Parsimonious Vole

A Systemic Functional Parser for English

Universität Bremen

Eugeniu Costetchi

Supervisor: Prof. John Bateman

Advisor: Dr. Eric Ras

Faculty 10: Linguistics and Literary Studies
University of Bremen

This dissertation is submitted for the degree of
Doctor of Philosophy

July 2018

I would like to dedicate this thesis to my loving parents . . .

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Eugeniu Costetchi
July 2018

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract ...

Table of contents

List of figures

List of tables

List of definitions

Chapter 1

On Graphs, Feature Structures and Systemic Networks

The parsing algorithm, whose pipeline architecture we have seen in Section **??**, operates mainly with operations on graphs, attribute-value matrices and ordered lists with logical operators. This chapter defines the main types of graphs, their structure and how they are used in the following chapters which detail on the parsing process. This chapter also covers the operations relevant to the parsing algorithm: *conditional traversal and querying* of nodes and edges, *graph matching*, *pattern-graph matching* and *pattern-based node selection, insertion and update.*

While developing the Parsimonious Vole parser a set of representational requirements arose that can be summarised as follows:

- graphic representation
- arbitrary relations (i.e. typed and untyped edges)
- description rich (i.e. features of nodes and edges)
- linear ordering and configurations (i.e. syntagmatic and compositional)
- hierarchical tree-like structure (with a root node) but also orthogonal relations among siblings and non-siblings
- statements of absence of a node or edge (i.e. negative statements in pattern graphs)
- disjunctive descriptions (handling uncertainty)

- conjunctive descriptions (handling multiple feature selections)
- (conditional) pattern specifications (i.e. define patterns of graphs)
- operational pattern specifications (i.e. a functional description to be executed in pattern graphs)

The general approach to construct an SFG parse structure revolves around the graph pattern matching and graph traversal. In the following sections I present the instruments used for building such structures, starting from a generic computer science definition of graphs and moving towards specific graph types covering also the feature structures and conditional sets.

1.1 On graphs

In the field of computational linguistics trees has been taken as the de facto data representation. In Section **??** I have mentioned already that I employ graph and not tree structures.

Firstly, the trees are a special kind of graphs. Anything expressed as a tree is as well a tree. Secondly, we gain a higher degree of expressiveness even if at the expense of computational complexity, a point to which we will come back latter in Section 1.4. This expressiveness is needed when dealing with interconnection of various linguistic theories which in practice is done by mapping the nodes of one tree structure onto the nodes of another one. In addition, the structures are not always trees. There are situations when a node has more than one parent or when a node is connected to its siblings which break the tree structure.

Definition 1.1.1 (Graph)**.** A *graph* $G = (V, E)$ is a data structure consisting of non-empty set V of nodes and a set $E \subseteq V \times V$ of edges connecting nodes.

Definition 1.1.2 (Digraph)**.** A *digraph* is a graph with directed edges. A directed edge $(u, v) \in E$ is an ordered pair that has a start node u and an end node v (with $u, v \in V$)

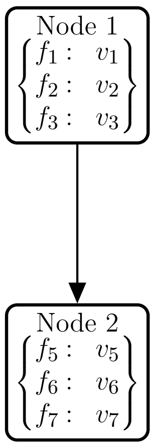
In this thesis the graph nodes are considered to be *feature structures* forming *Feature Rich Graphs* (see Definition 1.1.10). Before formally defining these graphs, I need to address first the notion of feature structure and a few kinds of sets.

In SFL the concept of *feature* takes up an important role. Also features are said to form systems of choices that are structured in relation to one another and are suitable for describing linguistic objects and phenomena.

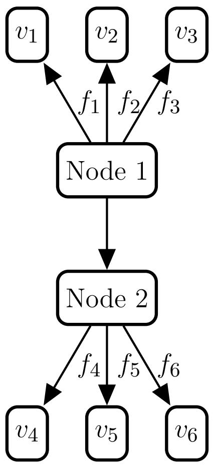
Pollard & Sag (1987) have formally described useful concepts for grammatical representations in the context of Head-Driven Phrase Structure Grammar (HPSG). He adopts the *typed feature structure theory* and extends it in ingenious ways applicable in computational linguistics. Among others, he provides formal definitions for the concepts of *feature structure*, *hierarchy*, *logical evaluation*, *composition* and *unification*, the latter, being key operations in parsing using feature structured grammars.

In this thesis, feature structures are important but only in a simplified version serving as graph node descriptions. The main reason is the difference in approach as the main parsing operations, here, are based on graph pattern matching (introduced in the sections below).

In a broad computer science sense, including Pollard's definition, feature structures are equivalent to graph structures. So any feature structure can be expressed as a graph and any graph can be expressed as a feature structure. But in a narrow sense, as adopted in this thesis, it is useful to employ both concepts but each for a given purpose. The feature structure is reduced to an attribute-value matrix (see Definition 1.1.3) and the graphs to a network of feature structure nodes (see Definition 1.1.10) i.e. no atomic nodes.


Node 1
$\left\{\begin{array}{ll} f_1: & v_1 \\ f_2: & v_2 \\ f_3: & v_3 \end{array}\right\}$
Node 2
$\left\{\begin{array}{ll} f_5: & v_5 \\ f_6: & v_6 \\ f_7: & v_7 \end{array}\right\}$

(a) The graph with feature structure nodes



(b) The graph with atomic nodes

Fig. 1.1 Graphs with atomic nodes and feature structure nodes

The main reasons in this separation are efficiency and practicality. First, it is about handling the atomic values (strings or integers) and (ordered) arrays only as values of feature structures and never as graph nodes. Second, the graphs remain limited in size, close to the conceptualised linguistic structures, i.e. dependency or constituency. Otherwise, the graphs would grow in complexity (a) by at least one more round of nodes for each dependency or constituency node and (b) by adoption of an additional node classification.

For example lets imagine a constituency graph fragment of two nodes *Node 1* and *Node 2* where each has three associated features as it can be seen in Figure 1.1a. If we would insist to dispose of the feature structure within the node and express the features as atomic graph nodes then the result would be a graph structure such as the one in Figure 1.1b.

Definition 1.1.3 (Feature Structure (FS))**.** A *feature structure* F is a finite set of attribute-value pairs $f_i \in F$. A *feature* $f_i = (a, v)$ is an association between an identifier a (a symbol) and a value v which is either a symbol, an ordered set or another feature structure.

Please note that the values of feature structures may be other feature structures allowing, if needed, to construct hierarchical descriptions. But in the current implementation, most of the time the values of the feature structure are either atomic values or arrays.

For convenience I define two functions to access the identifier and value in a feature structure. The function $att(f_i)$ returning the feature identifier $att(f_i) = a$ and the function $val(f_i)$ is a function returning the ascribed value of a feature $(f_i) = v$.

Definition 1.1.3 stipulates that the value of a feature may be also a set (besides an atomic value). The sets used in this thesis need to carry additional properties required for their interpretation. Specifically, it is the order need to be addressed here and the capacity to specify that set elements stand in a certain logical relation one to another (e.g. conjunction, disjunction, negation, etc.). These two properties are covered in Definition 1.1.4 and 1.1.5. For convenience I will assume from now on that sets (see Definition 1.1.4) preserve order even when it is not really required.

Definition 1.1.4 (Set)**.** An (ordered) *set* $S = \{o_1, o_2, ..., o_n\}$ is a finite well defined collection of distinct objects o_i. A set is said to be ordered if the objects are arranged in a sequence such that $\forall o_{i-1}, o_i \in S : o_{i-1} < oi$.

Definition 1.1.5 (Conjunction Set)**.** A *conjunction set* $S_{conj} = (S, conj)$ is a set S whose interpretation is given by the logical operand *conj* (also denoting the type of the set) such that $\forall o_i, o_j \in S : conj(o_i, o_j)$ holds.

The conjunction sets used in current work are *AND-set* (S_{AND}), *OR-set* (S_{OR}), *XOR-set* (S_{XOR}) and *NAND-set* (S_{NAND}). The assigned logical operands play a role in the functional interpretation of conjunction sets. Formally these sets are defined as follows.

Definition 1.1.6 (Conjunctive set)**.** Conjunctive set (also called *AND-set*) is a conjunction set $S_{AND} = \{a, b, c...\}$ that is interpreted as a logical conjunction of its elements $a \wedge b \wedge c \wedge ...$

Definition 1.1.7 (Negative conjunctive set)**.** Negative conjunctive set (also called NAND-set) is a conjunction set $S_{NAND} = \{a, b, c...\}$ that is interpreted as a negation of the logical conjunction of its elements $a \uparrow b \uparrow c \uparrow ...$ equivalent to $\neg(a \wedge b \wedge c \wedge ...)$

Definition 1.1.8 (disjunctive set)**.** Disjunctive set (also called OR-set) is a conjunction set $S_{OR} = \{a, b, c...\}$ that is interpreted as a logical disjunction of its elements $a \vee b \vee c \vee ...$

Definition 1.1.9 (exclusive disjunctive set)**.** Exclusive disjunctive set (also called XOR-set) is a conjunction set $S_{XOR} = \{a, b, c...\}$ that is interpreted as a logical exclusive disjunction of its elements $a \bigoplus b \bigoplus c \bigoplus ...$ equivalent to $(a \wedge \neg(b \wedge c \wedge ...)) \vee (b \wedge \neg(a \wedge c \wedge ...)) \vee (c \wedge \neg(a \wedge b \wedge ...))$

When conjunction sets are used as values in FSs then the type of logical operand dictates the interpretation of the FS. When the set type is S_{AND} then all the set elements hold simultaneously as feature values. If it is a S_{OR} then one or more of the set elements hold as values. If is S_{XOR} then one and only one of set elements holds and finally if it is a S_{NAND} set then none of elements hold as feature values.

The function $\tau(S)$, defined $\tau : S \rightarrow \{S_{AND}, S_{OR}, S_{XOR}, S_{NAND}\}$, returns the type of the conjunction set and the function $size(S)$, defined $size : S \rightarrow \mathbb{N}$, returns the number of elements in the set.

Now that all the necessary basic notions nave been formally defined I now define the feature rich graph and provide a couple of examples afterwards.

Definition 1.1.10 (Feature Rich Graph (FRG))**.** A *feature rich graph* is a digraph whose nodes V are feature structures and whose edges $(u, v, f) \in E$ are three valued tuples with $u, v \in V$ and $f \in F$ an arbitrary feature structure.

Further on, for convenience, when I refer to a graph I will refer to a feature rich digraph unless otherwise stated. The parsing algorithm operates with such graph and they are further distinguished, based on purpose as: *Dependency Graphs* (DG) (example figure 1.2), *Constituency Graphs* (CG) (example figure 1.3) and Pattern Graphs (PG) also referred as *Query Graphs* (QG).

Please note that the edges are defined to carry feature structures. This capacity will not be employed, for example, in the case of constituency graphs; only minimally employed in the case of dependency graphs, where the dependency relation is specified; and fully employed in pattern graphs. Nonetheless, treating all of them as feature rich graphs simplifies the implementation.

Definition 1.1.11 (Dependency Graph)**.** A *dependency graph* is a feature rich digraph whose nodes V correspond to words, morphemes or punctuation marks in the text and carry at least the following features: *word*, *lemma*, part of speech (*pos*) and, when appropriate, the named entity type (*net*); the edges E describe the dependency relation (*rel*).

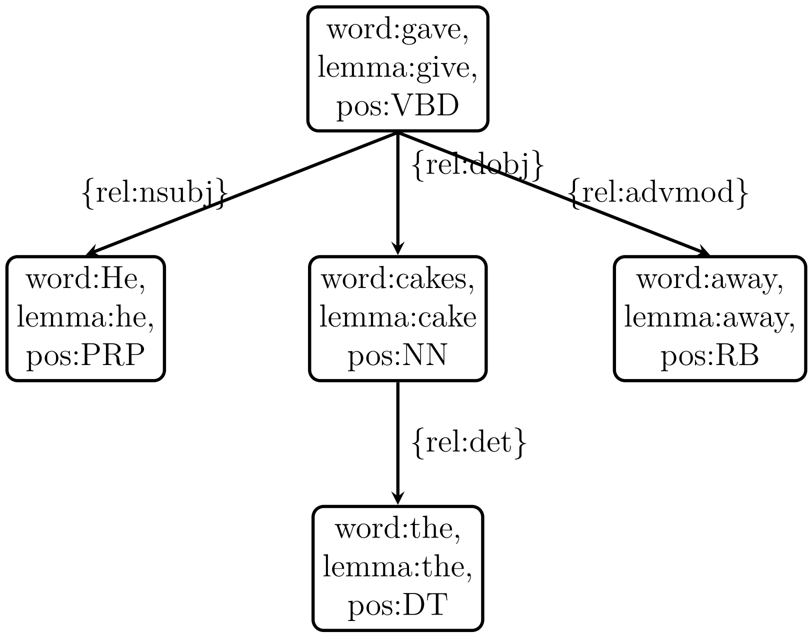

word:gave, lemma:give, pos:VBD
{rel:nsubj}
{rel:dobj}
{rel:advmod}
word:He, lemma:he, pos:PRP
word:cakes, lemma:cake pos:NN
word:away, lemma:away, pos:RB
{rel:det}
word:the, lemma:the, pos:DT

Fig. 1.2 Dependency graph example with FS nodes and edges

Definition 1.1.12 (Constituency Graph)**.** A *constituency graph* is a feature rich digraph whose nodes V correspond to SFL *units* and carry the *unit class* and the element function within the parent unit (except for the root node); while the edges E reflect constituency relations between constituents.

The basic features of a constituent node are the *unit class* and the function(s) it takes, which is to say the *element(s)* it fills in the parent unit (as described in the discussion of theoretical aspects of SFL in Chapter **??**). The root node (usually a clause) is an exception and it does not act as a functional element because it does not have a parent unit. The leaf nodes carry the same features as the DG nodes plus the word class feature which correspond to the traditional part of speech tags.

Apart from the essential features of class and function, the CG nodes carry additional class specific features selected from the relevant system network. The features considered in this thesis are described in Chapter **??**. In addition, the leaf CG nodes contain the features of dependency graph nodes enumerated in Definition 1.1.11. The way CG is enriched with features is described in the next chapter. Below in Figure 1.3, is an example CG that carries tense, modality and polarity features on the clause node in addition to class and element function.

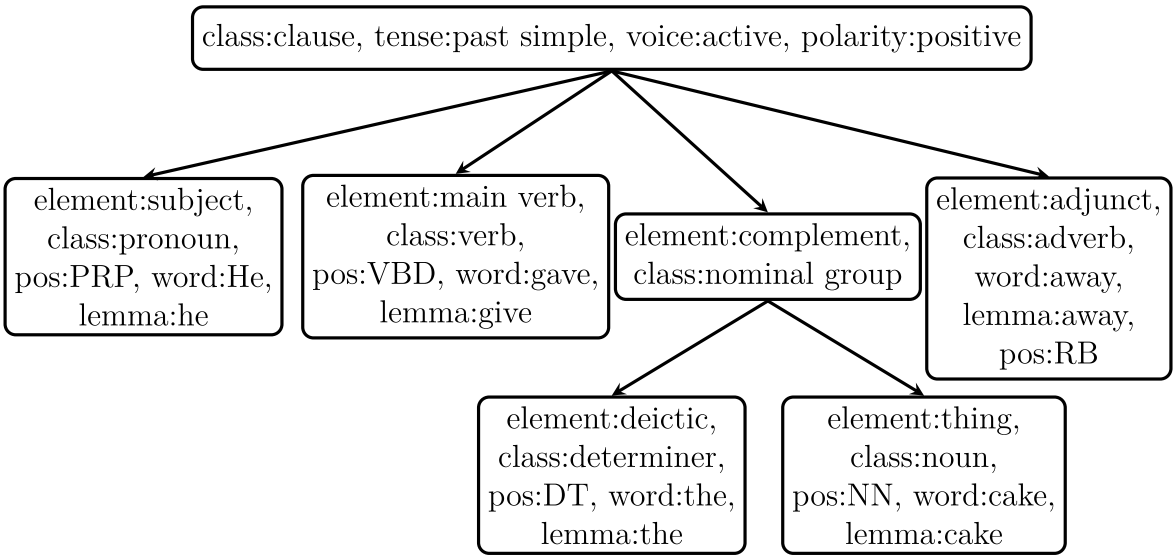

class:clause, tense:past simple, voice:active, polarity:positive
element:subject, class:pronoun, pos:PRP, word:He, lemma:he
element:main verb, class:verb, pos:VBD, word:gave, lemma:give
element:complement, class:nominal group
element:adjunct, class:adverb, word:away, lemma:away, pos:RB
element:deictic, class:determiner, pos:DT, word:the, lemma:the
element:thing, class:noun, pos:NN, word:cake, lemma:cake

Fig. 1.3 Constituency graph example

Next section describes what I call atomic queries and two types of special graph traversal: the conditional and the generative ones.

1.2 Graph query and traversal

Definition 1.2.1 (Atomic Query). Querying $q_V(F,G)$ or $q_E(F,G)$ over the nodes V or edges E of a graph G is an operation that returns a set of nodes or edges filtered by the conditional feature structure F.

For example in Figure 1.4 a potential query is to return all the dashed nodes in which case returns the set of nodes {3,5,7,8}. In a given dependency graph, similar

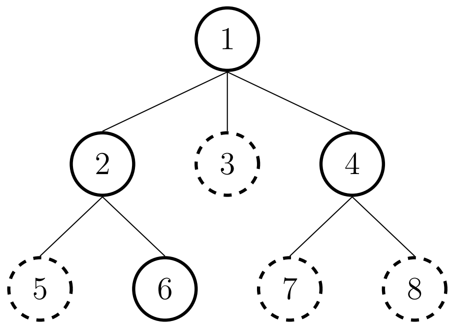


Fig. 1.4 Sample graph with numbered node of two types

operation can be used to select all the determiner nodes. In that case we query the nodes with the condition that the part of speech has DT value i.e. *pos=DT*. The same can be achieved by selecting all the edges connecting a noun to its determiner, then the query is formulated for all edges whose relation type is *rel=det*.

Definition 1.2.2 (Traversal)**.** Traversal $t(V_S, G)$ of a graph G starting from node V_S is a recursive operation that returns a set of sequentially visited nodes neighbouring each other in either *breadth first* (t_{BF}) or *width first* (t_{WF}) orders.

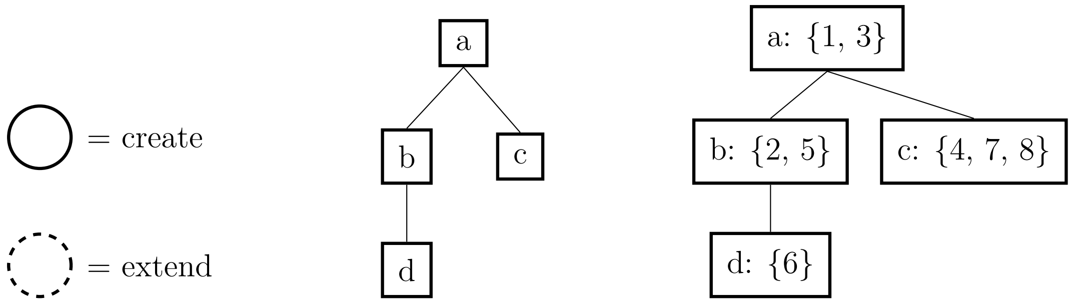
The graph traversal defined in 1.2.2 and especially its conditional and generative extensions defined in 1.2.3 and 1.2.4 are very important operation in this work. The graph traversal is employed either for searching for a node or an edge or finding a sub-graph that fulfils certain conditions on its nodes and edges if it is a conditional traversal. For example in the semantic enrichment phase (that will be described in Section **??**), to ensure that the semantic patterns are applied iteratively to each clause, from a multi-clause CG graphs are selected all clause sub-graphs without including the embedded (dependent) clauses.

Definition 1.2.3 (Conditional Traversal)**.** Conditional traversal $t(F_V, F_E, V_S, G)$ of the graph G starting from node V_S under node conditions F_V and edge conditions F_E is a traversal operation where a node is visited if and only if its feature structure conditionally fulfils the F_V and the edge that leads to this node conditionally fulfils the F_E.

One of the potential complete traversals for the graph from Figure 1.4 starting from the node 1 is {1, 2, 3, 4, 5, 6, 7, 8} using breadth first strategy or {1, 2, 5, 6, 3, 4, 7, 8} for depth first strategy. On the other hand, a conditional traversal of non-dashed nodes staring from the node 1 results in {1, 2, 4, 6}, {1, 4, 2, 6} or {1, 2, 6, 4}. The first two traversals corresponding to the width first strategy and the third one to the depth first strategy.

I also use the graph traversal to execute generative operations on a parallel graph. For example DG traversal is employed for bootstrapping (i.e. creating in parallel) the CG as it was previously motivated in Section **??**.

Definition 1.2.4 (Generative Traversal)**.** Generative traversal $g(M,G)$ of a source graph G via a operation matrix M is an operation resulting in the creation of the target graph H by contextually applying generative operations to bring the latter into existence. The operation matrix M is a set of tuples (ctx,o,p) that link the visited source node context ctx (as features of the node, the edge and previously visited neighbour) to a certain operation o that shall is executed on the target graph H with parameters p.



(a) The rule set (b) The target (c) The explained target

Fig. 1.5 The generative traversal result for Figure 1.4 using create and extend operations

Next I provide an rough description of what happens when a generative traversal is executed and the exact algorithm will be described in detail in Section **??**. For example lets assume that only two types of operation are needed for our task at hand. First is to create a new node on the target graph once a non-dashed node is visited on the source graph. And, second, is to pass without doing anything the dashed nodes. This is schematically represented in Figure 1.5a. Lets now apply these operations on traversing the example graph using breadth first strategy following the order provided above {1, 2, 3, 4, 5, 6, 7, 8}. The traversal graph is considered source graph and the target graph is empty at the beginning of the process. Upon visiting the node 1 a first node is created on the target graph which is labelled *a*. When traversing nodes 2 and 4 then each of them signal creation of the nodes *b* and *c* as children of *a* in the target graph and correspondingly node 6 signals creation of node *d*. The final target graph is depicted in Figure 1.5b and in Figure 1.5c the source nodes are embedded into the target node to make explicit that the non-dashed nodes (i.e. 3, 5, 7, 8) are simply passed over without any generative operation.

Now that generative traversal is defined, by analogy, *update*, *insert* and *delete* traversals can be defined on the source or target graph by using the same mechanism of *operation matrices* mapping contexts of visited nodes and edges to update, insert and delete operations. In this work, however, these operations are not used and therefore omitted here.

In Section 1.1 the last two definitions were for the constituency and dependency graphs. They are used in this thesis to represent grammatical analysis of of a sentence. Next we will look at a special type of graph which represents fragments of structure repeatable across multiple analysis. They represent generalisations or patterns that usually are associated with grammatical features or a set of features. These graphs are called pattern graphs and the next section is dedicated to them.

1.3 Pattern graphs

Regardless of the type, constituency or dependency, the parsing process (which will be described in Chapter **??**) relies heavily on identifying patterns in graphs. The patterning is described as both graph structure and feature presence (or absence) in the nodes or edges. The *pattern graphs* (defined in 1.3.1) are special kinds of graphs meant to represent small (repeatable) parts of parse graphs that, in the context of the current work, are used to identify grammatical features.

Definition 1.3.1 (Pattern Graph)**.** A *pattern graph* (PG) describes regularities in node-edge configuration and feature structure including descriptions of *negated nodes or edges* (i.e. absence of), logical operators over feature sets (AND, OR, XOR and NAND) and operations once the pattern is identified in a target graph (select, insert, delete and update).

Next I discuss two example of pattern graphs. One example shows a pattern graph encoding the present perfect continuous tense, which traditional grammar defines as in Table 1.1. Afterwards, the second example will show how the notion of linear succession among nodes is accounted for in the pattern graphs for declarative and interrogative mood.

Examples 1–3 show variations of present perfect continuous tense in a simple clause according to declarative and interrogative mood and the “has” contraction. Of course there are more variations possible for example according to voice (active and

has/have	+	*been*	+	*Vb-ing*
to have, present simple		to be, past participle		verb, present participle

Table 1.1 Present perfect continuous tense

passive) but they are omitted here because they adds combinatorially to the number of examples and the ones provided already serve their purpose here. The Figures 1.6-1.8 represent corresponding dependency parses for these examples (generated with Stanford dependency parser).

(1) He has been reading a text.

(2) He's been reading a text.

(3) Has he been reading a text?

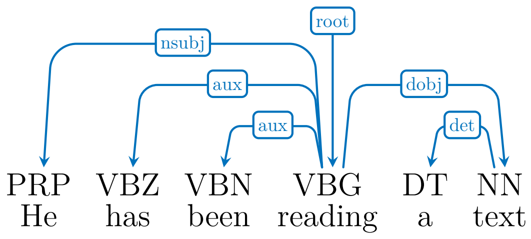


Fig. 1.6 Present perfect continuous: indicative mood, un-contracted "has"

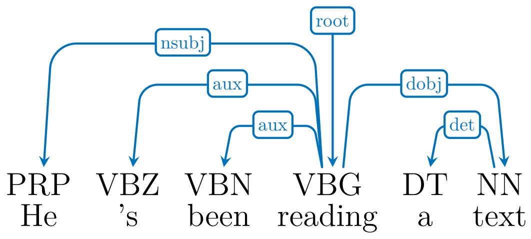


Fig. 1.7 Present perfect continuous: indicative mood, contracted "has"

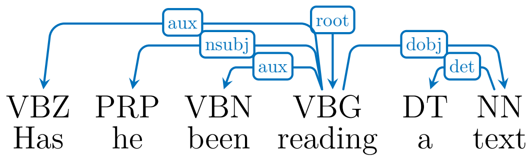


Fig. 1.8 Present perfect continuous: interrogative mood, un-contracted "has"

The present perfect continuous tense can be formulated as a pattern graph (including voice) over the dependency structure as illustrated in Figure 1.9. In this pattern the main lexical verb is *present participle* indicated via the *VBG* part of speech. It is accompanied by two auxiliary verbs: *to be* in *past participle* (*VBN*) form and *to have* in *present simple* form specified by either *VBZ* for 3rd person or *VBP* for non-3rd person. Also the *to be* can be either connected by the *aux* relation or in case of passive form by the *auxpass* relation. Note that the pattern in Figure 1.9 constraints the edge type (using an OR-set) to the verb *to be* which can be either *aux* or *auxpass* and the part of speech of the verb *to have* which can be *VBZ* or *VBP*.

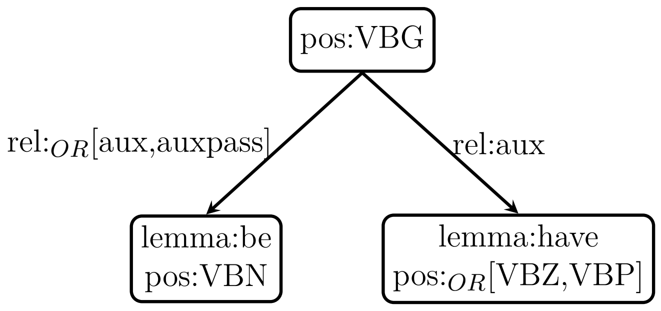


Fig. 1.9 The graph pattern capturing features of the present perfect continuous tense

One of the fundamental features of language is its sequentiality and directionality. This aspect is not inherent in graphs. In the simplest form, they just describe connections between nodes and are agnostic to any meaning or interpretation. Next, I introduce the way I deal with linear order in the pattern graphs.

Lets consider the clause mood and encode the distinction between *declarative* and *Yes/No interrogative* moods. In SFG this features is described in terms of the relative order of clause elements. If the finite is before the subject then the mood is Yes/No-interrogative, whereas when the finite succeeds subject then the mood is declarative. Example 3 contrasts in mood with 1 and 2.

Order can be specified in absolute or relative terms and partially or exhaustively. In order to cover these three kinds of constraints, I introduce three special features: the node *id*, *precede* and *position*. Node id takes a token to uniquely identify a node in the graph, the precede feature takes an ordered set to indicate the (partial) linear precedence to other node ids, and the position feature indicates the absolute position of a node.

One way to introduce order among nodes is then by marking them with an absolute position. This is a good method applicable to parse graphs. The DGs and CGs, are automatically assigned at creation time the absolute position of the node in the sentence text via the feature *position*. This feature is present in the leaf nodes only and corresponds to the order number in which they occur in the sentence text while the non-leaf node's position is considered to be the lowest position of its constituent nodes. The absolute position description is rarely used in the PGs. The only cases are to state that the constituent is first or last position in a sentence.

Another way to specify node order is through relative precedence, for which the node id and the precedence features are introduced above. This is the preferred method to provide linear precedence dimension in pattern graphs. It is also relative so the specification can be partial. With this method a node specifies that it precedes a set of other nodes.

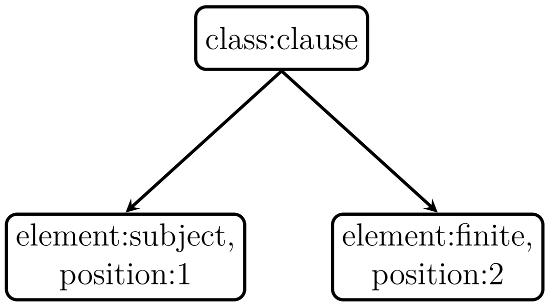


Fig. 1.10 Declarative mood pattern graph with relative element order

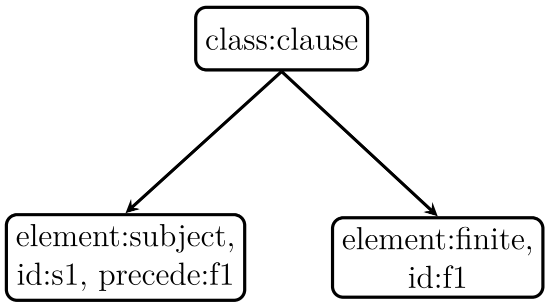


Fig. 1.11 Declarative mood pattern graph with absolute element order

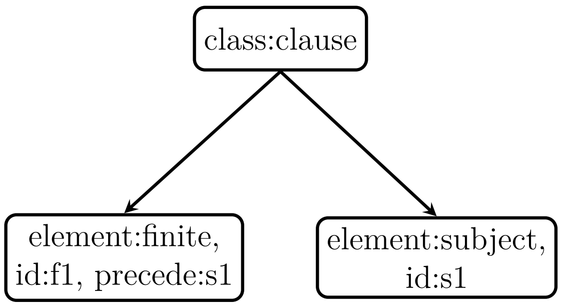


Fig. 1.12 Pattern graph for Yes/No interrogative mood

To continue the example of mood features, I illustrate in Figures 1.10 and 1.11 the use of relative and absolute node ordering constraints for declarative mood; and in

Figure 1.12, I depict the PG for the Yes/No interrogative mood. In both cases I use the relative node ordering.

Patterns like the ones explained above can be created for a wide range of grammatical features. Once the grammatical feature is encoded as a pattern graph it can be identified in parse graphs (DG or CG) via *graph pattern matching* operation. Moreover, once the pattern is identified it can act as a triggering condition to for various operation. For example an operation can be to inject the identified feature into the parse graph and this way enriching its content. Coming back to out tense example above, once the pattern 1.9 is identified then the clause can be marked with the *tense* feature. In the next section I address in detail the pattern graph matching operation.

1.4 Graph matching

So far we have discussed about Constituency and Dependency graphs and, in last section, about Pattern Graphs, which introduces the intuition that they are meant to be matched against or found in other graphs. I will address now what does it mean for two graphs to be the same, i.e. *(structurally) isomorphic*, and what does it mean in the current work. In mathematics the structure-preserving mappings from one structure to another one of the same type is called *morphism*.

Definition 1.4.1 (Morphism)**.** A morphism $f: X \rightarrow Y$ is a structure preserving map from one object X to the other Y where the objects are complex structures such as sets, feature structures or graphs.

Definition 1.4.2 (Isomorphism)**.** The morphism $f: X \rightarrow Y$ is called *isomorphism* if there exists an inverse morphism $g: Y \rightarrow X$ such that $f \circ g = id_X$ and $g \circ f = id_X$.

The graph matching is known in computer science as (sub-)graph isomorphism testing. Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are isomorphic if mapping the nodes of G with the nodes of H, preserving the edge neighbourhood, results in graph H.

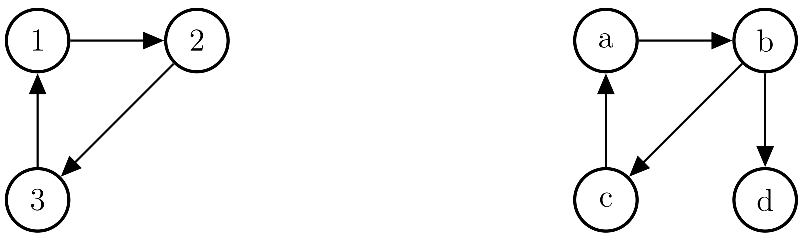
Definition 1.4.3 (Graph isomorphism)**.** An isomorphism of graph $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is a bijective function $f: V_G \rightarrow V_H$ such that if any two nodes $u, v \in V_G$ from G are adjacent $(u, v) \in E_G$ then $f(u), f(v)$ are adjacent in H as well $(f(u), f(v)) \in E_H$.

Graphs do not need to have the same number of nodes or edges. We say that a graph is smaller than another one, denoted $G \leq H$, when its number of nodes is less than that of the other graph. In this case the isomorphism is established to a sub-graph of H. In this work pattern graphs are usually simpler and smaller than the (constituency or dependency) graph they are matched against. In this case, the mapping function is relaxed from bijective (perfect mapping from first to second graph) to an injective one (each node from first has a correspondent in the second one).

Definition 1.4.4 (Sub-graph isomorphism)**.** Given two feature rich graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ and a congruence operator $\cong$, G is sub-graph isomorphic to G (denoted $G \subseteq H$) if there is an injective function $f : V_G \to V_H$ such that

- $\forall v \in V_G, f(v) \in V_H$ and
- any two nodes adjacent in G, $(u, v) \in E_G$, are also adjacent in H, $(f(u), f(v)) \in E_H$

Definition 1.4.5 (Graph matching)**.** For two given graphs G and H, where $G \leq H$, *graph matching* is the operation of testing whether G is structurally isomorphic to H or a sub-graph of H.



(a) The pattern graph (b) The target graph

Fig. 1.13 Sub-graph isomorphism {1=a, 2=b, 3=c}

In Figure 1.13a is depicted a labelled graph that is isomorphic to a sub-graph in Figure 1.13b. The example graphs presented in Figure 1.13 have atomic labels as nodes and the isomorphism is established as a mapping between labels. Section 1.1 above mentions that the graph considered in this thesis have feature structures as their nodes and not atomic nodes. But in case of feature rich graphs additional rules how to establish the isomorphism need to be provided because there are multiple ways to approaching it.

Lets consider Figure 1.14 where the graph nodes are feature structures instantiating features: f_1 and f_2. One way to approach isomorphism in this scenario is by the value of one feature, for example f_1. Then we can identify two sub-graph isomorphisms: {1=a, 2=b, 3=c} and {1=b, 2=d, 3=e}. This approach, besides additional specification what

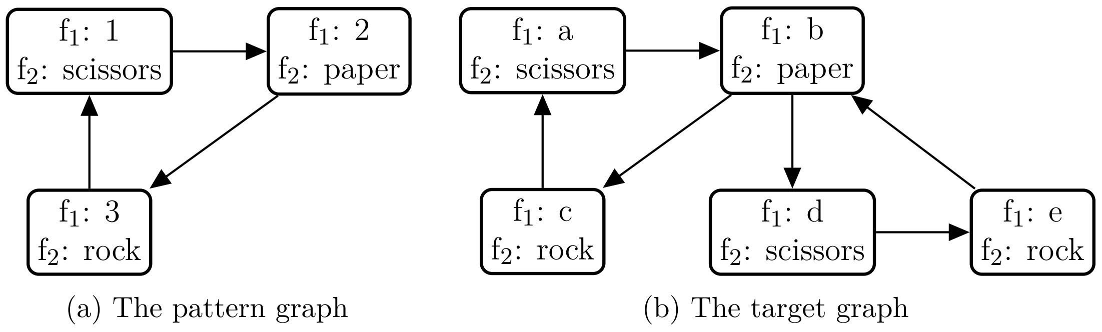


Fig. 1.14 An example of rich sub-graph isomorphism

values to compare, i.e. f$_1$s, is the same as providing a sub-graph isomorphism on the labelled graphs from Figure 1.13.

In addition to the rule above, lets add a constraint that the isomorphism is not only a mapping between the feature values (numbers to letters) but also that the mapped values are identical (strict value equality). If we consider the strict equality rule applied on f$_1$ feature, there is no isomorphism between the two graphs because first one uses numbers (1, 2, 3) and the second uses letters (a, b, c, d). Now if we turn to the values of f$_2$ and apply the same rule then there is one isomorphism possible {paper=paper, rock=rock, scissors=scissors}. The second one, even if it is a cycle, {paper=paper, rock=scissors, scissors=rock} is no longer acceptable because the "scissors" and "rock" switched places in the target graph and it would have been acceptable as a mapping, but not as strict value equality. Formally, the additional equality constraint can be expressed on the graph isomorphism f (defined in Definition 1.4.4) as $u = f(u)$.

This brings us to the idea that a *congruence operator* (denoted $\cong$) needs to be defined for the feature rich (sub-)graph isomorphism. The *feature congruence* (broadly defined in Definition 1.4.6) can be defined in terms of feature structures comparison in accordance with the goals of the particular task or satisfying certain set of conditions. Sometimes, in this thesis, I will refer to the congruence operator also as *identity settling function*.

Definition 1.4.6 (Congruence)**.** Congruence is a comparison operator returning a truth value $\cong: FS_1 \times FS_2 \to \{true, false\}$ implementing the conditions under which FS_1 is considered equal to FS_2.

Using the congruence instead of equality is a means to specify additional set of constraints in generating the graph isomorphism and is captured in Definition 1.4.7 below. Consequently the rich graph matching operation is testing for rich sub-graph isomorphism.

Definition 1.4.7 (Rich sub-graph isomorphism)**.** Given two feature rich graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ and a congruence operator $\cong$, G is sub-graph isomorphic to G (denoted $G \subseteq H$) if there is an injective function $f : V_G \to V_H$ such that

- $\forall v \in V_G, f(v) \in V_H$ and
- $v \cong f(v)$, i.e. each node in G is congruent with its correspondent in H and
- any two nodes which are adjacent in G, $(u,v) \in E_G$, are also adjacent in H, $(f(u), f(v)) \in E_H$

There are two types of congruence that are of relevance for the context of the current work. Also the primarily graph matching is tested between pattern graphs and dependency or constituency graphs. Next section, addresses this type of graph matching in more detail.

1.5 Pattern Graph Matching

An extension and particular case of rich graph matching is the *pattern graph matching* where H (Definition **??**) is a pattern graph (Definition 1.3.1) and the identity checking function(s) are not strict but permissive to feature over-specification of node and edge feature structures.

I will define now how the identity checking function (identity morphism) used for pattern graph matching. It operates on feature structure values, which can be either atomic types *simple* of one of the conjunctive sets: S_{AND}, S_{OR}, S_{XOR} and S_{NAND}. I use both set theoretic notations for inclusion $\subseteq$, intersection $\cup$ and element belonging to a set $\in$ and the logical notations for conjunction $\wedge$ and tautology $\top$. The unary function $T(x)$ returns the type of x element.

When checking the identity of two feature values, three cases can be asserted: $x = y$ i.e. x is definitely equal to y, $x \neq y$ i.e. x is definitely different from y and $x \sim y$ i.e. x is maybe (or could be) equal to y.

I define below two identity morphisms: (a) *permissive* $I_{permissive}$ (defined by Equation 1.2) which includes the uncertain cases and (b) *strict* I_{strict} (defined by Equation 1.1) which excludes the uncertain cases. The main difference between the two morphism functions is whether on the right side (the instance graph) any uncertainty is accepted. This is to say any of the disjunctive sets s_{OR} and S_{XOR}.

Definition 1.5.1 (Strict Pattern Graph Matching)**.** *Strict pattern graph matching* is a rich graph matching where a morphism for the pattern graph H is found in the target graph G given that $H \leq G$ and that for any node $p \in H$ there is a node $r \in G$ satisfying the strict identity morphism $I_{strict} : p \rightarrow r$

$$I_{strict} : p \rightarrow r \models \begin{cases} p = r, & \text{if } T(p) = simple \wedge T(r) = simple \\ p \in r, & \text{if } T(p) = simple \wedge T(r) = S_{AND} \\ p \subseteq r, & \text{if } T(p) = S_{AND} \wedge T(r) = S_{AND} \\ p \cap r \neq \varnothing, & \text{if } T(p) = S_{OR} \wedge T(r) = S_{AND} \\ r \in p, & \text{if } T(p) = S_{OR} \wedge T(r) = simple \\ r \in p, & \text{if } T(p) = S_{XOR} \wedge T(r) = simple \\ p \cap r = \varnothing, & \text{if } T(p) = S_{NAND} \wedge T(r) \in \{S_{AND}, S_{OR}, S_{XOR}\} \\ r \notin p, & \text{if } T(p) = S_{NAND} \wedge T(r) = simple \\ \top, & \text{if } T(p) = S_{NAND} \wedge T(r) = S_{NAND} \end{cases} \tag{1.1}$$

Definition 1.5.2 (Permissive Pattern Graph Matching)**.** *Permissive pattern graph matching* is a rich graph matching where a morphism for the pattern graph H is found in the target graph G given that $H \leq G$ and that for any node $p \in H$ there is a node $r \in G$ satisfying the permissive identity morphism $I_{permissive} : p \rightarrow r$

$$I_{permissive} : p \rightarrow r \models \begin{cases} p = r, & \text{if } T(p) = simple \wedge T(r) = simple \\ p \in r, & \text{if } T(p) = simple \wedge T(r) \in \{S_{AND}, S_{OR}, S_{XOR}\} \\ p \subseteq r, & \text{if } T(p) = S_{AND} \wedge T(r) = S_{AND} \\ p \cap r \neq \varnothing, & \text{if } T(p) = S_{OR} \wedge T(r) \in \{S_{AND}, S_{OR}, S_{XOR}\} \\ r \in p, & \text{if } T(p) = S_{OR} \wedge T(r) = simple \\ p \cap r \neq \varnothing, & \text{if } T(p) = S_{XOR} \wedge T(r) \in \{S_{OR}, S_{XOR}\} \\ r \in p, & \text{if } T(p) = S_{XOR} \wedge T(r) = simple \\ p \cap r = \varnothing, & \text{if } T(p) = S_{NAND} \wedge T(r) \in \{S_{AND}, S_{OR}, S_{XOR}\} \\ r \notin p, & \text{if } T(p) = S_{NAND} \wedge T(r) = simple \\ \top, & \text{if } T(p) = S_{NAND} \wedge T(r) = S_{NAND} \end{cases} \tag{1.2}$$

Of course these two are not the only identity morphisms that can be defined for the pattern matching. The implementation accepts any binary function that returns a truth value meaning that the two arguments shall be considered the same or not. Moreover the identity function can be provided for edges as well, but I skip it in the current thesis because I do not use. In the future it would be useful to define the edge morphism functions and identify the use cases that employ them.

Now that I have defined how patterns are identified in the graphs, lets take a look at more advanced applications of it. In the next section I explain how the graph isomorphism can be enacted once they are identified.

1.6 Graph matching complexity

The graph matching problem is known in computer science as *graph isomorphism problem* which is an NP-complete problem. A peculiar characteristic of such problems is that given a solution then it can be very quickly *verified* in polynomial time but the time required to find a solution increases exponentially with the size of the problem. Therefore to prove whether or not such problems can be solved quickly is one of the main unsolved problems in computer science and the performance of of algorithms solving NP-complete problems is an important issue and requires careful investigation.

To the moment no algorithm exist to solve the graph isomorphism problem in polynomial time, however the latest available algorithms such as VF2 Cordella et al. (2001, 2004) or QuickSI Shang et al. (2008) performs the task quickly when the addressed graphs are of limited size.

Next I present some estimate calculations and compare to benchmarking study of Lee et al. (2013).

The graphs used in benchmarking tests described by (Lee et al. 2013) on AIDS dataset are composed of 2–215 nodes and 1–217 edges on which VF2 algorithm performs the isomorphism problem on average in 20–25 milliseconds for sub-graphs sizing between 4–24 edges. The NASA dataset (used in the same benchmarking study) which contains 36790 graphs sizing between 2–889 nodes and 1–888 edges the VF2 algorithm performs on average in 250 milliseconds for sub-graphs of 4 edges.

To put it into the context we have to answer the questions: how big the sentence graphs are, what size are the patterns and what would be a rule of thumb estimation of performance?

According to (Koeva et al. 2012), on average, an English sentence is composed of 12-20 words(n) with about 1.6 clauses per sentence . The parse graph of an average English sentence is a tree or very close to a tree whose number of nodes is within the limits between $n+1$ and $2n-1$ for a n number of leaf nodes (in our case the words). So for a sentence of 20 words the parse tree would be maximum 39 nodes.

Lets assume the size of a sentence is ten times the average i.e. 200 words and a maximum estimate of 399 nodes in the parse tree. The patterns used in current work are 1–5 edges. Overall in the parsing algorithm, the graph matching is mostly applied at the clause level which on average in English is of 6 8 words yields an average maximum of 15 nodes per parse graph which is 0.38 of the average sentence and 0.01 of the unusually big sentence. As used in the current implementation and given relatively small graphs, the performance VF2 algorithm fits well within reasonable time limits.

1.7 Rich graph matching complexity

In order to accommodate feature rich graphs (FRG), VF2 algorithm is extended to perform custom identity checks. In the original implementation two node V_1 and V_2 are said to be equal if the nodes are of simple data types (e.g. integer or string) and they carry the same value. In our case, feature structures are attached to edges and stand for graph nodes. And there are cases when two nodes, even if the have somehow different structures, to be considered the same.

I already have defined (somehow ahead) what is graph isomorphism; and that the graph matching should always be an isomorphic function. However the nodes and edges shall be rather loosely identical or they shall only be considered identical but not necessarily be so. Therefore the node and edge morphism functions shall rather be *asymmetric* or simply a morphisms without any assumptions of exact identity. Now I can define the rich graph matching as follows.

So, the functions that compare whether two node or edge are the same in fact compare their feature structures and in fact the sameness is defined in accordance with the goals of the particular task. That's why identity checking function is provided as a parameter to the rich graph matching algorithm.

How is the node and edge identity checking done (or how are the morphism functions defined) is covered in the next section. What is important to mention here is the complexity of such nuanced checks since we have discussed the complexity of VF2 algorithm only on graphs where the edges and nodes are simple data structures.

The comparison of two FS is a PTIME problem that is efficiently solvable in polynomial time. Of course, this (slightly) increase the complexity of the matching process as a whole but still this lies well within the limits of practical computability.

The current implementation of VF2 algorithm includes the custom morphism functions for nodes and edges. So far I have not encountered performance issues with it. For the future however it would be of tremendous value to know exactly how much is the original VF2 algorithm burdened by such extra checks and what are the reasonable upper limits for the node size (i.e the complexity of the node feature structure). And perform some stress testing for the whole enterprise.

1.8 Pattern-Based Operations

The patterns are searched for in a graph always for a purpose. Graph isomorphism is only a precondition for another operation, be it a simple selection (i.e. non-affecting operation) or an affecting operation such as feature structure enrichment (on either nodes or edges), inserting or deleting a node or drawing a new connection between nodes. So it seems only natural that the end goal is embedded into the pattern, so that when it is identified, also the desired operation(s) is(are) triggered. I call such graph patterns *affordance patterns* (Definition 1.8.1). Next I explain how to embed the operations into the graph pattern and how they are used in the algorithm.

The operational aspect of the pattern graph is specified in the node FS via three special features: *id*, *operation* and *arg*. The *id* feature (the same as for relative node ordering) is used to mark the node for further referencing as argument of an operation, the *operation* feature names the function to be executed once the pattern is identified and the *arg* feature specifies the function arguments if any required and they are tightly coupled with function implementation. So far the implemented operations are *insert*, *delete* and *update*. But anyone that finds appropriate can extend it with any other operations that may be useful.

Definition 1.8.1 (Affordance Graph Pattern)**.** An *affordance graph pattern* is a graph pattern that, at least one node or edge, has *operation* and *arg* features.

I say that the affordance patterns are enacted once they are tested for isomorphism in another graph and if one is found then all the defined operations are executed accordingly.

Definition 1.8.2 (Affordance Graph Enacting)**.** For an affordance graph H and a target graph G, *affordance graph enacting* is a two step operation that first performs the permissive or strict pattern graph matching and if any isomorphism graph $G_1 \subseteq G$ is identified and second for every node $p \in H$ with an operation features, executes that operation on the corresponding node $r \in G_1$ of the isomorphism.

1.8.1 Pattern-Based Node Selection

It is often needed to select nodes from a graph that have certain properties and are placed in a particular configuration. This operation is very similar to the *atomic graph query* defined in 1.2.1. The main difference is ability to specify that the node is a part of the a certain structural configuration which is not possible via the atomic query.

For example let's say that we are interested in all nodes in a dependency graph that can take semantic roles specifically subjects, and complements of the clause. For the sake of simplicity example I exclude prepositional phrases and embedded clauses that sometimes can also take semantic roles. The pattern identifying such nodes looks like the one in Figure 1.15. It selects all the nodes that are connected via *nsubj*, *nsubjpass*, *iobj*, *dobj* and *agent* edges to a VB node.

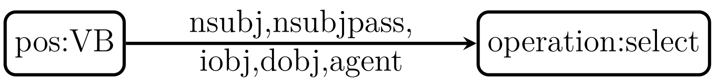


Fig. 1.15 Graph pattern that selects all the nodes that can receive semantic roles

1.8.2 Pattern-Based Node (FS) Update

There are other cases when the FS of the nodes needs to be updated either by adding or altering a feature value. This can be achieved via *pattern-based update* operation. For example, consider the example analysis 1.2 and the task to assign *Agent* feature to the subject node and *Possessed* feature to the complement. PG depicted in figure 1.16 fulfils exactly this purpose.

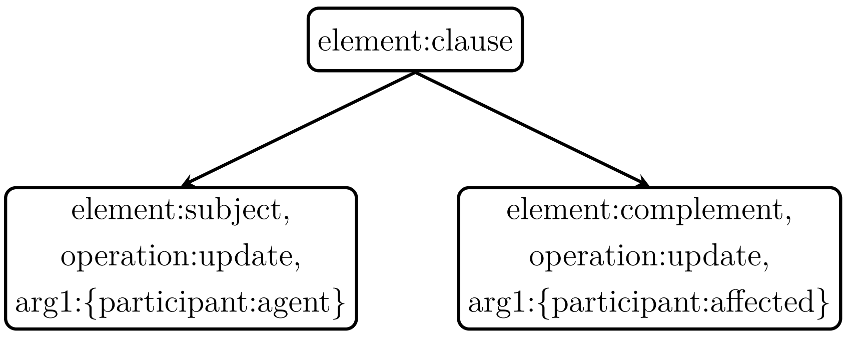


Fig. 1.16 Graph pattern for inserting the agent and affected participant role features to subject and direct object nodes.

Consider the very same pattern, but applied to a sentence in the Table 1.3. The clause has two complements and they are by no means distinguished in the pattern

class:clause				
element:subject	element: main verb	element:complement		element:adjunct
He	gave	the	cake	away.

Table 1.2 MCG with a transitive verb

graph. When such cases are encountered the PG yields two matches, (each with another complement) and the update operation is executed to both of the complements. To overcome such cases from happening PG allow defining *negative nodes*, meaning that those are nodes that shall be missing in the target graph.

For example to solve previous case I define the PG depicted in figure 1.17 whose second complement is a negative node and it is marked with dashed line. This pattern is matched only against clauses with exactly one complement leaving aside the di-transitive ones because of the second complement.

class:clause				
element:subject	element: main verb	element:complement	element:complement	
He	gave	her	the	cake.

Table 1.3 MCG with a di-transitive verb

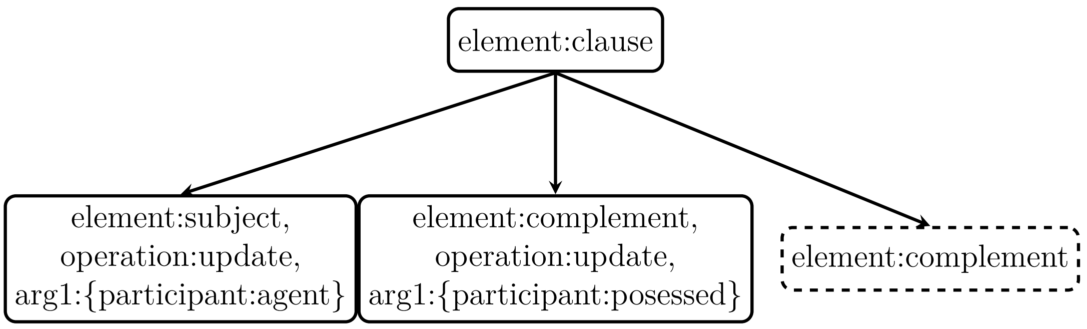

element:clause
element:subject, operation:update, arg1:{participant:agent}
element:complement, operation:update, arg1:{participant:posessed}
element:complement

Fig. 1.17 PG for inserting agent and possessed participant roles to subject and complement nodes only if there is no second complement.

The current implementation of matching the patterns that contain negative nodes is performed in two steps. First the matching is performed with the PG without the negative nodes and in case of success another matching is attempted with the negative nodes included. If the second time the matching yields success then the whole matching process is unsuccessful but if the second phase fails then the whole matching process is successful because no configuration with negative nodes is detected.

For the sake of explanation I call the pattern graph with all the nodes (turned positive) *big* and the pattern graph without the nodes marked negative *small.* So then, matching a pattern with negative nodes means that matching the *big* pattern (with negative nodes turned into positive) shall fail while matching the *small* one (without the negative nodes) shall yield success.

1.8.3 Pattern-Based Node Insertion

In English language there are cases when an constituent is missing because it is implied by the (grammatical) context. These are the cases of Null Elements treated in the Chapter **??**.

(4) Albert asked [∅ to go alone].

Consider the Example 4. There are two clauses: first in which Albert asks something and the second where he goes alone. So it is Albert that goes alone, however it is not made explicit through a subject constituent in the second clause. Such implied elements are called *null or empty constituents* discussed in detail in the Section **??**. The table 1.4 provides a constituency analysis for the example and the null elements (in italic) are appended for the explicit grammatical account. In the Section **??** I offer the grammatical account of the graph patterns that insert these null elements into the parse graphs (so in fact extensively using the pattern based node insertion treated here).

class:clause					
element: subject	element: main verb	element: complement, class:clause			
		element: subject	element: main verb		element: adjunct
Albert	asked	*Albert*	to	go	alone.

Table 1.4 The constituency analysis that takes null elements into consideration

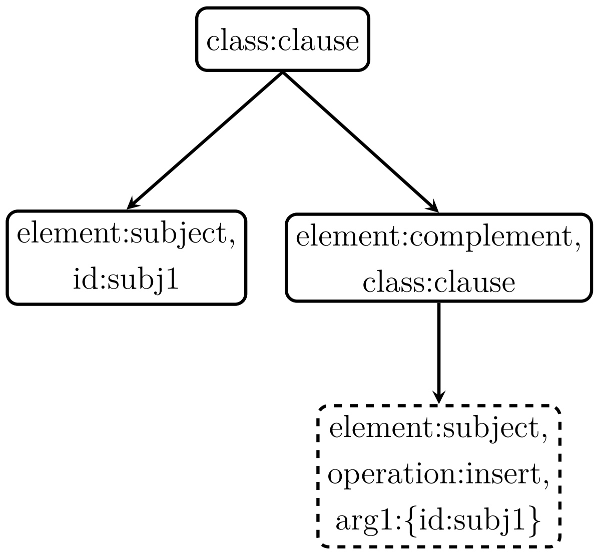

class:clause
element:subject, id:subj1
element:complement, class:clause
element:subject, operation:insert, arg1:{id:subj1}

Fig. 1.18 A graph pattern to insert a reference node

To insert a new node the, PG needs to specify that (1) the inserted node does not already exist, so it is marked as negative node, (2) specify *operation:insert* in the FS of the same and (3) provide id of the referenced node as FS argument (arg1) if one shall be taken.

In operational terms, the insertion operation means that the whole pattern will first go through a matching process. If there is a match then the new node is created. A peculiar thing about the created node is that it may keep a reference to another node or not. In our example it does keep a reference to the subject of dominant clause. If so, then all the features of the referee node are inherited by the new node. And if any are additionally provided then the new node overrides the inherited ones.

This section concludes our journey in the world of graph patterns, isomorphisms and graph based operations. Leaving only one more important data structure to cover: the system networks.

1.9 Systems and Systemic Networks

In the Section **??** I present the basic definition of System and System Network and the notations as formulated in the SF theory of grammar. In this section I formalise them in terms of what may be represented and instantiated in computational terms. In addition I cover few more useful concepts for implementation of system networks applied to enrichment of constituents with systemic features.

First I would like to introduce abstract concept of *hierarchy* defined in a computer scientific way by Pollard & Sag (1987). This is a formal rephrasing of Definition **??** that Haliday provides.

Definition 1.9.1 (Hierarchy). A hierarchy is finite bounded complete partial order $(\mathit{\Delta}, \prec)$.

The next concept that required higher order of formalization os that of a System first established in Definition **??**. For precision purposes, this one has a narrower scope without considering the system networks or precondition constraints which are introduced shortly afterwards building upon current one.

Definition 1.9.2 (System). A *system* $\Sigma = (p, C)$ is defined by a finite disjoint set of distinct and mutually defining terms called a *choice set* C and an *entry condition* p establishing the delicacy relations within a system network; subject to the following conditions:

1. the choice set is a S_{OR} or S_{XOR} conjunction set.
2. the entry condition is a S_{OR}, S_{XOR} or S_{AND} conjunction set.
3. $$\infty > size(C) \geq \begin{cases} 2, & \text{if } T(C) = S_{XOR} \\ 3, & \text{if } T(C) = S_{OR} \end{cases}$$

There is a set of functions applied to system: $label(\Sigma) = l$ is a function returning the system name, $choices(\Sigma) = C$ is a function returning the choice set, $precondition(\Sigma) = p$ is a function returning the entry condition, and the $size(\Sigma)$ return the number of elements in the system choice set.

Definition 1.9.3 (Systemic delicacy). We say that a system S_1 is more delicate than S_2 denoted as $S_1 \prec S_2$ if

1. both system belong to the same system network: $S_1, S_2 \in SN$
2. there is at least a feature but not all of S_1 which belong to the entry condition of S_2

Systems are rarely if ever used in isolation. SF grammars often are vast networks of interconnected systems defined as follows.

Definition 1.9.4 (System Network)**.** A *system network* $SN = (r, SS)$ is defined as a hierarchy within set of systems SS where the order is that of systemic delicacy where:

1. S_i is an arbitrary system within the hierarchy $S_i \in SS$
2. $r \in S_i$ is the unique root of the system network with empty precondition $precondition(r) = \varnothing$
3. $p_i = precondition(S_i)$ the entry condition of system S_i.
4. $\tau : f \times S_i \rightarrow S_j$ a transition function from a feature $f \in precondition(S_i)$ to a less delicate system $S_j, f \in choices(S_j)$. We say that $S_j \prec S_i$

subject to the following conditions:

1. $\forall x \in \cup\{P_i | \forall P_i \in SN\}, \exists y \in \cup\{choices(S_i) | \forall S_i \in SN\} : x = y$ every precondition value is among the choice values
2. $\forall x \in \cup\{P_i | \forall P_i \in SN\}$ there is a path π (i.e. a sequence of systems) such that $\tau(x, \pi) = r$ (ensuring the connectedness of entire systemic network and a unique root)
3. $\nexists x \in \cup\{P_i | \forall P_i \in SN\}$ and $\nexists \pi$ such that $\exists S_j = \tau(x, \pi)$ and that $S_j \in \pi \vee x \in values(S_j)$ (ensuring the system network is no cyclical)

Now you may ask a pertinent question: what is the basis on which is the systemic selection made? To answer it I must first introduce two types of constraints. First, The systems are interconnected with each other by a set of preselection (entry) conditions forming systemic networks (Definition 1.9.4). Second, is an aspect not always mentioned in the SFL literature, the systemic *realisation statements* which are shaping the context where the system is applied. These aspects are covered in Section 1.10 talking about execution of system networks.

The notation for writing system networks from (Halliday & Matthiessen 2013) uses colon (:) to symbolize entry condition leading to terms in systems, slash (/) for systemic contrast (disjunction) and ampersand (&) for systemic combination (conjunction). So a sample network will be written as follows:

(5) $\varnothing : i_1 / i_2 / i_3$

(6) $i_1 : i_4 / i_5$

(7) $i_2 \ \& \ i_4 : i_6 / i_7$

However in this thesis we need to account for the disjunction type and system name. So we adopt a slightly different notation of three slots separated by colon (:) where the first slot signifies the system name, second the set of system features and the third is the entry condition. Examples 8 to 10 show three systems definitions (without selection functions i.e. no realization statements).

(8) $S_1 : OR(i_1, i_2, i_3) : \varnothing$

(9) $S_2 : XOR(i_4, i_5) : OR(i_1)$

(10) $S_3 : XOR(i_6, i_7) : AND(i_2, i_4)$

The system network can be represented as a graph where each node is a system and edges represent precondition dependencies. All system features must be unique in the network i.e. $\forall S_1, S_2 \in SN : choice\_feature\_set(S_1) \cap choice\_feature\_set(S_2) = \varnothing$ and there must be no dependency loops in the system definitions.

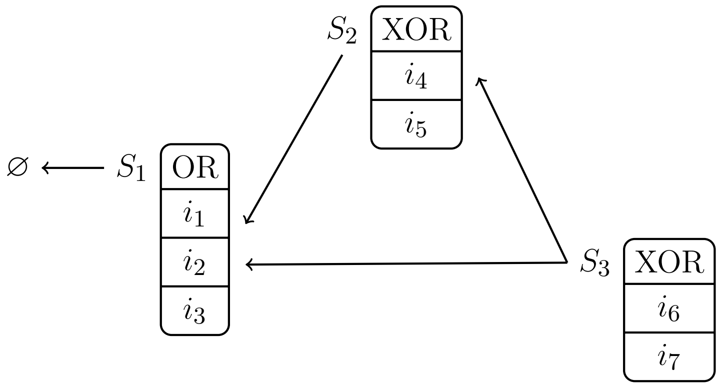


Fig. 1.19 Example System Network presented as graphs

In a systemic network SN where a system S_l depends on the choices in another system S_e (i.e. the preconditions of S_l are features of S_e) we call the S_e and *early(older) system* and the S_l a *late(younger) system*. This is just another way to refer to order systems according to their delicacy but applying this ordering to execution of systemic selection.

When the features are selected from systems within a network they form a path. It is often useful to check whether a set of arbitrary features belong to a *consistent* and *complete selection path*. Next I introduce a few concepts useful in addressing this task.

First a system network can be reduced to a graph of features called feature network (Definition 1.9.5 sometimes referred to as *maximal selection graph*) interconnected by system entry conditions.

Definition 1.9.5 (Feature Network)**.** We call *Feature Network* $FN(N,E)$ a directed graph whose nodes N are the union of choice sets of the systems in the network and edges E connect choice features with the entry condition features. Formally it can be expressed as follows:

1. $N = \bigcup choices(\Sigma_i)$ where $\Sigma_i \in SN$ for $0 < isize(SN)$
2. $E = \{(f_m, f_n)\}$ where $f_m \in choices(\Sigma_i), f_n \in precondition(\Sigma_i)$

The Feature Network in fact is an expansion of the System Network. The former is a network of interconnected features while the latter a network of systems.

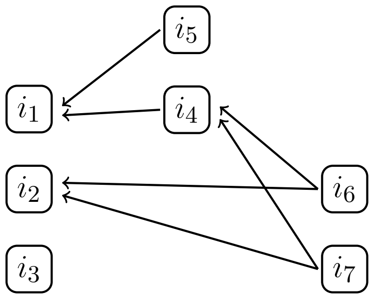


Fig. 1.20 Example Feature Network

Definition 1.9.6 (Selection Path)**.** A *Selection Path* $SP(N,E)$ is a connected sub-graph of the Feature Network representing system network instantiation through choice making traversal.

Definition 1.9.7 (Complete Selection Path)**.** A *Complete Selection Path* is a selection path starting from the network root and ending in one of the leafs.

We use terms related to age to underline order in which systems activated i.e. older systems must be chosen from before younger ones.

Definition 1.9.8 (System Network Instance)**.** A *System Network Instance* SNI of a constituent node n is a directed graph representing the union of all Complete Selection Paths applicable to a constituent.

Let's come back to Figure 1.20. As you can notice this is a handy device for efficiently checking the path completeness (whether the path is from head to tail of a feature network), consistency with respect to the order of elements (whether such a path exits). There is one aspect that cannot be checked in feature network and it is the conjunctive entry conditions which require that both system networks precede any choice in the current one. In other words, a conjunctive entry states that two paths merging into one and they can only be checked in isolation as two distinct paths, which happen to share a common portion. This shorcoming will be dressed in future work.

In this section there were mentions to selection, instantiation and traversal processes but no specific definition were provided. Next, let's turn our attention towards the system network instantiation through traversal and selection.

1.10 Systemic Network Execution

Every node from a constituency graph is enriched with feature selections grammatically characterising it. This is an important stage in the parsing algorithm discussed in Section **??**. The enrichment stage is in fact system network instantiation and ascription of complete selection paths to each constituent node .

Executing a system network is an incremental process that builds selection paths by making choices in the system networks. There are two ways to *instantiate* (or execute) a system network: either by *forward activation* or *backward induction* processes which both imply a different order of network traversal.

When it comes to traversing system networks and making choices there is a specific mechanism responsible for this instantiation process. The *choice makers* are selector functions associated to (some) systems. Selector functions implement realization statements corresponding to a system S_i and represents the instantiation mechanism turning the generic set of alternative choices into a concrete choice for a specific context.

Each node in a constituency graph carries features whose names and values are constrained to the set of systems defined in the grammar. In this sense, systems represent constraint definitions for what features may be used and what values those features can take. The algorithm has to evaluate these constraints in order to select the set of relevant features for a given constituent. Traversing system by system within the systemic network, with a known previously selected set of features and a given syntagmatic structure a selector function is executed to make the systemic choice.

Definition 1.10.1 (Selector Function)**.** A *selector function* $\sigma_{ctx} : S \to R$ is defined from a system S to a feature structure R within a given context ctx where:

1. the context $ctx = (G, fn)$ is a binary tuple of a constituency graph G and a focus node $fn \in G$ belonging to it

2. *preselection feature set* (PFS) is the already assigned set of features to the focus node $pfs = featureSet(fn)$

3. $size(R) \in \{0, 1\}$ meaning that there is either no choice made and an empty feature structure is returned or there is a choice made and a feature structure is returned with one feature bearing values from the system choice set

subject to the following condition:

1. if $size(R) = 1$ then for the only $f_i \in R$ it holds that $att(f_i) = name(S) \wedge val(f_i) \subset choices(S) \wedge val(f_i) \neq \varnothing$

If the PFS is an *OR set* then it requires that any of the features (at least one) must be in a Selection Path (Definition 1.9.6). If the PFS is an *AND set* then it requires that all of the features must be in a Selection Path.

1.10.1 Forward Activation

Forward activation is a process that enables systems to be executed (chosen from by selection function) only after choices from an older system has been already added to a selection path. In other words the selection path is constructed from older to younger systems/features.

We say that a system S_y *activates* another system S_o if and only if $\forall S_o, S_y : S_o < S_y, precondition(S_y) \cap choices(S_o) \neq \varnothing$. Activation process is the process that ensures advancement from an older to a younger system. This implies checking and ensuring entry condition is satisfied and executing the selection function. If the entry condition of the younger system is simple then the choice in the old system suffices, however if the entry condition is a complex conjunction, then first the older sibling systems have to be selected from before entering the younger one.

Algorithm 1: Forward Activation Algorithm

```
input  : sp (current selection path), sn (system network), node (constituent),
         cg (constituency graph)
1 def forward_activate(sp, sn, node, cg):
2     for system in sn systems activated by the last sp feature:
3         get choice set by executing system selection function (given system, node,
          cg)
4         append sp by the choice set
5     if sp has changed:
6         forward_activate (updated sp, sn, node, cg)
```

Algorithm 1 outlines how the forward activation is executed recursively. The systems that are active at a particular moment of the depend on the configuration of the *selection_path*. *activated_systems* function returns a set of systems from the system network whose preconditions are satisfied and their choices are not in the selection path (or the system has not yet been executed) $\forall S \in SN : precondition(S) \subset selection\_path$, $choice\_set(S) \cap selection\_path = \varnothing$.

For each activated system, its selector function is executed returning a selection set. The result selection is is used to extend the the selection_path thus potentially fulfilling preconditions of younger systems. If the path has been changed then the same procedure is applied recursively to the updated path until no more changes are done to the selection_path.

1.10.2 Backwards Induction

Backwards induction is a process opposite to forward activation. If a system is executed yielding a selection set then the preconditions of this system are induced as valid selections in the older systems defining those precondition features, and so on until a system is reached with no preconditions.

Algorithm 2: Naive Backwards induction

```
input : sp (current selection path), sn (system network), node (constituent),
        cg (constituency graph)
1 def backwards_induction_naive(sp, sn, node, cg):
2     for system in sn systems preconditioning selection sp features:
3         get choice set by executing system selection function (given system, node,
           cg)
4         for induced_system in dependecy_chain(act_sys,sn):
5             choice_set.add( precondition_set(induced_sys) )
6         selection_path += create_selection_path_from(choice_set)
7     return selection_path
```

The naive approach to is represented in Algorithm 2 which executes the selection functions of leaf systems and the yielded selections induce choices in the older systems through the precondition chain down to the oldest systems of the network.

So for example if SYNTACTIC-TYPE system in Figure 1.21 is executed and yields *verbal-marker* feature then the Algorithm 2 will add to the selection path the chain $negative \rightarrow interpersonal \rightarrow syntactic \rightarrow verbal - marker$.

This approach works very well in classification networks or networks covering a concise vocabulary such as determiners or pronouns. Such network has selection functions on the leaf systems only. However if in the middle of the selection path there are systems with selection functions then the there may exist a conflict between what is induced through precondition of younger systems and what is yielded by the selection function.

In fact confronting the preconditions with selection function is a good technique to verify whether the SN is well constructed. Following the previous example let's imagine that INTERPERSONAL-TYPE system has it's own selection function and it yields the *morphological* feature same time when the *verbal-marker* is selected in the SYNTACTIC-TYPE. Since the precondition of the latter system is the selection of *syntactic* feature, then we have a mismatch in either the way systems are constructed and the precondition of the latter system needs to be changed or the selection function is poorly implemented in the former system.

The Algorithm 3 implements the verification of whether the induced features match those from the selection function.

Algorithm 3: Backwards Induction with verification mechanism

```
def backwards_induction_verified(list_of_leafs, sn, constituent, mcg):
    for act_sys in list_of_leafs:
        choice_set = execute_selection_function(act_sys)
        if choice_set $\neq \varnothing$:
            induced_system_set = find_dependent(act_sys,sn)
        for induced_sys in induced_system_set:
            ind_choice_set = selection_function(induced_sys)
            minimal_valid_set = precondition_set(induced_sys) $\cap$
              choice_set(act_sys)
            if minimal_valid_set $\subseteq$ ind_choice_set:
                selection_path += create_selection_path_from(choice_set)
            else:
                raise Execption: The precondition set different from selection
                  function result
        backwards_induction_verified(induced_system_set, sn, constituent,
          mcg)
    return selection_path
```

It is a recursive algorithm that executes a system S_1, and also the systems $S_2..S_n$ which S_1 depends on an then verifies if $\forall S_i \in dependent\_systems(S_1)$: $precondition\_set(S_1) \cap choice\_set(S_i) \subseteq selection\_function(S_i)$

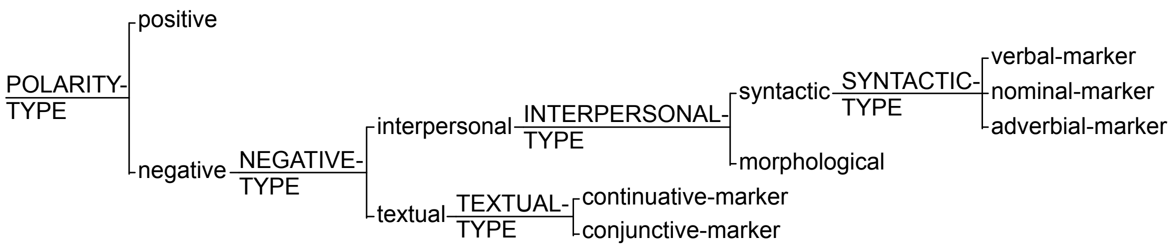


Fig. 1.21 Polarity System

Take for instance the POLARITY system in Figure 1.21. Its default selection is *positive* feature unless there is a negative marker. So we must asses NEGATIVE-TYPE system to resolve POLARITY system. But NEGATIVE-TYPE also must be postponed because we do not know if there is a negative marker unless we run tests for each marker type (i.e. presence of a “no” particle, negative subject or adjunct etc.). So we postpone selection decision and activate further the INTERPERSONAL-TYPE and TEXTUAL-TYPE systems and base the assessment on the selections yielded by the latter two systems. The same story is with INTERPERSONAL-TYPE which can make selections based on what SYNTACTIC-TYPE system yields. If SYNTACTIC-TYPE and TEXTUAL-TYPE systems yield no selection then we return recursively to INTERPERSONAL-TYPE and to NEGATIVE-TYPE and yield no selection in those systems as well. However if, for instance, *verbal-marker* is detected in the clause then the *syntactic* feature is yielded by the INTERPERSONAL-TYPE and *interpersonal* by the NEGATIVE-TYPE and thus *negative* is yielded by the POLARITY-TYPE.

Moreover the negative markers can be of various types and more than one can occur simultaneously without any interdependence between them so the algorithm needs to check presence of every type of negative marker i.e. verbal, nominal adverbial, conjunctive and continuative markers.

That being said the intermediary systems and features i.e. interpersonal, textual, syntactic, morphological are there for the classification purpose only and do not carry any particular algorithmic value making the network from Figure 1.21 reducible to the one in Figure 1.22.

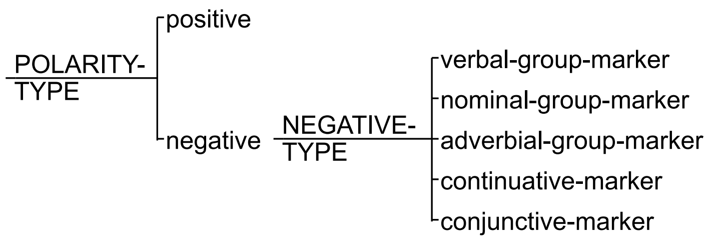


Fig. 1.22 Condensed Polarity System

Execution of system networks is subject to constituent *enrichment phase* of the parsing algorithm. Reducing the POLARITY network to the one in Figure 1.22 would lead to loss of information which may be relevant for choice-making in other systems (e.g. MODALITY) so it is useful to expand the selection set with dependent features to achieve feature rich constituents.

1.11 Discussion

This chapter describes the elemental data structure and the kinds of operations that current implementation applies to generate the SFG parse structures. It lays down the foundations for next chapter which focuses on the parsing pipeline and algorithms.

A central theme covered here are the graphs and graph patterns. They play the key role in identifying grammatical features in dependency and constituency structures. They are also excellent candidate for expressing *systemic realization rules*.

Robin Fawcett recurrently emphasises the role of realization rules in the composition of system networks. He often stresses "no system networks without realization rules". They are important because they formally express ways in which a feature is identified or realised. It is the *instantiation* process that in Halliday's words "is the relation between a semiotic system and the *observable* events or 'acts' of meaning" (Halliday 2003: emphasis added). The realisation rules for a systemic feature are the statement of operations through which that feature contributes to the structural configuration (that is being either generated or recognised) (Fawcett 2000: p.86).

It is not easy however for linguists and grammarians to provide such statements for the systemic features. Doing so means an explicit formalisation of grammar on top of charting the systemic composition and dependencies which is already a challenging task in its own. The realisation rules most of the time remain in the minds of the interpreters who can recognise a feature when it occurs. Adding the formal specification of the realisation rule requires tools for consistency checking with respect to the rest of the grammar and large corpus query tool to test various rule hypotheses.

Moreover the expression of rules is proposed in terms of atomic operations such as lexify, preselect, insert, order, etc. Which may not always be fully transparent to the grammarian. Expressing realization rules as operations contextualised in fragments of parse structure is a promising way to ease the grammar authoring process. They could then be used directly by the parser to recognise such structures making the corpus annotation and grammar construction an in-parallel evolving process.

The data structures and operations described in this chapter can be a suitable approach to address the problem of missing realisation rules from the system networks. To do so however requires creation of a system network authoring tool (such as the one available in UAM Corpus Tool (O'Donnell 2008)) which besides systemic network editor should contain also a graph pattern editor allowing association of graph patterns to systemic features and .

In current parser the pattern graphs are represented as compositions of Python dictionaries and lists such as the one below.

```
{
    NODES: {
        "cl": [
            {C_TYPE: 'clause',
             VOICE: ACTIVE},
            {CONFIGURATION: ['two-role-action', ['Ag', 'Ra', 'Cre']], }],
        'pred': [
            {C_TYPE: [PREDICATOR, PREDICATOR_FINITE], },
            {VERB_TYPE: "main", PROCESS_TYPE: 'two-role-action'} ],
        'subj': [
            {C_TYPE: SUBJECT, },
            {PARTICIPANT_ROLE: 'Ag'}],
        'compl1': [
            {C_TYPE: [COMPLEMENT, COMPLEMENT_DATIVE], },
            {PARTICIPANT_ROLE: 'Ra'}],
        'compl2': [
            {C_TYPE: [COMPLEMENT, COMPLEMENT_ADJUNCT, ], },
            {PARTICIPANT_ROLE: 'Cre'}],
    },
    EDGES: [
    ['cl', 'pred', None],
    ['cl', 'subj', None],
    ['cl', 'compl1', None],
    ['cl', 'compl2', None],]
}
```

This Python dictionary contains two top keys: NODES defined as with node identifiers each associated with a set of systemic features and EDGES defined as a list with three tuples of source, target and eventually a dictionary of features. The nodes contain a list of two dictionaries. The first dictionary enlists the features that the backbone structure should already carry, and against which the pattern matching is performed. The second dictionary contains the set of features that the node shall receive in case of a successful match of the entire pattern.

Writing such structures is cumbersome and requires in depth knowledge of the parser and employed system networks therefore the need for an editor is even higher. Unfortunately building such an editor is out of the scope of the current work and is among the priorities in the future developments just as switching to better technology for working with graphs such as Semantic Web suite of tools. This and other future work are described in the Section **??**.

In the next chapter I describe the parsing pipeline and how each step is implemented starting from Stanford dependency graph all the way down to a rich constituency systemic functional parse structure.

References

Cordella, L P, P Foggia, C Sansone & M Vento. 2001. An improved algorithm for matching large graphs. *3rd IAPRTC15 workshop on graphbased representations in pattern recognition* 219(2). 149–159. doi:10.1.1.101.5342. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.5342{&}rep=rep1{&}type=pdf>.

Cordella, L P, P Foggia, C Sansone & M Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. doi:10.1109/TPAMI.2004.75. <http://www.ncbi.nlm.nih.gov/pubmed/15641723>.

Fawcett, Robin. 2000. *A Theory of Syntax for Systemic Functional Linguistics*. John Benjamins Publishing Company paperback edn.

Halliday, Michael A. K. 2003. Systemic theory. In Michael A. K. Halliday & Jonathan J. Webster (eds.), *On language and linguistics. Volume 3 of collected works of M.A. K. Halliday*, 490. New York: Continuum.

Halliday, Michael A.K. & Christian M.I.M. Matthiessen. 2013. *An Introduction to Functional Grammar (4th Edition)*. Routledge 4th edn.

Koeva, Svetla, Borislav Rizov, Ekaterina Dimitrova, Tarpomanova Tsvetana, Rositsa Dekova, Ivelina Stoyanova, Svetlozara Leseva, Hristina Kukova & Angel Genov. 2012. Bulgarian-english sentence-and clause-aligned corpus. In *Proceedings of the second workshop on annotation of corpora for research in the humanities (acrh-2)*, 51–62.

Lee, Jinsoo, Wook-Shin Han, Romans Kasperovics & Jeong-Hoon Lee. 2013. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the 39th international conference on Very Large Data Bases* 133–144. doi:10.14778/2535568.2448946. <http://dl.acm.org/citation.cfm?id=2448936.2448946>.

O'Donnell, Mick. 2008. The UAM CorpusTool: Software for Corpus Annotation and Exploration. In Bretones Callejas & Carmen M. (eds.), *Applied linguistics now: Understanding language and mind*, vol. 00, 1433–1447. Universidad de Almería.

Pollard, Carl & Ivan Sag. 1987. *Information-Based Syntax and Semantics*. CSLI.

Shang, Haichuan, Ying Zhang, Xuemin Lin & Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *VLDB Endow* 1(1). 364–375.