x.\mathtt{love}(x,\mathtt{mary})$. Such cases are problematic, since the chart generation algorithm currently requires all the arguments of a predicate to be explicitly available, in order to generate from this predicate.

This work presents two grammars suitable for our generation algorithm: the running example grammar, and the larger one which incorporates the principles of Montague semantics. Both grammars are quite small, and are provided merely to justify the feasibility of the approach. Hence an immediate extension of this project would be to design larger, “real” grammars, covering substantial fragments of the natural language.

Appendix A

The running example grammar

This Appendix contains the running example grammar and demonstrates sample generation with this grammar. In what follows, Section A.1 presents the listing of the grammar. Section A.2 shows the normalized and the inverted versions of the grammar, and Section A.3 contains two generation examples.

A.1 Listing of the grammar

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  File: Running example grammar
%
%
%  Covers:
%
%  1) noun phrases
%  2) verb phrases (intransitive verbs)
%  3) modifiers - predicate adverbs
%
%  Parses:
%
%  1) John smokes
%  2) John smokes today
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%**********************  Type Hierarchy
%th

bot sub [sign, syn, syn_term, sem, args, list].
  sign sub [phrase].
    phrase sub [word] intro [syn:syn, sem:sem, args:args, str:list].
      word sub [].
  syn sub [] intro [cat:syn_term].
  syn_term sub [s, np, vp, advp].
    s sub [].
    np sub [].
    vp sub [vi].
      vi sub [].
    advp sub [].
  sem sub [const, funct].
    const sub [pn, adv].
      pn sub [john].
        john sub [].
      adv sub [today].
        today sub [].
    funct sub [predic, atomic, quant].
      predic sub [aux, verb].
        aux sub [mod].
          mod sub [].
        verb sub [v_intrans].
          v_intrans sub [smoke].
            smoke sub [].
      atomic sub[arg_1] intro [pred:sem].
        arg_1 sub [arg_2] intro [arg1:sem].
          arg_2 sub [] intro [arg2:sem].
      quant sub [l_bind] intro [var:sem].
        l_bind sub [] intro [rest:sem].
  args sub [] intro [larg:list].
  list sub [ne_list, e_list].
    ne_list sub [] intro [hd:bot, tl:list].
    e_list sub [].


%macros
%%%**********************  Macros
lex(Cat, Sem) macro
(word, syn:(syn, cat:Cat),
       sem:Sem).

cr(Sem) macro
(sem:Sem).

%grammar
%%%**********************  Grammmar Rules

o_2 rule
%sem_head_2

(phrase, syn:(syn, cat:s),
         sem:R6)
===>
cat>
(phrase, syn:(syn, cat:np),
         sem:(R5, sem)),
cat>            % head
(phrase, syn:(syn, cat:vp),
         sem:(l_bind, (var:R5, rest:(R6, funct)))).

o_3 rule
%sem_head_1
```

```
(phrase, syn:(syn, cat:vp),
         sem:(l_bind, (var:R5,
                        rest:(atomic, (pred:mod,
                                       arg1:R6,
                                       arg2:R7)))))
===>
cat>            % head
(phrase, syn:(syn, cat:vp),
         sem:(l_bind, (var:R5, rest:(R6, funct)))),
cat>
(phrase, syn:(syn, cat:advp),
         sem:(R7, sem)).

%lexicon
%%%**********************  Lexical Entries

% o_8: "John"
john --->
( @ lex(np, john) ).

% o_11: "smokes"
% Lambda x. smoke(x)
smokes --->
( @ lex(vi,
        (l_bind, (var:R5,
                   rest:(atomic, (pred:smoke,
                                  arg1:R5))))) ).

% o_14: "today"
today --->
( @ lex(advp, today) ).

%connective_registry
%%%**********************  Connective Registry Entries

% modifier
)-->
( @ cr((l_bind, (var:R5,
                 rest:(R1, atomic, (pred:mod,
                                    arg1:  (R6, funct),
                                    arg2:  R7))))) ).
```

A.2 Normalization and inversion of the running example grammar

A.2.1 Normalized sample grammar

```
***** Printing normalized rules ... *****

***** Printing FI rules ... *****
```

```
***** Normalized FI rule - image of rule 1 ***** (rule N_3)

[55]phrase(
  syn:[56]syn(
    cat:[6]vp),
  sem:[57]l_bind(
    var:[92]sem,
    rest:[50]arg_2(
      pred:[41]mod,
      arg1:[93]funct,
      arg2:[132]sem)),
  args:[58]args(
    larg:[138]list),
  str:[137]ne_list(
    hd:[102]list,
    tl:[136]ne_list(
      hd:[134]list,
      tl:[135]e_list)))
===>
[98]phrase(
  syn:[99]syn(
    cat:[66]vp),
  sem:[100]l_bind(
    var:[92],
    rest:[93]),
  args:[101]args(
    larg:[138]),
  str:[102]),
[130]phrase(
  syn:[131]syn(
    cat:[109]advp),
  sem:[132],
  args:[133]args(
    larg:[122]e_list),
  str:[134]),
[57]
> This rule has no goals.

***** Printing lexicon-derived FI rules ... *****

***** Normalized lexicon-derived FI rule -
      image of lexicon-entry 0              ***** (rule N_8)

[0]word(
  syn:[1]syn(
    cat:[2]np),
  sem:[3]pn,
  args:[4]args(
    larg:[5]list),
  str:[8]ne_list(
    hd:[3],
    tl:[7]e_list))
===>
[4],
[3]
> This rule has no goals.

***** Normalized lexicon-derived FI rule -
      image of lexicon-entry 1              ***** (rule N_11)

[0]word(
  syn:[1]syn(
    cat:[2]vi),
  sem:[3]l_bind(
    var:[4]sem,
    rest:[5]arg_1(
      pred:[6]v_intrans,
      arg1:[4])),
  args:[7]args(
    larg:[8]list),
  str:[11]ne_list(
    hd:[5],
    tl:[10]e_list))
===>
[7],
[3]
> This rule has no goals.

***** Normalized lexicon-derived FI rule -
      image of lexicon-entry 2              ***** (rule N_14)

[0]word(
  syn:[1]syn(
    cat:[2]advp),
  sem:[3]adv,
  args:[4]args(
    larg:[5]list),
  str:[8]ne_list(
    hd:[3],
    tl:[7]e_list))
===>
[4],
[3]
> This rule has no goals.

***** Printing AF rules ... *****

***** Normalized AF rule - image of rule 0 ***** (rule N_2)

[26]phrase(
  syn:[27]syn(
    cat:[6]s),
  sem:[97]funct,
  args:[29]args(
    larg:[19]list),
  str:[109]ne_list(
    hd:[63]list,
    tl:[108]ne_list(
      hd:[106]list,
      tl:[107]e_list)))
===>
[102]phrase(
  syn:[103]syn(
    cat:[70]vp),
  sem:[104]l_bind(
    var:[96]sem,
    rest:[97]),
  args:[105]args(
    larg:[110]ne_list(
      hd:[59]phrase(
        syn:[60]syn(
          cat:[37]np),
        sem:[96],
        args:[62]args(
          larg:[50]e_list),
        str:[63]),
      tl:[19])),
  str:[106])
> This rule has no goals.

***** All normalized rule printed. *****
```

A.2.2 Inverted sample grammar

```
***** Printing inverted rules ... *****

***** Inverted rule - terminated with FI rule 0 ***** (rule I_2)

[0]phrase(
  syn:[1]syn(
    cat:[2]s),
  sem:[3]arg_2(
    pred:[4]mod,
    arg1:[5]funct,
    arg2:[6]sem),
  args:[7]args(
    larg:[8]e_list),
  str:[9]ne_list(
    hd:[10]list,
    tl:[11]ne_list(
      hd:[12]ne_list(
        hd:[13]list,
        tl:[14]ne_list(
          hd:[15]list,
          tl:[16]e_list)),
      tl:[17]e_list)))
===>
[18]phrase(
  syn:[19]syn(
    cat:[20]vp),
  sem:[21]l_bind(
    var:[22]sem,
    rest:[5]),
  args:[23]args(
    larg:[24]ne_list(
      hd:[25]phrase(
        syn:[26]syn(
          cat:[27]np),
        sem:[22],
        args:[28]args(
          larg:[29]e_list),
        str:[10]),
      tl:[8])),
  str:[13]),
[30]phrase(
  syn:[31]syn(
    cat:[32]advp),
  sem:[6],
  args:[33]args(
    larg:[34]e_list),
  str:[15]),
[35]l_bind(
  var:[22],
  rest:[3])
> This rule has no goals.

***** Inverted rule - terminated with FI rule 0 ***** (rule I_4)

[0]phrase(
  syn:[1]syn(
    cat:[2]vp),
  sem:[3]l_bind(
    var:[4]sem,
    rest:[5]arg_2(
      pred:[6]mod,
      arg1:[7]funct,
      arg2:[8]sem)),
  args:[9]args(
    larg:[10]ne_list(
      hd:[11]phrase(
        syn:[12]syn(
          cat:[13]np),
        sem:[4],
        args:[14]args(
          larg:[15]e_list),
        str:[16]list),
      tl:[17]e_list)),
  str:[18]ne_list(
    hd:[19]list,
    tl:[20]ne_list(
      hd:[21]list,
      tl:[22]e_list)))
===>
[23]phrase(
  syn:[24]syn(
    cat:[25]vp),
  sem:[26]l_bind(
    var:[4],
    rest:[7]),
  args:[27]args(
    larg:[10]),
  str:[19]),
[28]phrase(
  syn:[29]syn(
    cat:[30]advp),
  sem:[8],
  args:[31]args(
    larg:[32]e_list),
  str:[21]),
[3]
> This rule has no goals.

***** Inverted rule -
      terminated with lexicon-derived FI rule 1 ***** (rule I_11)

[0]word(
  syn:[1]syn(
    cat:[2]np),
  sem:[3]pn,
  args:[4]args(
    larg:[5]e_list),
  str:[8]ne_list(
    hd:[3],
    tl:[7]e_list))
===>
[3]
> This rule has no goals.

***** Inverted rule -
      terminated with lexicon-derived FI rule 2 ***** (rule I_6)

[0]phrase(
  syn:[1]syn(
    cat:[2]s),
  sem:[3]arg_1(
    pred:[4]v_intrans,
    arg1:[5]sem),
  args:[6]args(
    larg:[7]e_list),
  str:[8]ne_list(
    hd:[9]list,
    tl:[10]ne_list(
      hd:[11]ne_list(
        hd:[3],
        tl:[12]e_list),
      tl:[13]e_list)))
===>
[14]phrase(
  syn:[15]syn(
    cat:[16]np),
  sem:[5],
  args:[17]args(
    larg:[18]e_list),
  str:[9]),
[19]l_bind(
  var:[5],
  rest:[3])
> This rule has no goals.

***** Inverted rule -
      terminated with lexicon-derived FI rule 2 ***** (rule I_7)

[0]word(
  syn:[1]syn(
    cat:[2]vi),
  sem:[3]l_bind(
    var:[4]sem,
    rest:[5]arg_1(
      pred:[6]v_intrans,
      arg1:[4])),
  args:[7]args(
    larg:[8]ne_list(
      hd:[9]phrase(
        syn:[10]syn(
          cat:[11]np),
        sem:[4],
        args:[12]args(
          larg:[13]e_list),
        str:[14]list),
      tl:[15]e_list)),
  str:[16]ne_list(
    hd:[5],
    tl:[17]e_list))
===>
[9],
[3]
> This rule has no goals.

***** Inverted rule -
      terminated with lexicon-derived FI rule 3 ***** (rule I_14)
```

```
[0]word(
  syn:[1]syn(
    cat:[2]advp),
  sem:[3]adv,
  args:[4]args(
    larg:[5]e_list),
  str:[8]ne_list(
    hd:[3],
    tl:[7]e_list))
===>
[3]
> This rule has no goals.

***** All inverted rule printed. *****
```

A.2.3 Connective Registry

```
***** Printing Connective Registry ... *****

***** CR entry ***** (mod)

[39]l_bind(
  var:[32]sem,
  rest:[33]arg_2(
    pred:[24]mod,
    arg1:[22]funct,
    arg2:[16]sem))

***** All CR entries printed. *****
```

A.2.4 Semantics Knowledge Base (SKB)

```
%skb

|-->
(john,0)
(R3,john).

|-->
(smokes,0)
(R3,l_bind,(
  var:(R4,sem),
  rest:(R5,arg_1,(
    pred:(R6,smoke),
    arg1:(R4))))).

|-->
(today,0)
(R3,today).

|-->
(_,0)
```

```
(R39,l_bind,(
  var:(R32,sem),
  rest:(R33,arg_2,(
    pred:(R24,mod),
    arg1:(R22,funct),
    arg2:(R16,sem))))).
```

A.3 Generation examples

A.3.1 Example 1

Query

```
%query

%%%% john smokes
%%%% smoke(john)
>>>>
(phrase,(
  syn:(syn,(
        cat:s)),
  sem:(atomic,(
        pred:smoke,
        arg1:(R1, john))))).
```

Generation results

```
Results:

Result number 1:
[0]phrase(
  syn:[1]syn(
    cat:[2]s),
  sem:[3]arg_1(
    pred:[4]smoke,
    arg1:[5]john),
  args:[6]args(
    larg:[7]e_list),
  str:[8]ne_list(
    hd:[9]ne_list(
      hd:[5],
      tl:[10]e_list),
    tl:[11]ne_list(
      hd:[12]ne_list(
        hd:[3],
        tl:[13]e_list),
      tl:[14]e_list)))

john smokes

Result number 2:
[0]word(
  syn:[1]syn(
    cat:[2]vi),
  sem:[3]l_bind(
    var:[4]john,
    rest:[5]arg_1(
      pred:[6]smoke,
      arg1:[4])),
  args:[7]args(
    larg:[8]ne_list(
      hd:[9]word(
        syn:[10]syn(
          cat:[11]np),
        sem:[4],
        args:[12]args(
          larg:[13]e_list),
        str:[14]ne_list(
          hd:[4],
          tl:[15]e_list)),
      tl:[16]e_list)),
  str:[17]ne_list(
    hd:[5],
    tl:[18]e_list))

smokes
```

A.3.2 Example 2

Query

```
%query

%%%% john smokes today
%%%% mod(smoke(john),today)
>>>>
(phrase,(
  syn:(syn,(
        cat:s)),
  sem:(atomic,(
        pred:mod,
        arg1:(R5, atomic,(
              pred:smoke,
              arg1:(R1,john))),
        arg2:(R6, today))))).
```

Generation results

```
Results:

Result number 1:
[0]phrase(
  syn:[1]syn(
    cat:[2]s),
  sem:[3]arg_2(
    pred:[4]mod,
    arg1:[5]arg_1(
      pred:[6]smoke,
      arg1:[7]john),
    arg2:[8]today),
  args:[9]args(
    larg:[10]e_list),
  str:[11]ne_list(
    hd:[12]ne_list(
      hd:[7],
      tl:[13]e_list),
    tl:[14]ne_list(
      hd:[15]ne_list(
        hd:[16]ne_list(
          hd:[5],
          tl:[17]e_list),
        tl:[18]ne_list(
          hd:[19]ne_list(
            hd:[8],
            tl:[20]e_list),
          tl:[21]e_list)),
      tl:[22]e_list)))

john smokes today

Result number 2:
[0]phrase(
  syn:[1]syn(
    cat:[2]vp),
  sem:[3]l_bind(
    var:[4]john,
    rest:[5]arg_2(
      pred:[6]mod,
      arg1:[7]arg_1(
        pred:[8]smoke,
        arg1:[4]),
      arg2:[9]today)),
  args:[10]args(
    larg:[11]ne_list(
      hd:[12]word(
        syn:[13]syn(
          cat:[14]np),
        sem:[4],
        args:[15]args(
          larg:[16]e_list),
        str:[17]ne_list(
          hd:[4],
          tl:[18]e_list)),
      tl:[19]e_list)),
  str:[20]ne_list(
    hd:[21]ne_list(
      hd:[7],
      tl:[22]e_list),
    tl:[23]ne_list(
      hd:[24]ne_list(
        hd:[9],
        tl:[25]e_list),
      tl:[26]e_list)))

smokes today
```

Appendix B

The Montague sample grammar

This Appendix presents an additional sample grammar for a fragment of English which incorporates Montague semantics (Gamut, 1991, Chapter 5). In what follows, Section B.1 contains the grammar per se, while Section B.2 lists a number of sample queries.

An important feature of Montague grammar is the compositionality of meaning and syntax. The grammar consists of ***pairs of rules***: the *syntactic* rules analyze the phrase structure, and the *translation* rules utilize the former to assign NL expressions a meaning in a logical language. The compositionality of semantics is achieved through the extensive use of λ-calculus. Since we work in a "rich", unification-based formalism, the rules of our grammar embody both the syntactic and translation informaion of original Montague rules.

We model our discussion of the grammar on (Gamut, 1991, Chapter 5), therefore the numbering of rules coincides with that found ibid. Our sample grammar covers the following language phenomena:

- noun phrases,
- verb phrases (intransitive and transitive verbs),
- noun modification with prenominal adjectives (e.g., "good man"),
- verb modification with predicate adverbs, (e.g., "talks slowly"),
- sentence-modifying adverbs (e.g., "Necessarily John smokes"),
- definite and indefinite articles (e.g., "a, the"),
- determiners (e.g., "every, one"),
- conjunction and disjunction ("and, or"),
- infinitival complements (e.g., "John wants to smoke"),
- passive verbs (e.g., "Mary is loved by John"),
- relative clauses (e.g., "Every man who smokes suffers").

Let us briefly review the effect grammar rules:

1. Rule T_2 combines noun phrases with intransitive verbs into sentences, e.g., "John smokes" or "Every man smokes". The rule applies the meaning of a noun phrase to that of an intransitive verb, and forms the meaning of a sentence.
2. Rule T_3' categorimatically treats determiners and articles, and produces noun phrases from common nouns, e.g., "A man" or "Every woman". The semantics of determiners uses the universal and existential quantifiers. For instance, the meaning of the word "every" is given by $\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$. When "every" is combined with the word "woman", the meaning of the former is applied to that of the latter ($\lambda y.\texttt{woman}(y)$), resulting (after a double β-reduction) in $\lambda Q.\forall x(\texttt{woman}(x) \rightarrow Q(x))$.

3. Having considered Montagovian semantics of common nouns and verbs, let us discuss the semantics of proper nouns. Pure Montague approach defines the meaning of "John" as $\lambda P.P(\texttt{john})$, where P is a predicate which can be instantiated to the meaning of a verb (e.g., when rule T_2 is applied). This semantics could be encoded with typed feature structures as follows:

$$fs_j = \left[\begin{array}{ll} \textbf{word} & \\ \mathit{syn}: & \left[\begin{array}{ll} \textbf{syn} & \\ \mathit{cat}: & \left[\textbf{np}\right] \end{array}\right] \\ \mathit{sem}: & \left[\begin{array}{ll} \lambda\textbf{-bind} & \\ \mathit{var}: & \boxed{5}\left[\textbf{sem}\right] \\ \underline{\mathit{rest}}: & \boxed{1}\left[\begin{array}{ll} \textbf{arg\_1} & \\ \mathit{pred}: & \boxed{5} \\ \mathit{arg1}: & \left[\textbf{john}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

But then we encounter a problem with our generation scheme, as the β-reduction "built into" the rule invalidates the predicate-argument structure, which is one of the basic assumptions of the generation algorithm. Rule $T\_2 = NP\ V_i \Rightarrow S$ which combines this noun phrase with an intransitive verb is a chain rule, which (as mentioned above) applies the meaning of the former to that of the latter. Nevertheless, after two implicit β-reductions take place, the resulting expression actually applies the meaning of the verb to that of "John".

The normalized version of rule T_2 is $T\_2_N = NP(V_i) \Rightarrow S$. During inversion, this rule is combined with the (lexicon-derived) Functor-Introducing rule $A\ [NP] \Rightarrow NP(A)$, thus resulting in the inverted rule $T\_2_I = V_i\ [NP] \Rightarrow S$. For this rule to be applicable in bottom-up generation, the semantics of NP should be recognized as the predicate, and that of V_i as as argument, which is clearly not the case as we have seen above. Currently, the postorder flattening procedure would place the item for NP before that of V_i, rendering rule T_2_I useless.

There is also an additional problem with the above representation for the semantics of "John". One of the key concepts of our generation algorithm is its ability to map semantic primitives (or, more precisely, their *semantic cores*) into natural language constructs. The semantic core of fs_j (denoted by $\boxed{1}$) contains an unbound variable in the predicate position. Therefore it unifies with (the semantic core of) numerous lexicon entries (for example, all those representing intransitive verbs as unary predicates), and not only with that for "John".

To overcome the above problems we resort to the following solution. We define the semantics of proper names similarly to that of quantified terms, i.e., the meaning of "John" becomes similar to that of "every man": $\lambda P.\exists x.((x = \texttt{john})\ \wedge\ P(x))$.

$$fs'_j = \left[\begin{array}{ll} \textbf{word} & \\ \mathit{syn}: & \left[\begin{array}{ll} \textbf{syn} & \\ \mathit{cat}: & \left[\textbf{np}\right] \end{array}\right] \\ \mathit{sem}: & \left[\begin{array}{ll} \lambda\textbf{-bind} & \\ \mathit{var}: & \boxed{5}\left[\textbf{sem}\right] \\ \underline{\mathit{rest}}: & \boxed{1}\left[\begin{array}{ll} \textbf{arg\_1} & \\ \mathit{pred}: & \left[\begin{array}{ll} \exists\textbf{-quant} & \\ \mathit{var}: & \textsf{[2]}\left[\textbf{sem}\right] \\ \mathit{scope}: & \left[\begin{array}{ll} \textbf{and} & \\ \mathit{wff1}: & \left[\begin{array}{ll} \textbf{equal} & \\ \mathit{wff1}: & \textsf{[2]} \\ \mathit{wff2}: & \textsf{[john]} \end{array}\right] \\ \mathit{wff2}: & \boxed{5}\left[\begin{array}{ll} \textbf{atomic} & \\ \mathit{param1}: & \textsf{[2]} \end{array}\right] \end{array}\right] \end{array}\right] \\ \mathit{arg1}: & \boxed{5} \end{array}\right] \end{array}\right] \\ \mathit{str}: & \boxed{1} \end{array}\right]$$

This way the meaning of "John" is represented with a predicate ($\boxed{1}$) that operates on the semantics of a verb ($\boxed{5}$).

4. Conventional Montague approach obtains the meaning of the phrase "John loves Mary" (`love(john, mary)`) in three steps. First it applies the meaning of "loves" ($\lambda y.\lambda x.\texttt{love}(x,y)$) to that of "Mary"

$(\lambda Q.Q(\mathtt{mary}))$, to obtain $\lambda x.\mathtt{love}(x, \lambda Q.Q(\mathtt{mary}))$. Then the meaning of "John" $(\lambda P.P(\mathtt{john}))$ is applied to the outcome of the previous operation, resulting (after two β-reductions) in $\mathtt{love}(\mathtt{john}, \lambda Q.Q(\mathtt{mary}))$. Finally, Notational Convention NC2 (see (Gamut, 1991, p. 176)) reduces the last expression to `love(john, mary)`.

Our approach is similar, except it doesn't perform the last step since it's not clear how to implement Meaning Postulates and Notational Conventions in a TFS-based formalism. Rule T_7 builds intransitive verbs from transitive verbs and noun phrases. For example, the meaning of "loves Mary" is obtained by applying the meaning of "loves" $(\lambda w.\lambda z.\mathtt{love}(z, w))$ to that of "Mary" $(\lambda Q.\exists y.((y = \mathtt{mary}) \wedge Q(y))$; cf. the previous item on the semantics of proper nouns). This results in the expression $\lambda z.\mathtt{love}(z, \lambda Q.\exists y.((y = \mathtt{mary}) \wedge Q(y)))$. The semantics of "John" $\lambda P.\exists x.((x = \mathtt{john}) \wedge P(x))$ is then applied to the last expression, yielding $\exists x.((x = \mathtt{john}) \wedge \mathtt{love}(x, \lambda Q.\exists y.((y = \mathtt{mary}) \wedge Q(y))))$.

If it were possible to implement Meaning Postulates and Notational Conventions in a TFS-based formalism, Meaning Postulate MP2 (see (Gamut, 1991, p. 175)) and then Notational Convention NC2 would reduce the last expression to $\exists x.((x = \mathtt{john}) \wedge \exists y.((y = \mathtt{mary}) \wedge \mathtt{love}(x, y)))$.

5. Rules T_2_pas and T_7_pas handle the formation of passive sentences. These rules are similar to their "active" counterparts (T_2 and T_7, respectively), only the surface order of the body constituents corresponds to the passive voice.

   For example, the sentence "Mary is loved by John" can be derived using these two rules. Observe that the semantics of this sentence is given by the logical form $\exists x.((x = \mathtt{john}) \wedge \mathtt{love}(x, \lambda Q.\exists y.((y = \mathtt{mary}) \wedge Q(y))))$, and is identical to that of the sentence "John loves Mary". This example demonstrates that AMALIA is capable of generating paraphrases: given this meaning as input, AMALIA would generate both the active and the passive sentences.

6. Rule T_9_10 handles conjunction and disjunction of sentences, (e.g., "John smokes and/or Mary talks"). Unification-based formalism allows us to abstract over specific connective values, and thus to combine two Montague rules $T9$ and $T10$ (which handle conjunction and disjunction, respectively) into one.

7. Rule T_11_12 similarly addresses conjunction and disjunction of intransitive verbs.

8. Rule T_16 handles the construction of sentences with an infinitival complement. For example, it can be used to derive the sentence "John wants to smoke" with the meaning $\exists x.((x = \mathtt{john}) \wedge \mathtt{want}(x, \mathtt{smoke}(x)))$. Observe that both the main verb ("wants") and the complement verb ("to smoke") share the same subject (the respective semantic primitives in the logical form are reentrant on x).

9. Rule T_17 modifies common nouns with prenominal adjectives (cf. Example 4 in Section 3.2.4).

10. Rule T_18 handles the formation of relative clauses. Thus, a common noun ("man") may be combined with a restrictive relative clause ("who smokes") to form a "complex" common noun expressing a complex property: to be a man and to smoke. For example, this rule may be used to create the sentence "Every man who smokes suffers" with the semantics $\forall x.((\mathtt{man}(x) \wedge \mathtt{smoke}(x)) \rightarrow \mathtt{suffer}(x))$.

    For another example, consider the sentence "Every good man who talks loves Mary", whose meaning is given by the logical form $\forall x.(((\mathtt{good}(\mathtt{man}))(x) \wedge \mathtt{talk}(x)) \rightarrow \mathtt{love}(x, \lambda Q.\exists y.((y = \mathtt{mary}) \wedge Q(y))))$. Because of the implementation of passive voice (see item 5 above), generation from this logical form also produces the passive sentence "Mary is loved by every good man who talks".

11. Rule T_19 modifies intransitive verbs with predicate adverbs. Similarly to infinitival complements, predicate adverbs are regarded semantically as second-order functions from properties of individuals to sets of individuals.

    Now let us consider an example of an ambiguous sentence, which is assigned by our grammar two different readings (with different semantics). The example is based on the modification of verbs with infinitival complements and predicate adverbs. Consider the sentence "John wants to leave urgently". Depending on the order in which rules T_16 and T_19 are applied, the following two readings may result:

(a) the adverb "urgently" modifies the main verb "wants", yielding the semantics $\exists x.((x = \texttt{john}) \land (\texttt{urgently}(\texttt{want}))(x, \texttt{leave}(x)))$;

(b) the adverb "urgently" modifies the complement verb "to leave", yielding the semantics $\exists x.((x = \texttt{john}) \land \texttt{want}(x, (\texttt{urgently}(\texttt{leave}))(x)))$.

It should be observed that $\mathcal{A}$MALIA produces the same sentence when generating from either semantics.

12. Finally, rule T_20 handles the case of sentence-modifying adverbs, by applying the semantics of an adverb to that of a sentence.

B.1 Montague grammar

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  File: Montague grammar
%
%  Covers:
%
%  1) noun phrases
%  2) verb phrases (intransitive and transitive verbs)
%  3) modifiers - prenominal adjectives and predicate adverbs
%  4) sentence-modifying adverbs
%  5) articles - a, the
%  6) determiner expressions - every, one
%  7) conjunction, disjunction
%  8) infinitival complements
%  9) passive verbs
%  10) relative clauses
%
%  Parses:
%  1) John smokes, John loves Mary passionately
%  2) Every good man smokes slowly, A man smokes
%  3) Every man loves Mary passionately
%  4) John smokes and/or Mary talks
%  5) John smokes and/or talks
%  6) Necessarily John smokes
%  7) John wants to smoke
%  8) John wants to leave urgently (2 parses)
%  9) Mary is loved by John (same semantics as "John loves Mary")
%  10) Every man who smokes suffers
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%**********************  Type Hierarchy
%th

bot sub [sign, syn, syn_term, sem, args, list, form].
  sign sub [phrase].
    phrase sub [word] intro [syn:syn, sem:sem, args:args, str:list].
      word sub [].
  syn sub [] intro [cat:syn_term].
  syn_term sub [s, np, vp, det, cn, mod, conj, reltvzr].
  s sub [].
  np sub [].
  vp sub [vi, vt, vic].
    vi sub [].
    vt sub [].
    vic sub [].
  det sub [].
  cn sub [].
  mod sub [adj, pred_adv, sent_adv].
    adj sub [].
    pred_adv sub [].
    sent_adv sub [].
  conj sub [and_conj, or_conj].
    and_conj sub [].
    or_conj sub [].
  reltvzr sub [].
sem sub [const, funct].
  const sub [pn].
    pn sub [john, mary].
      john sub [].
      mary sub [].
  funct sub [predic, atomic, quant, bool].
    predic sub [noun, verb, modifier, relativizer].
      noun sub [man, woman].
        man sub [].
        woman sub [].
      verb sub [v_intrans, v_trans, v_inf_comp].
        v_intrans sub [smoke, talk, suffer, leave].
          smoke sub [].
          talk sub [].
          suffer sub [].
          leave sub [].
        v_trans sub [love].
          love sub [].
        v_inf_comp sub [want].
          want sub [].
      modifier sub [adjective, adverb].
        adjective sub [good].
          good sub [].
        adverb sub [slowly, passionately, urgently, necessarily].
          slowly sub [].
          passionately sub [].
          urgently sub [].
          necessarily sub [].
      relativizer sub [who].
        who sub [].
    atomic sub[arg_1, param_1] intro [pred:sem, form:form].
      arg_1 sub [arg_2, atomic_1_1] intro [arg1:sem].
        arg_2 sub [arg_3, atomic_2_1] intro [arg2:sem].
          arg_3 sub [atomic_3_1] intro [arg3:sem].
            atomic_3_1 sub [atomic_3_2].
              atomic_3_2 sub [atomic_3_3].
```

```
                  atomic_3_3 sub [].
            atomic_2_1 sub [atomic_2_2, atomic_3_1].
              atomic_2_2 sub [atomic_3_2, atomic_2_3].
                atomic_2_3 sub [atomic_3_3].
          atomic_1_1 sub [atomic_1_2, atomic_2_1].
            atomic_1_2 sub [atomic_1_3, atomic_2_2].
              atomic_1_3 sub [atomic_2_3].
        param_1 sub [param_2, atomic_1_1] intro [param1:sem].
          param_2 sub [param_3, atomic_1_2] intro [param2:sem].
            param_3 sub [atomic_1_3] intro [param3:sem].
      quant sub [l_bind, gen_quant] intro [var:sem].
        l_bind sub [] intro [rest:sem].
        gen_quant sub [a_quant, e_quant] intro [scope:sem].
          a_quant sub [].
          e_quant sub [].
      bool sub [equal, if, iff, and_or, rel_and] intro [wff1:sem, wff2:sem].
        equal sub [].
        if sub [].
        iff sub [].
        and_or sub [and, or].
          and sub [].
          or sub [].
        rel_and sub [].
  args sub [] intro [larg:list].
  list sub [ne_list, e_list].
    ne_list sub [] intro [hd:bot, tl:list].
    e_list sub [].
  form sub [vform].
    vform sub [fin_inf, pas].
      fin_inf sub [fin, inf].
        fin sub [].
        inf sub [].
      pas sub [].

%macros
%%%**********************  Macros
lex(Cat, Sem) macro
(word, syn:(syn, cat:Cat),
       sem:Sem).

cr(Sem) macro
(sem:Sem).

%grammar
%%%**********************  Grammar Rules

t_2 rule
%sem_head_1

(phrase, syn:(syn, cat:s),
         sem:R6)
===>
```

```
cat>            % head
(phrase, syn:(syn, cat:np),
         sem:(l_bind, (var:R5, rest:R6))),
cat>
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, (var:R7, rest:(R5, atomic, form:fin)))).

t_2_pas rule
%sem_head_2

(phrase, syn:(syn, cat:s),
         sem:R6)
===>
cat>
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, (var:R7, rest:(R5, atomic, form:pas)))),
cat>            % head
(phrase, syn:(syn, cat:np),
         sem:(l_bind, (var:R5, rest:R6))).

t_3_prime rule
%sem_head_1

(phrase, syn:(syn, cat:np),
         sem:R6)
===>
cat>            % head
(phrase, syn:(syn, cat:det),
         sem:(l_bind, (var:R5, rest:R6))),
cat>
(phrase, syn:(syn, cat:cn),
         sem:(l_bind, (var:R7, rest:R5))).

t_7 rule
%sem_head_1

(phrase, syn:(syn, cat:vi),
         sem:(R6, l_bind, rest:(atomic, form:fin)))
===>
cat>            % head
(phrase, syn:(syn, cat:vt),
         sem:(l_bind, (var:R5, rest:R6))),
cat>
(phrase, syn:(syn, cat:np),
         sem:R5).

t_7_pas rule
%sem_head_2

(phrase, syn:(syn, cat:vi),
         sem:(R6, l_bind, rest:(atomic, form:pas)))
===>
```

```
cat>
(phrase, syn:(syn, cat:np),
         sem:R5),
cat>           % head
(phrase, syn:(syn, cat:vt),
         sem:(l_bind, (var:R5, rest:R6))).

t_9_10 rule
%sem_head_2

(phrase, syn:(syn, cat:s),
         sem:R1)
===>
cat>
(phrase, syn:(syn, cat:s),
         sem:R5),
cat>           % head
(word, syn:(syn, cat:conj),
       sem:(R1, atomic,
            (pred:(and_or, wff1:R5, wff2:R6)))),
cat>
(phrase, syn:(syn, cat:s),
         sem:R6).

t_11_12 rule
%sem_head_2

(phrase, syn:(syn, cat:vi),
         sem:(l_bind, var:R2, rest:R1))
===>
cat>
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, var:R2, rest:(R5, form:Vform))),
cat>           % head
(word, syn:(syn, cat:conj),
       sem:(R1, atomic,
            (pred:(and_or, wff1:R5, wff2:R6),
             form:Vform, param1:R2))),
cat>
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, (var:R2, rest:(R6, form:Vform)))).

t_16 rule
%sem_head_1

(phrase, syn:(syn, cat:vi),
         sem:R6)
===>
cat>           % head
(phrase, syn:(syn, cat:vic),
         sem:(l_bind, var:R5, rest:R6)),
cat>
```

```
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, rest:R5)).

t_17 rule
%sem_head_1

(phrase, syn:(syn, cat:cn),
         sem:R1)
===>
cat>            % head
(phrase, syn:(syn, cat:adj),
         sem:(l_bind, (var:R5, rest:(R1, l_bind, var:R6)))),
cat>
(phrase, syn:(syn, cat:cn),
         sem:(l_bind, (var:R6, rest:R5))).

t_18 rule
%sem_head_2

(phrase, syn:(syn, cat:cn),
         sem:(l_bind, var:R2, rest:R1))
===>
cat>
(phrase, syn:(syn, cat:cn),
         sem:(l_bind, var:R2, rest:R5)),
cat>            % head
(word, syn:(syn, cat:reltvzr),
       sem:(R1, atomic,
            (pred:(rel_and, wff1:R5, wff2:R6),
             param1:R2))),
cat>
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, (var:R2, rest:(R6, form:fin)))).

t_19 rule
%sem_head_2

(phrase, syn:(syn, cat:vi),
         sem:R1)
===>
cat>            %head
(phrase, syn:(syn, cat:vi),
         sem:(l_bind, (var:R6, rest:(R5, form:(Vform, fin_inf))))),
cat>
(phrase, syn:(syn, cat:pred_adv),
         sem:(l_bind, var:R5, rest:(R1, l_bind, var:R6,
                                    rest:(form:Vform)))).

t_20 rule
%sem_head_1

(phrase, syn:(syn, cat:s),
         sem:R1)
===>
cat>            %head
(phrase, syn:(syn, cat:sent_adv),
         sem:(l_bind, (var:R5, rest:R1))),
cat>
(phrase, syn:(syn, cat:s),
         sem:R5).

%lexicon
%%%**********************  Lexical Entries

% lex
% Lambda X. Lambda Y. Forall x. ( X(x) --> Y(x) )
every --->
( @ lex(det,
        (l_bind,
         (var:R5,
          rest:(l_bind,
                (var:R6,
                 rest:(atomic,
                       (pred:(a_quant,
                              (var:R2,
                               scope:(if,
                                      wff1:(R5, atomic, param1:R2),
                                      wff2:(R6, atomic, param1:R2))))),
                        arg1:R5,
                        arg2:R6))))))).

% lex
% Lambda X. Lambda Y. Exists x. ( Forall y. (X(y) <--> x=y) & Y(x) )
the --->
( @ lex(det,
       (l_bind,
        (var:R5,
         rest:(l_bind,
               (var:R6,
                rest:(atomic,
                      (pred:(e_quant,
                             (var:R2,
                              scope:(and,
                                     wff1:(a_quant,
                                           (var:R3,
                                            scope:(iff,
                                              wff1:(R5, atomic, param1:R3),
                                              wff2:(equal,
                                                   wff1:R2, wff2:R3)))),
                                     wff2:(R6, atomic, param1:R2))))),
                       arg1:R5,
                       arg2:R6))))))).

% lex
% Lambda X. Lambda Y. Exists x. ( X(x) & Y(x) )
a --->
( @ lex(det,
       (l_bind,
        (var:R5,
         rest:(l_bind,
               (var:R6,
                rest:(atomic,
                      (pred:(e_quant,
                             (var:R2,
                              scope:(and,
                                     wff1:(R5, atomic, param1:R2),
                                     wff2:(R6, atomic, param1:R2))))),
                       arg1:R5,
                       arg2:R6))))))).

% lex
% Lambda X. Lambda Y. Exists x. Forall y. ( (X(y) & Y(y)) <--> x=y)
one --->
( @ lex(det,
        (l_bind,
         (var:R5,
          rest:(l_bind,
                (var:R6,
                 rest:(atomic,
                       (pred:(e_quant,
                              (var:R2,
                               scope:(a_quant,
                                      (var:R3,
                                       scope:(iff,
                                              wff1:(and,
                                                     wff1:(R5, atomic,
                                                           param1:R3),
                                                     wff2:(R6, atomic,
                                                           param1:R3)),
                                              wff2:(equal,
                                                     wff1:R2,
                                                     wff2:R3))))))),
                        arg1:R5,
                        arg2:R6))))))).

% lex
% Lambda x. man(x)
man --->
( @ lex(cn,
        (l_bind, (var:R5,
                   rest:(atomic, (pred:man, arg1:R5, param1:R5)))))).

% lex
% Lambda x. woman(x)
woman --->
( @ lex(cn,
        (l_bind, (var:R5,
                   rest:(atomic, (pred:woman, arg1:R5, param1:R5)))))).

% lex
% Lambda P. Exists x. ((x = john) & P(x))
john --->
( @ lex(np,
        (l_bind,
         (var:R5,
          rest:(atomic,
                pred:(e_quant,
                      (var:R2,
                       scope:(and,
                              wff1:(equal, wff1:R2, wff2:john),
                              wff2:(R5, atomic, param1:R2)))),
                 arg1:R5))))).

% lex
% Lambda P. Exists x. ((x = mary) & P(x))
mary --->
( @ lex(np,
        (l_bind,
         (var:R5,
          rest:(atomic,
                pred:(e_quant,
                      (var:R2,
                       scope:(and,
                              wff1:(equal, wff1:R2, wff2:mary),
                              wff2:(R5, atomic, param1:R2)))),
                 arg1:R5))))).

% lex
% Lambda x. smoke(x)
smokes --->
( @ lex(vi,
        (l_bind, (var:R5,
                   rest:(atomic, (pred:smoke, form:fin,
                                  arg1:R5, param1:R5))))) ).

% lex
% Lambda x. smoke(x)
to_smoke --->
( @ lex(vi,
        (l_bind, (var:R5,
                   rest:(atomic, (pred:smoke, form:inf,
                                  arg1:R5, param1:R5))))) ).

% lex
% Lambda x. talk(x)
talks --->
( @ lex(vi,
        (l_bind, (var:R5,
                rest:(atomic, (pred:talk, form:fin,
                               arg1:R5, param1:R5))))) ).

% lex
% Lambda x. suffer(x)
suffers --->
( @ lex(vi,
        (l_bind, (var:R5,
                rest:(atomic, (pred:suffer, form:fin,
                               arg1:R5, param1:R5))))) ).

% lex
% Lambda x. leave(x)
to_leave --->
( @ lex(vi,
        (l_bind, (var:R5,
                rest:(atomic, (pred:leave, form:inf,
                               arg1:R5, param1:R5))))) ).

% lex
% Lambda y. Lambda x. love(x,y)
loves --->
( @ lex(vt,
        (l_bind, (var:R6,
                rest:(l_bind, (var:R5,
                                rest:(atomic, (pred:love, form:fin,
                                               arg1:R5,
                                               arg2:R6,
                                               param1:R5,
                                               param2:R6))))))) ).

% lex
% Lambda y. Lambda x. love(x,y)
is_loved_by --->
( @ lex(vt,
        (l_bind, (var:R6,
                rest:(l_bind, (var:R5,
                                rest:(atomic, (pred:love, form:pas,
                                               arg1:R5,
                                               arg2:R6,
                                               param1:R5,
                                               param2:R6))))))) ).

% lex
% Lambda Y. Lambda x. want(x, Y(x))
% Note that Vic shares the subject (R5) with the verb it modifies (R6).
wants --->
( @ lex(vic,
        (l_bind, (var:(R6, form:inf, param1:R5),
                rest:(l_bind, (var:R5,
                                rest:(atomic, (pred:want, form:fin,
                                               arg1:R5,
                                                     arg2:R6,
                                                     param1:R5,
                                                     param2:R6))))))) ).

% lex
% Lambda X. Lambda x. good(X(x))
good --->
( @ lex(adj,
        (l_bind, (var:R5,
                   rest:(l_bind, (var:R6,
                                   rest:(atomic,(pred:good,
                                                 arg1:R5,
                                                 param1:R6))))))) ).

% lex
% Lambda X. Lambda x. slowly(X(x))
slowly --->
( @ lex(pred_adv,
        (l_bind, (var:R5,
                   rest:(l_bind, (var:R6,
                                   rest:(atomic,(pred:slowly,
                                                 arg1:R5,
                                                 param1:R6))))))) ).

% lex
% Lambda X. Lambda x. passionately(X(x))
passionately --->
( @ lex(pred_adv,
        (l_bind, (var:R5,
                   rest:(l_bind, (var:R6,
                                   rest:(atomic,
                                         (pred:passionately,
                                          arg1:R5,
                                          param1:R6))))))) ).

% lex
% Lambda X. Lambda x. urgently(X(x))
urgently --->
( @ lex(pred_adv,
        (l_bind, (var:R5,
                   rest:(l_bind, (var:R6,
                                   rest:(atomic,
                                         (pred:urgently,
                                          arg1:R5,
                                          param1:R6))))))) ).

% lex
% Lambda X. necessarily(X)
necessarily --->
( @ lex(sent_adv,
        (l_bind, var:R5,
         rest:(atomic, pred:necessarily,
                       arg1:R5))) ).

% lex
and --->
( @ lex(conj,
        (atomic,
         (pred:(and, wff1:R5, wff2:R6),
          arg1:R5, arg2:R6)))).

% lex
or --->
( @ lex(conj,
        (atomic,
         (pred:(or, wff1:R5, wff2:R6),
          arg1:R5, arg2:R6)))).

% lex
who --->
( @ lex(reltvzr,
        (atomic,
         (pred:(rel_and, wff1:R5, wff2:R6),
          arg1:R5, arg2:R6)))).
```

B.2 Sample queries

```
%macros
%%%**********************  Macros
syn(Cat) macro
(syn:(syn, cat:Cat)).

%query

%%%% every man smokes
%%%% Forall x. (man(x) -> smoke(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(a_quant,
            var:R2,
            scope:(if,
                   wff1:(R5, atomic, pred:man,   arg1:R2, param1:R2),
                   wff2:(R6, atomic, pred:smoke, arg1:R2, param1:R2))),
      arg1:R5,
      arg2:R6)).

%%%% every good man smokes
%%%% Forall x. ((good(man))(x) -> smoke(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(a_quant,
            var:R2,
            scope:(if,
                   wff1:(R5, atomic, pred:good,
                         arg1:(R7, pred:man,
                               arg1:R2, param1:R2),
                         param1:R2),
                   wff2:(R6, atomic, pred:smoke,
                         arg1:R2, param1:R2))),
      arg1:R5,
      arg2:R6)).

%%%% every good man smokes slowly
%%%% Forall x. ((good(man))(x) -> (slowly(smoke))(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(a_quant,
            var:R2,
            scope:(if,
                   wff1:(R5, atomic,
                         (pred:good,
                          arg1:(R7, pred:man, arg1:R2, param1:R2),
                          param1:R2)),
                   wff2:(R6, atomic,
                         (pred:slowly,
                          arg1:(R8, pred:smoke, arg1:R2, param1:R2),
                          param1:R2)))),
      arg1:R5,
      arg2:R6)).

%%%% john smokes
%%%% Exists x. ((x = john) & smoke(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic, pred:smoke,
                         arg1:R2, param1:R2))),
      arg1:R5)).

%%%% john smokes slowly
%%%% Exists x. ((x = john) & (slowly(smoke))(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic,
                         (pred:slowly,
                          arg1:(R7, pred:smoke,
                                arg1:R2, param1:R2),
                          param1:R2)))),
      arg1:R5)).

%%%% john loves mary
%%%% Exists x. ((x = john) &
%%%%            love(x, Lambda P. Exists y.((y = mary) & P(y))))
>>>>
(phrase,
  syn:(syn, cat:s),
  sem:(arg_1,
    pred:(e_quant,
      var:R2,
      scope:(and,
        wff1:(equal, wff1:R2, wff2:john),
        wff2:(R5,atomic_2_2,
          pred:love,
          arg1:R2,
          arg2:(R6,l_bind,
            var:R7,
            rest:(arg_1,
              pred:(e_quant,
                var:R1,
                scope:(and,
                  wff1:(equal, wff1:R1, wff2:mary),
                  wff2:(R7, param_1, pred:sem, param1:R1))),
              arg1:R7)),
          param1:R2,
          param2:R6))),
    arg1:R5)).

%%%% john loves mary passionately
%%%% Exists x. ((x = john) &
%%%%            (passionately(love))(x, Lambda P. Exists y.((y = mary) & P(y))))
>>>>
(phrase,
  syn:(syn, cat:s),
  sem:(arg_1,
    pred:(e_quant,
      var:R2,
      scope:(and,
        wff1:(equal, wff1:R2, wff2:john),
        wff2:(R5, atomic,
          pred:passionately,
          arg1:(R6,atomic_2_2,
                 pred:love,
                 arg1:R2,
                 arg2:(R7,l_bind,
                   var:(R8,param_1,(pred:sem, param1:sem)),
                   rest:(arg_1,
                     pred:(e_quant,
                       var:R1,
                       scope:(and,
                         wff1:(equal, wff1:R1, wff2:mary),
                         wff2:R8)),
                     arg1:R8)),
                 param1:R2,
                 param2:R7),
          param1:R2))),
    arg1:R5)).

%%%% every man who smokes suffers
%%%% Forall x. ((man(x) & smoke(x)) -> suffer(x))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(a_quant,
            var:R2,
            scope:(if,
                   wff1:(R5, atomic,
                         pred:(rel_and,
                               wff1:(R3, pred:man,
                                     arg1:R2, param1:R2),
                               wff2:(R4, pred:smoke,
                                     arg1:R2, param1:R2)),
                         arg1:R3, arg2:R4,
                         param1:R2),
                   wff2:(R6, atomic, pred:suffer,
                         arg1:R2, param1:R2))),
      arg1:R5,
      arg2:R6)).

%%%% john smokes and mary talks
%%%% Exists x. ((x = john) & smoke(x)) & Exists y. ((y = mary) & talk(y))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic, pred:(and,
                    wff1:(R7, atomic,
                          pred:(e_quant,
                                var:R2,
                                scope:(and,
                                       wff1:(equal,
                                             wff1:R2, wff2:john),
                                      wff2:(R5, atomic, pred:smoke,
                                            arg1:R2, param1:R2))),
                          arg1:R5),
                    wff2:(R8, atomic,
                          pred:(e_quant,
                                var:R3,
                                scope:(and,
                                       wff1:(equal,
                                             wff1:R3, wff2:mary),
                                       wff2:(R6, atomic, pred:talk,
                                             arg1:R3, param1:R3))),
                          arg1:R6)),
      arg1:R7, arg2:R8)).

%%%% necessarily john smokes
%%%% necessarily(Exists x. ((x = john) & smoke(x)))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:necessarily,
      arg1:(R1, atomic,
                pred:(e_quant,
                      var:R2,
                      scope:(and,
                             wff1:(equal, wff1:R2, wff2:john),
                             wff2:(R5, atomic, pred:smoke,
                                   arg1:R2, param1:R2))),
                arg1:R5))).

%%%% john smokes or talks
%%%% Exists x. ((x = john) & ((smoke(x)) | (talk(x))))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic,
                         pred:(or,
                               wff1:(R7, atomic, pred:smoke,
                                     arg1:R2, param1:R2),
                               wff2:(R8, atomic, pred:talk,
                                     arg1:R2, param1:R2)),
                         arg1:R7, arg2:R8, param1:R2))),
      arg1:R5)).

%%%% john wants to smoke
%%%% Exists x. ((x = john) & want(x, smoke(x)))
>>>>
```

```
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic, pred:want,
                         arg1:R2,
                         arg2:(R3, atomic, pred:smoke,
                               arg1:R2, param1:R2),
                         param1:R2,
                         param2:R3))),
      arg1:R5)).

%%%% john wants to leave urgently
%%%% ('urgently' modifies 'wants')
%%%% Exists x. ((x = john) & (urgently(want))(x, leave(x)))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic, pred:urgently,
                         arg1:(R6, atomic, pred:want,
                               arg1:R2,
                               arg2:(R3, atomic, pred:leave,
                                     arg1:R2, param1:R2),
                               param1:R2,
                               param2:R3,
                         param1:R2)))),
      arg1:R5)).

%%%% john wants to leave urgently
%%%% ('urgently' modifies 'leave')
%%%% Exists x. ((x = john) & want(x, (urgently(leave))(x)))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(e_quant,
            var:R2,
            scope:(and,
                   wff1:(equal, wff1:R2, wff2:john),
                   wff2:(R5, atomic, pred:want,
                         arg1:R2,
                         arg2:(R3, pred:urgently,
                               arg1:(atomic, pred:leave,
                                     arg1:R2, param1:R2),
                              param1:R2),
                        param2:R2,
                        param2:R3))),
      arg1:R5)).

%%%% every good man who talks loves mary
%%%% Forall x. (( (good(man))(x) & talk(x) )->
%%%%            love(x, Lambda P. Exists y. ((y=mary) & P(y))))
>>>>
(phrase,
 syn:(syn, cat:s),
 sem:(atomic,
      pred:(a_quant,
            var:R2,
            scope:(if,
                   wff1:(R5, atomic,
                         pred:(rel_and,
                               wff1:(R8, atomic, pred:good,
                                     arg1:(pred:man, arg1:R2, param1:R2),
                                     param1:R2),
                               wff2:(R9, pred:talk,
                                     arg1:R2, param1:R2)),
                         arg1:R8, arg2:R9,
                         param1:R2),
                   wff2:(R6, atomic,
                         pred:love,
                         arg1:R2,
                         arg2:(R3,l_bind,(
                           var:R7,
                           rest:(atomic,(
                             pred:(e_quant,(
                               var:R1,
                               scope:(and,
                                 wff1:(equal, wff1:R1, wff2:mary),
                                 wff2:(R7,param_1,(pred:sem, param1:R1))))),
                             arg1:R7)))),
                         param1:R2,
                         param2:R3))),
      arg1:R5,
      arg2:R6)).
```

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Inc., Reading, MA.

Aho, Alfred V. and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.* Series in Automatic Computation. Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA.

Calder, Jonathan, Mike Reape, and Henk Zeevat. 1989. An algorithm for generation in unification categorial grammar. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240, Manchester, England.

Carpenter, Bob. 1992a. ALE - the attribute logic engine: User's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, December.

Carpenter, Bob. 1992b. *The Logic of Typed Feature Structures. With Applications to Unification Grammars, Logic Programs and Constraint Resolution.* Cambridge University Press.

Cole, Ronald A., Joseph Mariani, Hans Uszkoreit, Annie Zaenen, and Victor Zue, editors. 1995. *Survey of the State of the Art in Human Language Technology.* Available electronically at `http://www.cse.ogi.edu/CSLU/HLTsurvey/`, Oregon Graduate Institute of Science and Technology, Portland, Oregon, USA.

Earley, Jay. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February.

Elhadad, Michael, 1989. *FUF: the Universal Unifier User Manual.* Department of Computer Science, Columbia University, New York.

Gamut, L.T.F. 1991. *Logic, Language, and Meaning. Volume 2: Intensional Logic and Logical Grammar.* The University of Chicago Press, Chicago, USA.

Gerdemann, Dale D. 1991. Parsing and generation of unification grammars. Cognitive Science Technical Report CS-91-06 (Language Series), The Beckman Institute, 405 North Mathews Avenue, Urbana, IL 61801, USA.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Inc., Reading, MA.

Hovy, Eduard, Gertjan van Noord, Guenter Neumann, and John Bateman. 1995. Language generation. In Cole et al. (Cole et al., 1995), pages 161–188.

Johnson, Mark. 1988. *Attribute-Value Logic and the Theory of Grammar*, volume 16 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford University, Stanford, CA 94305, USA.

Kaplan, Ronald M. and Joan Bresnan. 1982. Lexical-functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations.* The MIT Press, Cambridge, MA, pages 173–281.

Kay, Martin. 1996. Chart generation. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 200–204, Santa Cruz, CA, USA. Association for Computational Linguistics.

Kernighan, Brian W. and Dennis M. Ritchie. 1988. *The C Programming Language.* Prentice Hall Software Series. Prentice Hall, Inc., Englewood Cliffs, New Jersey, USA, second edition.

Nerbonne, John. 1992. A feature-based syntax/semantics interface. Research Report RR-92-42, Deutsches Forschungszentrum fur Kunstliche Intelligenz, GmBH, August.

Neumann, Guenter. 1994. *A Uniform Computational Model for Natural Language Parsing and Generation.* Ph.D. thesis, Universitaet des Saarlandes, Saarbruecken, Germany.

Ousterhout, John K. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, Inc.

Pereira, Fernando C.N. and Stuart M. Shieber. 1987. *Prolog and Natural-Language Analysis.* Number 10 in CSLI Lecture Notes. CSLI, Stanford University, Stanford, CA.

Pollard, Carl J. and Ivan A. Sag. 1994. *Head-driven Phrase Structure Grammar.* Studies in Contemporary Linguistics. The University of Chicago Press, Chicago, USA.

Popowich, Fred. 1996. A chart generator for shake and bake machine translation. In *Proceedings of AI-96 – 11th Canadian Conference on Artificial Intelligence*, Toronto, Canada, May.

Reiter, Ehud. 1994. Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 163–170, Nonantum Inn, Kennebunkport, Maine, June 21-24,.

Samuelsson, Christer. 1995. An efficient algorithm for surface generation. In *Proc. of the 14th International Joint Conference on Artificial Intelligence, Montreal, Canada*, pages 1414–1419. Morgan Kaufmann, August.

Shieber, Stuart M. 1986. *An Introduction to Unification-based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. CSLI, Stanford University, Stanford, CA 94305, USA.

Shieber, Stuart M. 1988. A uniform architecture for parsing and generation. In *Proc. of the 12th International Conference on Computational Linguistics*, volume 1, pages 614–619, Budapest, Hungary.

Shieber, Stuart M., Gertjan van Noord, Fernando C. N. Pereira, and Robert C. Moore. 1990. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42, March.

Strzalkowski, Tomek, editor. 1994. *Reversible Grammar in Natural Language Processing.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, The Netherlands.

Trujillo, Arturo. 1997. Determining internal and external indices for chart generation. In *Proc. of the 7th International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-97).*

Warren, David H. D. 1983. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, August.

Whitelock, P. 1992. Shake-and-bake translation. In *Proc. of the 14th International Conference on Computational Linguistics*, pages 784–791, Nantes, France, August.

Wintner, Shuly. 1997. *An Abstract Machine for Unification Grammars.* Ph.D. thesis, Technion, Israel Institute of Technology, Haifa, Israel, January.

Wintner, Shuly, Evgeniy Gabrilovich, and Nissim Francez. 1997a. AMALIA – a unified platform for parsing and generation. In R. Mitkov, N. Nicolov, and N. Nicolov, editors, *Proc. of "Recent Advances in Natural Language Processing" (RANLP'97)*, pages 135–142, Tzigov Chark, Bulgaria, September.

Wintner, Shuly, Evgeniy Gabrilovich, and Nissim Francez, 1997b. *AMALIA – Abstract MAchine for LInguistic Applications – User's Guide.* Laboratory for Computational Linguistics, Computer Science Deparmtent, Technion, Israel Institute of Technology, Haifa, Israel, June.

Younger, D.H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208.