

Uppsala Chart Parser Light

Improving Efficiency in a Chart Parser

Per Weijnitz
perweij@stp.ling.uu.se

Master's thesis in Computational Linguistics
Språkteknologiprogrammet
(Language Engineering Programme)
Uppsala University · Department of Linguistics

August 2002

Supervisors:
Anna Sågvall Hein, Uppsala University
Mats Dahllöf, Uppsala University

Abstract

This report describes the design, implementation and evaluation of UCP Light, a procedural chart parser aimed at syntactic analysis. The goal of the work was to produce a faster version of its predecessor, Lisp based UCP2, in standard C. This goal was achieved, with an increase in speed by a factor of 21. The morphological functionality of UCP2 was not included in the specification for UCP Light.

Contents

Acknowledgements	iii
1 Introduction	1
1.1 Purpose	1
1.2 Outline of this report	1
2 Chart Parsing	2
2.1 Data Structures	3
2.2 Algorithm	3
2.2.1 The Fundamental Rule	4
2.3 Linguistic Data Structures	4
3 UCP - a Procedural Chart Parser	6
3.1 Tasks	7
3.2 The Fundamental Rule	8
3.3 Feature Structures	8
4 UCP Light	9
4.1 UCP Grammar Operators	10
4.2 Path Operators	12
4.3 Grammar File Instructions	12
4.4 Declaring the Symbols - Code Table Format	13
4.5 Declaring the Input - Code Strings	13
4.6 Filter Format	14
4.7 The Start Rule	14
4.8 Logging	14
4.9 Compiling UCP Light	15
5 Design and Implementation	16
5.1 Agenda, Chart and Tasks	16
5.2 Feature Structures	16
5.2.1 Symbols are Identical	19
5.3 The Rule Interpreter	19
5.4 Parsing input data	19
5.5 Macros as Templates	19
5.6 Memory Management	20
5.6.1 Simplified Garbage Collection	20
5.7 API - Application Programmer Interface	20

5.8	Output of UCP Light	21
6	Evaluation	22
6.1	Performance	22
6.2	Compatibility with UCP2	23
6.3	ANSI C Conformance	24
7	Conclusion	25
	References	25
A	Appendix	28
A.1	Executing a Task Stack	28
A.2	Creating code strings for evaluation	28
A.3	Key Data Structures	29
A.4	Example Grammar	30
A.5	Errors	30
A.6	ScarCheck-specific Extensions	31
A.7	Compatibility Test	31

Acknowledgements

Grateful thanks to all of you who have been involved, especially for your patience during this time consuming process. Anna Sångvall Hein, for supervision, guidance, and finding loads of bugs. Per Starbäck, for kindly letting me tap into his deep source of UCP knowledge. Leif-Jöran Olsson, both for constant and reliable general support, and for fixing bugs in my absence 1999. Together with Lars von Wachenfeldt they helped designing the API and assisted in technical questions. Mats Dahllöf, for encouragement and thesis writing expertise, and Anders Weijnitz for proof reading and support.



1 Introduction

This report describes the design and implementation of Uppsala Chart Parser Light. It is based on UCP2, the previous implementation of the conceptual framework of a chart parser introduced by Sgvall Hein (1980)¹. The work was carried out within the EU project SCARRIE (SCARRIE 1996)². In this project ScarCheck (ScarCheck 1999) was developed, a system for spell checking and grammar checking. The specification for UCP Light is due to Anna Sgvall Hein, Per Starbck and Leif-Jran Olsson.

There was a need for a fast, stand alone parser module in ScarCheck. A new specification was drafted for a light C version of UCP2, compatible with its syntactic grammars, but with less overhead. It was meant to be embedded in environments that provide the supplementary services, such as dictionary lookup, user interaction etc. It was implemented in C, restricted to a common standard for portability reasons.

It is taken for granted that the reader is familiar with the basic concepts of imperative programming, as this reports describes a C implementation. Some knowledge about computational linguistics and chart parsing is also presumed, although the background chapter should provide a sufficient introduction³. When the text mentions UCP, it refers to the conceptual framework. UCP2 refers to the previous implementation of UCP.

1.1 Purpose

The goal of the work was to implement a faster version of UCP2 in standard C. It was not necessary to retain the format of the input data and output data of UCP2. The goal of the evaluation of UCP Light was to see if a test corpus was equally analysed by UCP2 and UCP Light. Time consuming tasks, like the morphological capability was removed, hence making it light. To ensure portability, the code was restricted to ANSI C, a standard which is widely supported.

1.2 Outline of this report

The three chapters following the introduction give an overview of chart parsing, UCP and a description of UCP Light. The design and implementation chapter describes some aspects of the making of the software. The following chapter evaluates the compatibility with UCP2, and compares the time efficiency of the two programs. The final chapter concludes the report.

¹See further Carlsson 1982, Sgvall Hein 1983, Sgvall Hein 1987.

²LE3-4239

³See further Thompson and Ritchie 1984, Tomita 1985, Allen 1987, Gazdar and Mellish 1989, Kay 1989, Carroll 1993, Caraballo and Charniak 1998.

2 Chart Parsing

The common feature of chart parsers is using a chart to store unique states of the parsing process. The purpose for doing this is to be able to reuse the current partial or complete analysis at a later stage. Unlike backtracking parsers, a chart parser does not have to recompute anything as all intermediate results can be stored in the chart, and may be accessed directly.

- The chart data structure does not impose any particular parsing method, and is used by parsers with different types of grammars, depth of analysis and focus on efficiency.
- The degree of ambiguity of an input segment is shown by the number of completed - passive - states in the chart enclosing the segment. In the same way, it is possible for parts of the segments to be ambiguous.
- The chart is a practical data structure to use with non-deterministic algorithms, as it can handle alternatives. Non-deterministic parsing is suitable for ambiguous natural language, where it is not always clear what will be the next state in the process.
- Most chart parsers use at worst $O(n^3)$ time for an input string of n symbols. Stack based parsers may use $O(\text{time}^n)$ at worst case, exponential time. This is because a stack can only represent one particular path at a time. Backtracking means forgetting computations, which means the same computation may happen many times.
- Although the great majority of existing chart parsers are serial, the processing is best described as parallel, as each state represents a unique path in the search space. Once a state is has been reached and stored, it can be reused instead of recomputed.

There are three common ways parsers relate to grammars (see further Hellwig 1999):

- An interpreting parser uses a declarative grammar, independent of processing algorithm. The parser processes such grammars only according to the grammar formalism (meaning the grammar notation definition), not according to the contents of the particular grammar (Aho, Sethi and Ullman 1986:160).

Examples: a Recursive Transition Network (Woods 1970) is a set of interconnected, labeled finite state networks equivalent to a context-free grammar. The arcs in the networks may not only be labeled with lexical categories, but also with labels of a finite state network. Traversing an arc with a network label means recursively entering that network. This also makes the data structure modular, e.g. with an VP or NP network. See further CKY below and Shieber, Uszkoreit, Pereira, Robinson and Tyson's (1983) PATR-II.

- Procedural parsers are controlled by the actual grammar, which implements both the parsing algorithm as well as the syntactic descriptions. The grammar rules are executed like program procedures.

Examples: An Augmented Transition Network (Woods 1970) is an RTN with registers and conditional jumps. It has the power of a Turing Machine (Martin, Church and Patil 1981). Kaplan's (1973) General Syntactic Processor provides a framework of a chart and a set of operators for adding and accessing edges

and their properties, along with control structures. It also provides ways for recursion, non-determinism and controlling processes. Kaplan shows that both ATN and Kay's (1967) parser can be implemented in GSP. The procedural approach is quite similar to the one of UCP Light.

- Compiled parsers are a hybrid, where a declarative grammar is transformed into a parsing program by a parser generator (Aho et al. 1986:22).

2.1 Data Structures

The chart consists of a set of vertices, and a set of unique edges running from start and stop vertices (where start \leq stop). An edge may be either passive or active. Passive edges span well formed substrings of the input data, according to the grammar. An active edge is not fully processed, and is still considered an hypothesis. It contains a list of what has been parsed and what is left to be parsed.

```

| 1-----| 2-----| 3
.np-----
+np-----
+
.AL----- .N-----

```

Figure 2.1: A chart showing passive and active edges (. for passive and + for active). The examples in this section are the output of UCP Light.

Examples: Earley's version of the chart is a set of sets, one for each input symbol, representing the condition of the analysis at that point in the scanning of the input. The sets contain states representing (1) the grammar production being pursued, (2) a position marker in that production, showing how much of the right hand side that has been recognised so far, (3) the position in the input string where this instance of the production started and (4) syntactically allowed successors to this instance of the production.

The agenda is a data structure used to control the order of the parsing. It may be either a stack (depth-first parsing) or a queue (breadth-first parsing). There are two main approaches of how the agenda is used:

- It is used to order the evaluations of configurations between active and passive edges. In this case, the elements of the agenda are either the tasks that will perform the evaluations, or pairs of edges constituting the configurations.
- It is used to order the assertion of new edges to the chart. In this case, the elements of the agenda are new edges.

See section 2.2.1 about the fundamental rule for more details.

2.2 Algorithm

Below is a list of general parsing strategies, different ways of performing the analysis.

- Bottom-up parsing starts with the sequence of terminal nodes and finds matching rules in the grammar, reducing the right hand side of the rules to the left hand side symbol.

Example: the CKY algorithm (J Cocke and J T Schwartz 1970, Kasami 1965, Younger 1967) is a bottom-up algorithm for context-free grammars. It is passive, as it does not use active edges. It works by incrementally parsing all substrings of input length 1 to k, where k is the length of the input. When parsing at some position a, the completed substrings resulting from parsing positions $< a$ may be used. The worst case efficiency is $O(n^3)$.

- Top-down parsing means using the derivation rules of the grammar from left to right, beginning with a set of start rules (sentence rules). Non-terminals in the production, names of other grammar rules, are expanded until a sequence of terminal nodes is reached, which represent the input string.

Example: Earley's (1970) algorithm uses an active chart and is similar to the CKY algorithm and uses a top-down, left-right approach. The efficiency varies between $O(n^3)$ - $O(n)$ (time needed to parse a string of length n) depending on grammar ambiguity. With a single algorithm, Earley either matches or surpasses the contemporary parser algorithms of CKY and Knuth's LR(k) (Knuth 1965) in efficiency and grammar class coverage. It accepts any context-free grammar, ambiguous or not, and does not have any problems with left recursive grammars.

- Depth first parsing means that the left-most or right-most symbol in any production is processed first. It is suitable to combine with top-down parsing, as it will exclude rules from further expansions that do not combine with the terminal it found.
- Breadth first parsing means processing rules in the order they are created. It is suitable to combine with bottom-up parsing as constituents will build evenly across the input string.

2.2.1 The Fundamental Rule

Whenever a new edge is asserted to the chart, a configuration with the operative end of an active edge meeting the starting edge of an inactive edge may take place. The fundamental rule states that, when such a configuration occurs, an action is taken which might lead to the introduction of new edges (Kay 1989). Figure 2.2 is an extension of figure 2.1 with three configurations:

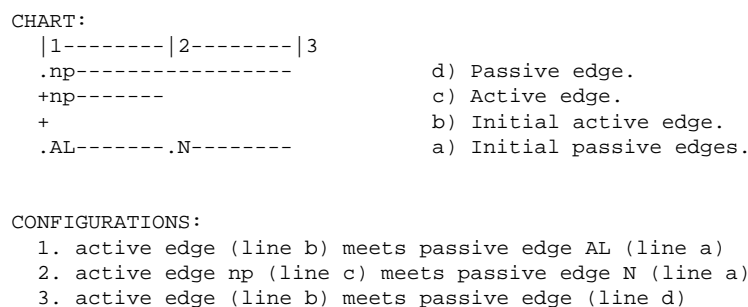


Figure 2.2: A chart with its configurations.

Initialising is done by, for each token of the input string, adding a passive edge to the chart between vertex n and $n + 1$ where n is the position of the token in the input string. For each edge added to the chart, new ones may arise due to the fundamental rule. Parsing is typically started by introducing one or more active edges.

2.3 Linguistic Data Structures

A common way to store linguistic information in the edges of the chart is to use feature structures. FSs are directed graphs. Directed Acyclic Graphs (DAGs) are commonly used, although they do not allow true value sharing. Value sharing is when two nodes share the same daughter-node (which may occur as a result of a unification). Internal references within a graph can be used to implement value sharing, as shown in figure 2.3, where $\langle * \text{ SUBJ} \rangle$ and $\langle * \text{ PRED } 1 \rangle$ both share the same value¹. Such graphs are not acyclic as a node may refer to its parent nodes.

¹Such path notation is used by UCP to specify a place in the feature structure, see section 3.

```
(* = (SUBJ = #0:(SPEC = A
          NUM = SG
          LEX = GIRL)
      TENSE = PAST
      PRED  = (ACTION = HAND
              1      = #0
              2      = #2
              3      = #1)
      #1:OBJ = (SPEC = THE
              NUM  = SG
              LEX  = BABY)
      #2:OBJ2 = (SPEC = A
              NUM  = SG
              LEX  = TOY)))
```

Figure 2.3: An example of a feature structure representing “A girl handed the baby a toy” according to some grammar.

3 UCP - a Procedural Chart Parser

UCP is procedural in that its grammar implements both the grammatical description of the input language and the parsing strategy (top-down or bottom-up). A declarative grammar only describes the input language, it is up to the parser to implement a parsing strategy. A feature of UCP is that all chart extensions are made explicitly from the grammar rules, providing a fine grained control of the chart manipulation¹.

The UCP grammars can be top-down, bottom-up or a mix of both approaches, as the parsing strategy is implemented in the grammar rules. Dictionary search, morphological analysis and syntactic analysis is handled in the same chart, using the same type of rules for the dictionary and the grammar. Top-down parsing is started by a task executing the start rule which is defined in the grammar.

The grammar rules are not unlike the functions found in imperative programming languages. They are sets of statements combined by control operators such as AND, OR and branching operators. Below is a summarisation of the UCP operators (following Sågvall Hein and Starbäck 1999):

<code>process(arg)</code>	inserts an active edge from and to the final vertex of the active edge; <code>arg</code> is the name of a dictionary or a grammar rule
<code>majorprocess(arg)</code>	inserts an active edge from and to the initial vertex of the active edge; <code>arg</code> is a grammar rule
<code>advance(arg)</code>	inserts an active edge from the initial vertex of the active edge to the final vertex of the inactive edge; <code>arg</code> is a subrule name; if it is left out, the next operation in the rule sequence will be executed
<code>store</code>	inserts an inactive edge from the initial vertex of the active edge to the final vertex of the inactive edge; it inherits its feature structure from the active edge
<code>minorstore</code>	inserts an inactive edge from the initial vertex of the active edge to the final vertex of the active edge; it inherits its feature structure from the active edge

Figure 3.1: Chart building operators.

¹Except at initialisation, when the initial edges are asserted to the chart.

Unification	<code>:=:</code>	
Equality	<code>=</code>	
Not	<code>not</code>	
Path	<code><* val1 ... valn></code>	denotes the value of the attribute specified by the path from <code>*</code> to <code>valn</code> in the feature structure of the inactive edge
	<code><* char :property></code>	denotes the value of <code>property</code> associated with the character attribute of the inactive edge in the dictionary of characters
	<code><* lem :property></code>	denotes the value of <code>property</code> associated with the lemma attribute of the inactive edge in the dictionary of lemmas
	<code><& val1 ... valn></code>	denotes the value of the attribute specified by the path from <code>&</code> to <code>valn</code> in the feature structure of the active edge
	<code><& val1 ... valn :new></code>	generates a new integer attribute in the feature structure of the active edge starting by 1 and adding 1 with every new call
	<code><& val1 ... valn :last></code>	denotes the value of the last attribute in the path specified by <code>val1</code> to <code>valn</code> in the feature structure of the active edge
Nil	<code>nil</code>	a symbolic feature value that unifies with any other value
Atom	<code>'atom</code>	a distinct symbolic feature value

Figure 3.2: Operators for test and assignment.

3.1 Tasks

Tasks are the processes of UCP. They are used to execute grammar rules in a specific context. When created, in accordance with the fundamental rule, they are put on the agenda for controlled selection. The agenda may also contain tasks representing partly processed rules that have been put there as the result of process control operators (eg `advance`, 4.1). The tasks are consecutively selected from the agenda and executed. When the agenda is empty, the parsing is finished.

Once the parsing is finished, UCP scans the chart for successful parses. In case there are passive edges spanning all vertices, those are selected. If not, it is possible to extract partial parses. The chart-scanning component of UCP, *reportchart*, uses the following algorithm:

Starting at the first vertex:

1. Find the passive edges with the longest span.
2. If the final vertex is reached, the scan is finished. Otherwise, repeat from 1 starting from the stop vertex of the previously selected edge.

A task's principal parts are a stack of rule operators to execute (derived from the grammar), a feature structure on which the process operates and a filter to avoid a combinatorial explosion. The root symbol of the active edge's FS

Sequence	(op1 , op2 , ... , opn)	boolean 'and'
Dependent disjunction	(op1 / op2 / ... /opn)	boolean 'or'
Independent disjunction	(op1 // op2 // ... //opn)	parallel processing
If-then	(if op1 then op2 else op3)	condition
Subrule	rulename('val1, 'val2, ... 'valn)	subrule call (with optional parameters)
True value	continue	always true
False value	failure	always false

Figure 3.3: Control operators.

is & and the root symbol of the passive edge's FS is *. The contents of the edges must not be changed, still it is convenient to use unification operations on the feature structures. This is why the task may have its feature structure replaced by a copy when there is a need for manipulating the FS. See section A.1 for a detailed description of how stacks work.

3.2 The Fundamental Rule

When a configuration with the operative end of an active edge meeting the starting edge of an inactive edge takes place, the filter of the active edge determines whether there will be a new task on the agenda. The filter is an expression of what root features there must or must not be in the active edges FS. The new task will take its actions from the active edge, and its feature structure from both edges.

3.3 Feature Structures

Feature structures play a central role in UCP. All linguistic data are stored as such. The main operation on FSs is unification, which is done by recursively calculating the union of the two FSs. Two unified FSs are **identical**, meaning they share the same value in memory. See figure 2.3 where the values of the paths <* SUBJ> and <* PRED 1> are identical as the result of a unification. The internal order of the attributes of an FS is not important. An arbitrary order is chosen when printing FSs.

Please observe the special nil value which is of importance when working with unification. Nil is an unspecified value of an attribute. It can be assigned explicitly by unifying a path with the reserved symbol nil, or it may occur when unifying two new paths. The unspecified value may later be instantiated by an actual value, either a simple symbol or a feature structure, if used in a unification.

4 UCP Light

UCP Light is designed mainly for syntactic analysis, relying on an external lexical component. The input consists of morpho-syntactic codes that are expanded into feature structures during initialisation. The mapping between the codes and the feature structures is defined in the code file, see section 4.4. It is also possible to do bottom-up parsing by associating rules with the codes. The rules will result in active edges, from and to the starting edge of that code.

Here is an example of an input string, consisting of morpho-syntactic codes along with additional feature structures:

```
((NNUSDB [* [lem [sym bil.nn],
              [lex [sym bil.nn.1]],
              [trglex [sym truck.nn.1]]]])
 (VBAPC [* [lem [sym vara4.vb],
            [lex [sym vara4.vb.1],
                [val va.vara]],
            [trglex [sym be.vb.1]]]])
 (AVUSIBP [* [lem [sym ny.av],
              [lex [sym ny.av.1]],
              [trglex [sym new.av.1]]]])
 (SRA [* [lem [sym stop.sr],
          [lex [sym stop.sr.1]],
          [trglex [sym stop.sr.1]]]])
```

Figure 4.1: An input string representing “Bilen är ny.” (“The truck is new.”).

Unlike UCP2, UCP Light is not designed with a user interface of its own. It is a library meant to be used by separate interfaces. There are a few example interfaces distributed with the program. If you want to parse a natural language sentence, the interface needs to be supplied with a lexical component that transforms the sentence into a code string as described in 4.5. Please note that UCP Light is case sensitive regarding all input.

There are a few changes in the grammar rule file format since UCP2:

- The LISP style is now gone. The grammar files now only contain grammar file instructions (see section 4.3).
- ‘1 is now written 1 (e.g. `<& pers> = 1`).
- `do(arg)` can be used to evaluate the argument to a value which is then invoked as a rule (e.g. `do(?1)`).
- Vertices are numbered from 0 instead of 1.
- `<& start>` is now !a, `<* start>` is now !b and `<* end>` is now !c. There is also !x which represents the number of the last vertex.
- The `:property` path operator of UCP is not implemented.

4.1 UCP Grammar Operators

In this section we say that the active edge goes from vertex A to vertex B, and the inactive one from vertex B to vertex C. Parentheses may enclose grammar constructs to control precedence.

Designation: Constant

Notation: /symbol or number

Semantics: The atomic values of feature structures.

Designation: Constant Operators

Notation: !a or !b or !c or !x

Semantics: These will expand to a numerical constant during execution. !a, !b and !c will expand to the number of vertex A, B and C respectively. !x expands to the number of the last vertex.

Designation: Retrieval (PATH)

Notation: < pathitem1 ... pathitemN >

where pathitem may be either a symbol, a number or a path operator.

Semantics: This is how to retrieve values from feature structures. All retrieval operations begin with a reserved path item, namely the ones for specifying which feature structure to work on: & for the active edge's and * for the passive edge's. Path operators are evaluated to simple symbols (symbols or numbers) in real time, see section 4.2.

Designation: Truth Values

Notation: continue or failure

Semantics: The two truth value constants. failure can break execution of a rule.

Designation: Negation

Notation: not expr

Semantics: Succeeds if and only if expr fails.

Designation: Conjunction

Notation: expr1 , ... , exprN

Semantics: expr1 to exprN are tried in sequence. If one of them fails, the rest are not tried and the whole execution fails. Otherwise, the conjunction succeeds. Syntactically, , takes precedence over / and //.

Designation: Dependent Disjunction

Notation: expr1 / ... / exprN

Semantics: expr1 to exprN are tried in sequence. If one of them succeeds, the rest are not tried and the whole disjunction succeeds. Otherwise, the disjunction fails. Syntactically, / yields precedence to , but takes precedence over //.

Designation: Independent Disjunction

Notation: expr1 // ... // exprN

Semantics: The running task is terminated, and a number of new tasks are created. The running task is duplicated into each new task, which are given expr1 to exprN as program counter (the point where processing will continue). This is similar to the way fork behaves on various systems.

Designation: Condition
Notation: `if cond then expr1 (else expr2)`
Semantics: If condition *cond* is true, execute *expr1*.
The `else` is optional, and if *cond* is false, *expr2* will be executed.

Designation: Equivalence
Notation: `fs1 = fs2`
Semantics: This succeeds if and only if *fs1* and *fs2* are equivalent feature structures.
See section 3.3.

Designation: Identity
Notation: `fs1 == fs2`
Semantics: This succeeds if and only if *fs1* and *fs2* are identical feature structures.
See section 3.3.

Designation: Unification
Notation: `fs1 ::= fs2`
Semantics: See section 3.3 for information on unification. The arguments become identical after the operation (if successful).

Designation: Forward Motion
Notation: `advance` or `advance (rule name)`
Semantics: The running process is terminated. A new active edge is stored in the chart. This edge spans from vertex A to vertex C, and contains control information about what remains to be done in the terminated process. Processing then continues from the termination point. The old task's feature structure is reused in the new task.
The second form is equivalent to: `advance , rule name`

Designation: New Process
Notation: `process (rule name)` or `process (rule name , path expr)`
Semantics: An active edge is stored from and to vertex B. If *path expr* is given, `<&>` will be set in the new edge, and its value is the value of *path expr*. If no *path expr* is given, no feature structure will be stored in the edge. The new edge will have *rule name* as its action. This operator always succeeds.

Notation: `majorprocess (rule name)` or `majorprocess (rule name , path expr)`
Semantics: Like `process` with the exception that the active edge is stored to and from vertex A.

Designation: Rule Call
Notation: `name`, `name (arg1 (..., argN))` or `do (?1)`
Semantics: This will process the grammar rule *name* before continuing with the next instruction. The optional arguments are sent call-by-name, and may consist of rule names, constants, path expressions or formal parameters. Formal parameters are written `?n` where *n* is an integer referring to the *n*-th parameter in the call to the current rule. If a formal parameter occurs in a path expression, its actual parameter has to be a path expression itself. It will be flattened so that the formal parameter will expand to the elements of the actual parameter. In the case of `do`, the argument is first evaluated to a value which is then invoked as a rule.

Designation: Storing Data
Notation: `store` or `store (path expr)`
Semantics: A passive edge is stored in the chart from vertex A to vertex C.
 If no argument is given, `<*>` will be set to the value of `<&>`, otherwise the value will be the value of `path expr`.
Notation: `minorstore` or `minorstore (path expr)`
Semantics: Like `store` except the new passive edge will span from vertex A to vertex B.

4.2 Path Operators

A path operator, used in path expressions as defined above, is evaluated during execution and will expand to an attribute. Currently there are two path operators, `:new` and `:last`. `:new` introduces a new numerical attribute in the current attribute/value list. If no numerical attribute is found, it will be 1, otherwise it will be the highest numerical attribute value + 1. `:last` will retrieve the highest numerical attribute value found. If no such exists, a error will break execution.

Please observe that `:new` will make difference in which order the right and left arguments are given to the unification operator. The operator is implemented so that the left argument is evaluated before the right one. An example:

```
before: (& = nil)
<& :new> ::= <& :last>
after: (& = (1 = nil))
```

whereas:

```
before: (& = nil)
<& :last> ::= <& :new>
```

would result in an error as the left argument will not be able to find a numerical attribute. UCP Light only supports `:new` and `:last`. UCP has additional operators related to morphology.

4.3 Grammar File Instructions

The grammar file format is no longer LISP based. A grammar file may consist of the operators listed below. See section A.4 for an example grammar and code file.

defrule `name filter { body }`

Starts a new rule declaration. See section 4.6 for a description of the filter syntax, and section 4.1 for the rule body format. Grammar rules may call each other recursively. They may take arguments. In addition to subroutine calls, the rules contain primitives for chart manipulation and task control. There are sequencing constructs and conditionals. A truth value register and a control stack is necessary for the interpretation of the rules. The tasks may fork, which makes it necessary that a control stack is the starting point for every task, instead of just a rule.

load ```file.grm```

This operator will point the parser to `file.grm`, and parse its contents before continuing with the next line of the current file. Default search path is the current directory (i.e. the directory containing the current grammar file). If `file.grm` is located in another directory, and contains a `load` operator, the default search path will be that directory.

phrasewords on/off

It is possible to choose whether the words which form a phrase in the code string (see section 4.5) should generate edges at chart initialisation. If set to off, only the phrase code will generate edges.

comment

Adds a comment to the file.

\ single character

This is a quote operator which instructs the lexical scanner to accept the following character as part of a string. This is necessary when the character is either an operator itself, or it is not included in the scanner's character set. For example: `<& reg subj \:new> ::= <*>`. Normally `:new` would generate a numerical path, see section 4.1, but here it is interpreted as any other path item.

4.4 Declaring the Symbols - Code Table Format

The symbols that are used to form the code string mentioned above have to be declared in a table¹. This is for two reasons, first the parser need to know what linguistic features the symbol represents (these will eventually be put in a feature structure). Second, rules can be associated with each symbol. Each of these result in new active edges, from and to the starting point of the symbol, with the specified rule on the stack.

The entries of this code table file are organised into clusters by word category. A BNF description of such clusters follows (upper case strings denote BNF terminals, lower case strings denote BNF non terminals and 'NL', ':' etc denotes a new line character, the character ':' etc):

```
cluster    :- ':' WC attrlist 'NL' entries
entries   :- entry | entries entry
entry     :- CODE defline | CODE '+' deflines
deflines  :- defline | deflines defline
defline   :- vallist '!' rulelist 'NL'
attrlist  :- ATTR | attrlist ATTR
vallist   :- VAL | vallist VAL
rulelist  :- RULE | rulelist ',' RULE
```

Figure 4.2: A BNF representation of the syntax of the code file.

NB the number of elements in `attrlist` and `vallist` must be the same. This is because when the chart is initiated, the passive edge based on a code entry will have a feature structure based on the ATTR - VAL pairs of that entry. The active edge will use the `rulelist` data: for each rule that is listed, an active edge will be created. The WC is special, it is the value of the reserved attribute `word.cat` which will be added to the feature structures of the passive edges, along with the ATTR - VAL pairs. Use # to comment the rest of a line out. CODE is merely the name of the symbol. An example:

```
# NN = Noun
:NOUN  gender numb  form  case
#####
NNUPIB utr    plur  indef basic !np_noun # i.e. mattor (carpets)
NNUPIG utr    plur  indef  gen  !np_noun # i.e. mattors (carpets')
```

4.5 Declaring the Input - Code Strings

The code string consists of a parenthesis containing morpho-syntactic codes representing the words. The parenthesis contains elements:

¹The code table format was defined in the Scarrie project.

- (CODE) simply means CODE.
- #(LEMMA CODE) means CODE and passes on a lemma string into the feature structure of the code (at `<* lem>`).
- (CODE [* [lem LEMMA]]) means CODE and unifies the supplied feature structure with the one found in the code file for CODE.
- (CODE1 CODE2 ...) means either CODE1 or CODE2 ... for this position in the code string. A way of dealing with ambiguity.
- (:PHRASE (PHRASE_CODE) (element1 element2 ...)) means a phrase with a phrase code, and it consists of elements. The list of elements is like an input string itself with one element for one position in the main string (although phrase codes may not be nested).

4.6 Filter Format

In a configuration between an active edge and a passive edge, a new task will be created only if the filter of the active edge matches the feature structure of the passive edge. Interpretation of filter expressions:

- A name is interpreted as a test that `<* name>` has a value in the edge being filtered.
- NOT logically negates some condition.
- AND works as a conjunction of conditions.
- OR works as a disjunction of conditions.
- Parentheses may be used to group conditions.

An example: `NOT foo AND (word.cat OR (fie AND fum))`.

4.7 The Start Rule

There is a reserved rule name `start.rule`. If it is declared in the grammar, an active edge will be added to the chart when it is initialised, with the start rule on it's stack. The edge will span from and to the first vertex, with an empty feature structure.

4.8 Logging

There are a number of options available to control what is written to the log file. The log interface is found in `src/logdump.h`.

- GRAMMAR - prints the grammar to the log.
- TRACE - prints a line numbered grammar trace to the log.
- FILTER - prints filter evaluations to the log, as they occur.
- CHART - prints information about edges when they are added to the chart.
- CHARTFS - prints the feature structure of edges as they are added to the chart.

- AGENDA - prints information about tasks when they are added to the agenda.
- ENDCHART - prints the chart as it looks after parsing has finished.
- STARTCHART - prints the chart as it looks after initiation has finished.
- CHARTTABLE - prints the chart as a table.
- ACT - include active edges in the log.
- PASS - include passive edges in the log.

4.9 Compiling UCP Light

The build system is based in GNU Make, with a file called `Makefile` in every relevant directory. The standard way of compiling UCP Light is to enter an interface directory, e.g. `interfaces/simple`, and typing `make clean all`. A few customisations are possible by editing `src/config.h`, and general compiler flags can be found in `config.make`.

5 Design and Implementation

This section is dedicated to selected matters in the implementation process that are relevant to the purpose of this work.

A straight conversion of the Interlisp (Teitelman, Hartley, Goodwin, Lewis, Bobrow and Masinter 1974) code of UCP2 into C would not have been efficient. The data structures should be carefully designed to be simple and efficient to implement in the chosen programming language. Converting the lists of Lisp directly into C data structures would not necessarily be time optimal nor space optimal. It was appropriate to base the C implementation on a specification describing functionality rather than algorithmic details.

5.1 Agenda, Chart and Tasks

The incremental task IDs of UCP2 are assigned to all kinds of tasks. As UCP Light does not have dictionary search tasks and rewriting tasks, it made sense to make incremental IDs, but not trying to imitate the ID numbers of UCP2 (which would have been difficult, and lead to confusing gaps in the incremental number sequence).

5.2 Feature Structures

The implementation of the feature structures and their functions turned out to be crucial for the performance of UCP Light (see section 3.3 for a description of feature structures). Benchmarking the first complete prototype of UCP Light showed that a high percentage of the processing time was spent on the feature structures (operations like copying etc). The prototype was careful to make copies of any edge's feature structure that was to be used by a task. This was seen as necessary as the contents of data submitted to the chart must not be changed. The need for always making these copies was removed by making it possible to write protect feature structures. All edges' feature structures are now write protected. If a task tries to alter the original feature structure, a copy is made dynamically which replaces the task's original data. This is implemented in the rule interpreter's unification function. Before starting the unification, it checks whether the feature structure is write protected, and if it is, a copy replaces the task's original FS. This decreased the overall workload for the parser, as copying only occurs when there is really a need for it.

It is convenient to think of feature structures as graphs, with nodes and labeled arcs. They can be cyclic as they may contain references to other nodes. All feature structures are implemented as blocks of integers in a single array. Each block represents one graph node (excluded extension blocks, which conceptually belong to the originating block). The data type FS is simply an integer holding the index to a node in the array. References, like mother-node to daughter-node, are actually indices to other nodes in the array. This makes the array relocatable and independent of the underlying memory addresses. See figure 5.2 for a BNF representation of the blocks. The capital strings represent reserved symbol numbers.

Below follows an example of a section of the array, representing a feature structure. As a notational convention, arguments will from here on be attached to their operator (`NIL(arg0)`, `REDIR(arg0)` etc).

Array index	Data	Mnemonic	Comment
1	1025 0	META(0)	metadata: e.g. writeprotected
3	1026 1034 11	FS('a', 11)	(a = <value at index 11>)
6	1026 1035 17	FS('b', 17)	(b = <value at index 17>)
9	1029 0	STOP(0)	argument: placeholder
11	1025 0	META(0)	
13	1027 1035	ATOM('b')	
15	1029 0	STOP(0)	
17	1024 0	NIL(0)	metadata: e.g. writeprotected

Figure 5.1: Example of how the array works. The FS at position 1 is (a = b, b = NIL). The mnemonics are simply the symbols associated with the numbers in the second column.

```

block      :- metatypes | redir | nil
metatypes  :- META int ( atom | fs )
redir      :- REDIR int
nil        :- NIL int
atom       :- ATOM int STOP int
fs         :- ( FS int int )+ ( STOP int | CONT int )

```

Reserved keywords:

META int The argument holds bit-packed information about the block: if it is write protected, if it is an extension of another block (happens when a feature structure gets more children by means of unification), and the length of the block in bytes.

ATOM int The argument holds the symbol that the atom represents.

FS int int The first argument holds the symbol and the second argument holds the position of the target block.

REDIR int The argument holds the position the block that is the target of the redirection.

NIL int The argument holds bit packed information whether the block is write protected.

STOP int The argument is a placeholder, needed if the the STOP is converted into a CONT (happens when a feature structure gets more children by means of unification).

CONT int The argument holds the position the block which is the extension to the current block.

Figure 5.2: A BNF representation of how feature structure node blocks are built (+ means one or more occurrence, a | b means a or b, and parentheses are used for grouping).

`NIL(meta)` implements nil values. The argument number holds information on whether this is write protected data. If a nil value is unified with another value, the `NIL(meta)` block is overwritten by a `REDIR(dest)` (in case it is not write protected), where `dest` is a reference to the other value's block.

`REDIR(dest)` means that this block is to be found at the index held by the argument. This is how shared values are implemented.

`META(data)` is the start of a block holding either an atom (a leaf) or a node with zero or more outgoing labeled arcs. `META`'s argument holds information on whether this block is write protected, if it is a tail of another block that has been extended, and the number of symbols constituting this block (knowing this speeds up the copying of blocks). The `STOP(UNUSED)` always indicates the end of a `META` block (the `UNUSED` argument is a place holder, see below for information about `CONT(dest)`). Only one `ATOM` block will be created for any symbol, making it universal. See figure 5.3 for some examples.

Feature Structure	Blocks (as array index: representation)
'a	1: META(data), ATOM('a'), STOP(UNUSED)
'nil	6: NIL(0)
(b = a)	8: META(data), FS('b', 1), STOP(UNUSED)
(a = (b = a), (a = a))	14: META(data), FS('a', 20), STOP(UNUSED) 20: META(data), FS('a', 1), FS('b', 1), STOP(UNUSED)

Figure 5.3: Some examples of feature structure representation. Note that index 0 is reserved, representing the absence of a feature structure.

By means of unification, non-terminal nodes can be extended with more daughter nodes. Simply inserting new data into the array, making space by shifting all data after the insertion point forward, is not acceptable. As we have seen, references are made to indices in the array, which means a block cannot be moved. To add another `FS(attribute, dest)` to the list of a block (taken the block is not write protected), we overwrite the `STOP(UNUSED)` element with a `CONT(dest)`, which means that the list continues in the block specified by the argument. This is why `STOP` has a place holder argument, holding place for future `CONT` extensions. That block's `META(data)` has information that it is a continued block from somewhere else. Apart from that, it is a normal attribute list block, like its originating block. If two compatible feature structures are unified, they should be identical, meaning that they share the same representation. This is why there is a `REDIR(dest)`, which overwrites the `META(data)` of a block. The redirection always happens before any other operation, even identity checks, which is why feature structures with static positions in the array can share a single block.

A design decision was made that the internal order of daughter nodes is unordered. This means no time is wasted on sorting when inserting. However, in applications where it is expected that nodes will have great numbers of daughters, it might be faster to keep the daughters sorted, as traversing a path will be faster.

When copying a feature structure, shared values must be intact. The algorithm is quite simple, using the block system. No atom blocks need copying. The blocks are traversed, and for every block that has not already been traversed, a copy is made, and a cross reference table is updated that maps the original block index with the copy's index. If the copy contains any `FS(attribute, dest)`, the index of the destination argument is saved in a list. When ready, the argument list is traversed, and the destination arguments of the copies are updated using the cross reference table. At the same time, blocks with `CONT(dest)` tails are merged into a single block.

UCP Light needs to know if a symbol represents a numeric value. This is because of the path operators (see section 4.2). Instead of having to mark up the symbol with additional data, all values in the symbol table below a threshold value are reserved for numbers. This means that a symbol with a number below the threshold represents the same number. This way, it is very easy to check if it is a numerical symbol.

5.2.1 Symbols are Identical

As described in the purpose of this work, it is of higher priority to create a correct behaviour than maintaining compatibility with UCP2, should a conflict arise. One such conflict is that symbols are not identical in UCP2, which is demonstrated in the following test:

```
<& a> == <& b>,      ;; <& a> and <& b> are undefined up until now.  
'a :=: <& a>,      ;; line 1  
'a :=: <& b>,      ;; line 2  
<& a> == <& b>,      ;; line 3
```

In UCP2, the identity check at line 3 fails. It is common to think of symbols as universal. It is also a waste of memory to have separate copies of every occurrence of one single symbol (which is needed to make the identity check above fail, as the identity check is defined as checking whether two references point at the same copy). UCP Light is implemented to think of symbols as identical, which means it will succeed with the test on line 3 above.

5.3 The Rule Interpreter

The grammar parser builds annotated syntax trees (Aho et al. 1986:33-36). Every node has a type, which defines its semantics. Every rule is executed in the context of a task, which provides the current feature structure, and is done by traversing the syntax tree. See figure 5.4 for an example.

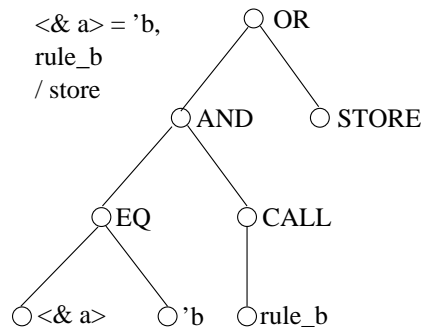


Figure 5.4: A code example with the corresponding syntax tree.

5.4 Parsing input data

It is common practise to construct compiler front ends and other complicated parsers using a *compiler compiler*, a compiler that produces a compiler (Levine, Mason and Brown 1990). They allow the programmer to describe the input language as an annotated context free grammar. That grammar is then compiled into a stand alone file of C code. The three input parsers, for the code file, the grammar and the input string, were implemented using Bison and Flex. The generated C code is very efficient, which in turn makes the initialisation of UCP Light efficient.

5.5 Macros as Templates

Many frequently called functions, and in some cases entire modules, have been implemented as macros. A C preprocessor macro is a name (with optional arguments) that is associated with a text string. The preprocessor replaces all occurrences of a macro name with the associated text string before it is sent to the compiler.

The stack and array data types have been fully implemented as macros, usable for all atomic data types. This is because ANSI C does not cater for templating (as does C++), which means that you would have to have a separate set of array structures and functions for each data type you would like to store in the array. See table 5.1.

Definition	<pre>#define ArSet(array, type, pos, elem) \ ((type *)array->mem)[pos] = elem;</pre>
Used with int	<pre>ArSet(freq, int, at, 10);</pre>
Used with pointer	<pre>ArSet(ptrs, char *, at, str);</pre>

Table 5.1: Example of a macro definition and its flexible uses.

5.6 Memory Management

The program keeps two memory pools. One that is kept from the initialisation of the parser until the parser exits by `ucpQuit()`. The other pool is reset between each parse. Instead of `malloc`, use `memNew`. See the header file. Usually you switch memory pool with `memSet(glob->statMem)` and `memSet(glob->sessMem)`.

5.6.1 Simplified Garbage Collection

By having a pool which is reset between the parses, the program does not care to try to free allocated memory in the pool during parsing. This is a simple model and it minimises the overhead that would follow from sorting the memory pool to collect garbage.

5.7 API - Application Programmer Interface

In order to use UCP Light in another program, it is possible to compile a library which can be used the standard way. The simplest way to understand the API of UCP Light is to study the existing interfaces. After unpacking the source code, they can be found under the directory in `interfaces/`.

For simple tasks, it is enough to use the following functions, located in the files `src/ucp.h` and `src/ucp.c`.

Function name: `int ucpInit(char *codeFile, char *ruleFile)`

Call time: Initialisation

To accomplish: Initiate the parser with the code file and the rule file.
There are example files provided with the source code.

Function name: `UcpResult * ucpParse(char *codeStr)`

Call time: Main

To accomplish: Sends a code string to parse, starts parsing.

Function name: `void ucpQuit()`

Call time: End

To accomplish: Return resident memory to system and shut down.

Function name: `void ucpVersion()`

Call time: Anytime

To accomplish: Returns version information.

A very simple example:

1. `ucpInit('`codes`', '`rules.grm`');`

Initiates the parser with the code and rule files.

2. `ucpParse('`((NN1 NN2))`');`

Parses a code string. When this is finished, the result is stored in the chart. If post processing is desired, this is when to do it. Otherwise there are various ways of printing the chart.

3. `void ucpQuit();`
Shuts down UCP Light and frees the memory.

5.8 Output of UCP Light

Calling `UcpResult * ucpParse(char *codeStr)` returns a pointer to a static struct that looks like this:

```
typedef struct _UcpResult {
    short errors;          /* No errors: 0, gram error: >0, tech. error: <0 */

#ifdef LOG_TO_FILE
    struct _Array *errLog; /* These are made of session memory. */
    struct _Array *warnLog;
    struct _Array *msgLog;
#endif

#ifdef SCARRIE_INTERFACE
    RESULT_SCARRIE_ERR
#endif
} UcpResult;
```

`UcpResult->errors` indicates the outcome of the parsing. A value less than 0 indicate a technical error, arising from a malformed code string, out of memory error etc. No errors means no technical errors occurred. If the ScarCheck interface is used, a value greater than or equal to 0 indicates how many error triplets were found, and are stored in `UcpResult->errItems`. The `errItems->start` indicates the start vertex, `errItems->stop` indicates the stop vertex and `errItems->error` indicates actual error as a static string.

6 Evaluation

The purpose of the work (see section 1.1) was to create a fast C-version of UCP, able to reproduce the syntactic processing of UCP2. To compare UCP2 with UCP Light, a grammar is needed which can be used by both parsers. As both UCP Light and UCP2 are adapted to ScarCheck, it is suitable to use the ScarCheck grammar for the evaluation. There is a minor difference in the way edge positions is represented in UCP Light and UCP2 which means the grammars are not completely equal at the rule level. UCP Light uses the !a, !b and !c constant operators (see section 4.1), whereas UCP2 stores edge position information in the feature structures. This makes the feature structures of UCP2 slightly bigger.

6.1 Performance

Measuring speed in a multitasking environment requires a timer that can measure the CPU time of a process, which is not the same as the actual elapsed time. If a task requires c CPU seconds to complete, and the time (wall clock) elapsed from starting the process until it stops is e , the percentage of CPU attention it gets is $(c / e * 100)$ ¹. From this follows that using elapsed times is not useful for time comparisons, instead CPU time should be used.

Figure 6.1 shows the speed comparison between UCP2 and UCP Light, based on the times in figures 6.1 and 6.1. According to the final test, UCP Light is faster by a factor of about 21, which means that the goal of creating a faster version has clearly been achieved. Although UCP2 has somewhat larger feature structures to copy, it does not weigh out the difference in speed.

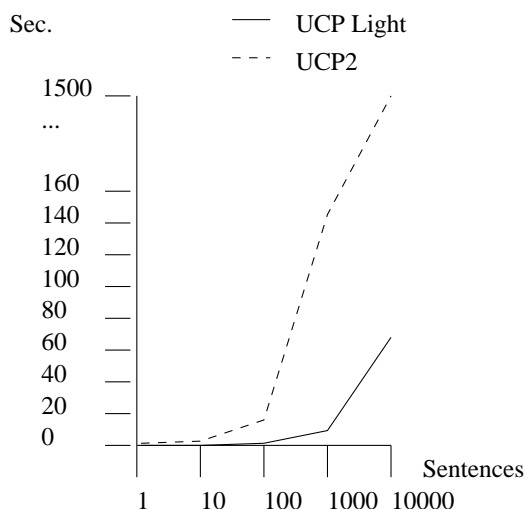


Figure 6.1: Speed comparison using the ScarCheck grammar and the newspaper text.

¹This can easily be tested by measuring the time of some task, with and without a heavy background task.

Sentences	User CPU sec	System CPU sec	Total CPU sec
1	0.08	0.07	0.15
10	0.15	0.08	0.23
100	0.80	0.06	0.86
1000	7.07	0.12	7.19
10000	71.78	0.63	72.41

Figure 6.2: Measured times using UCP Light with the ScarCheck grammar and the newspaper text.

Sentences	User CPU sec	System CPU sec	Total CPU sec
1	0.25	0.06	0.31
10	1.71	0.10	1.81
100	15.39	0.52	15.91
1000	150.65	4.17	154.82
10000	1495.03	43.95	1538.98

Figure 6.3: Measured times using UCP2 with the ScarCheck grammar and newspaper text.

The statistics was derived using GNU time v1.7. The grammar and code file were taken from ScarCheck. The grammar has 133 rules (2200 lines, 52KB), the code file has 369 codes (20KB). The ScarCheck grammar is written in the UCP2 syntax, but a conversion program can be used to produce an equivalent grammar in the UCP Light syntax. The conversion program was developed by Leif-Jöran Olsson. The code file is written in the UCP Light syntax, although it may also be automatically converted. The input sentences were extracted from a newspaper corpus and converted into code strings using ScarCheck (see section A.2). The corpus contains text from Uppsala Nya Tidning 1995-1996. UCP2 and UCP Light have different ways of providing this information. The feature structures of UCP2 are somewhat bigger because of this.

The test was run under RedHat GNU/Linux 7.1, on a machine with a 750MHz AMD Athlon and 256MB DRAM.

UCP2 uses less memory than UCP Light. In the final test in section 6.1, it stabilised at about 3.1MB, whereas UCP Light stabilised at 16MB. This is most likely not a problem on modern computers. Most memory is allocated at initialisation, with big defaults values for the data buffers. The default values have not been tuned, so they may well be decreased in the future.

6.2 Compatibility with UCP2

The method of evaluation is straightforward. An analysis is produced by analysing a text using UCP2 together with the ScarCheck grammar. The same text is analysed by UCP Light using the same grammar. The chart of UCP Light either matches or mismatches the chart of UCP2. This will not prove that UCP2 and UCP Light always work the same, but it is a reasonable test of quality. If it can process the test corpus successfully, it can replace UCP2 as a parser in ScarCheck. Acceptable and expected differences are due to the additional edge position feature structures of UCP2².

As specified in section 1.1, the format of the grammar rule files has changed:

However, the main importance is that equal data are stored in the charts, given equal input. There are various output styles available in UCP Light, including one that looks like the one of UCP2. The difference in internal

²In exceptional cases there will be differences stemming from grammar constructions depending on symbols not being identical. This follows from the fact that symbols are identical in UCP Light, and not identical in UCP2 (see section 5.2). This should not happen, as a proper grammar should not have such dependencies.

```

UCP2:
  (define sve.gram-entry np_qual
    #u      store(<& gd>);)

UCP Light:
  defrule np_qual {
    store(<& gd>)
  }

```

Figure 6.4: A small example displaying the differences in the grammar rule format.

representation of edge ID-numbers (described in section 5.1) can be observed but is irrelevant.

Section A.7 shows an authentic test to see whether UCP2 and UCP Light can produce an equivalent analysis in ScarCheck. It uses the input data described in 6.1. After the input data was converted, both parsers processed its data, and the charts were printed. A manual inspection shows that the test is successful.

6.3 ANSI C Conformance

To test whether the code complies to ANSI C, it was automatically checked by `gcc`³. Using the flags `-pedantic` and `-ansi`, `gcc` produces warnings for non conformities. The test shows that the code is conforming to the standard, as no such warnings were produced.

³GNU Compiler Collection

7 Conclusion

The purpose of this work was to implement a version of UCP that is faster than UCP2, and to do so in ANSI C. Given equivalent input they should produce equivalent output, taken the input is within the shared grammar scope. The evaluation indicates it has been a success. It was faster by a factor of 21 in the bigger tests, and the code complies to ANSI C. No proof has been presented that UCP Light and UCP2 always produce equivalent output given equivalent input. Instead, tests were made indicating that UCP Light is capable of producing the desired output. UCP Light has another way of handling symbols in feature structures, in that symbols are now universal and identical. This was a design decision, based on the assumption that it is more important to be correct than to maintain backward compatibility with UCP2.

UCP Light was initially implemented during seven months in 1998 and 1999. Since then, it has undergone different phases of debugging and improvements. It is presently used in various applications and prototypes, such as the Scania Checker (Sågvall Hein and Almqvist 2000), ScarCheck (Sågvall Hein and Starbäck 1999, ScarCheck 1999, SCARRIE 1996), the MATS project (MATS 2001), the KOMA project (KOMA 2002) and in Fasty (FASTY 2001).

Future work involves making a visualisation tool for the data structures to help the author of the grammar rules. The possibility to export a complete parsing session as XML would provide a nice way for a graphical interface to recreate the session visually. The XML export could also be used to compare one session with another. When the need arises, it is possible to reintroduce the removed morphological features of UCP.

Bibliography

- Aho, A. V., Sethi, R. and Ullman, J. D. (1986). *Compilers, Principles, Techniques, and Tools*, Addison. ISBN 0-201-10088-6.
- Allen, J. (1987). *Natural Language Understanding*, Benjamin Cummings.
- Caraballo, S. A. and Charniak, E. (1998). New figures of merit for best-first probabilistic chart parsing, *Computational Linguistics* **24**(2): 275–298.
- Carlsson, M. (1982). *UCP2 System Documentation*, Center for Computational Linguistics, Uppsala University.
- Carroll, J. (1993). Practical unification-based parsing of natural language. Computer Laboratory Technical Report 314.
- Earley, J. (1970). An efficient context-free parsing algorithm, *j-CACM* **13**(2): 94–102.
- FASTY (2001). The fasty project, World Wide Web. <http://www.fortec.tuwien.ac.at/reha.e/projects/fasty/fasty.html>.
- Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in LISP: an introduction to computational linguistics*, Addison-Wesley Publishing Company, chapter 6. Well-formed substring tables and charts.
- Hellwig, P. (1999). Natural Language Parsers - A "Course in Cooking".
- J Cocke and J T Schwartz (1970). *Programming Languages and Their Compilers*, Courant Institute.
- Kaplan, R. M. (1973). *Natural Language Processing*, Randall Rustin (editor), New York: Algorithmics Press, chapter A general syntactic processor.
- Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context free languages.
- Kay, M. (1967). Experiments with a powerful parser, *Technical Report RM-5452-PR*, The Rand Corporation, Santa Monica, Calif.
- Kay, M. (1989). Head driven parsing, *Proceedings of the 1st International Workshop on Parsing Technologies*, Stanford, CA: CSLI Publications, pp. 64–68.
- Knuth, D. E. (1965). On the translation of languages from left to right, *Information and Control*, **8**(6):607–639.
- KOMA (2002). The koma project, World Wide Web. <http://www.ida.liu.se/~nlplab/koma>.
- Levine, J. R., Mason, T. and Brown, D. (1990). *Lex & Yacc*, O'Reilly & Associates, Inc.
- Martin, W. A., Church, K. W. and Patil, R. S. (1981). Preliminary analysis of a breadth-first parsing algorithm: Theoretical and experimental results, *Technical Report MIT/LCS/TR-261*, Laboratory for Computer Science, Massachusetts Institute of Technology. Section 1.3.

- MATS (2001). The mats project, World Wide Web. <http://stp.ling.uu.se/mats>.
- ScarCheck (1999). The SCARRIE prototype, World Wide Web. <http://stp.ling.uu.se/~ljo/scarrie-pub/>.
- SCARRIE (1996). The SCARRIE project, World Wide Web. <http://www.scarrie.com/>.
- Sågvall Hein, A. (1980). An overview of the uppsala chart parser version 1 (ucp-1), *Technical Report UC DL-R-80-1*, Uppsala University, Center for Computational Linguistics.
- Sågvall Hein, A. (1983). A parser for Swedish, status report for Sve.Ucp, *Technical Report UC DL-R-83-2*, Uppsala University, Center for Computational Linguistics.
- Sågvall Hein, A. (1987). Parsing by means of Uppsala Chart Processor, in L. Bolc (ed.), *Natural Language Parsing Systems*, Berlin: Springer.
- Sågvall Hein, A. and Almqvist, I. (2000). A language checker of controlled language and its integration in a documentation and translation workflow, *Proceedings from Translating and the Computer 22*. Forthcoming.
- Sågvall Hein, A. and Starbäck, P. (1999). A test version of the grammar checker for swedish, deliverable 6.5.1, *Working Papers in Computational Linguistics Language Engineering*.
- Shieber, S. M., Uszkoreit, H., Pereira, F. C. N., Robinson, J. and Tyson, M. (1983). The formalism and implementation of patr-ii, in B. J. Grosz and M. E. Stickel (eds), *Research on Interactive Acquisition and Use of Knowledge*, SRI report.
- Teitelman, W., Hartley, A. K., Goodwin, J. W., Lewis, D. C., Bobrow, D. G. and Masinter, L. M. (1974). *Interlisp Reference Manual*, Palo Alto, Xerox Parc.
- Thompson, H. S. and Ritchie, G. D. (1984). Implementing natural language parsers, in T. O'Shea and M. Eisenstadt (eds), *Artificial Intelligence: Tools, Techniques, and Applications*, Harper & Row.
- Tomita, M. (1985). An efficient context-free parsing algorithm for natural languages, *Proc. of the 9th IJCAI*, Los Angeles, CA, pp. 756–764.
- Woods, W. A. (1970). Transition network grammars for natural language analysis, *J-CACM* **13**(10): 591–606.
- Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3 , *Information and Control* **10**(2): 189–208.

A Appendix

A.1 Executing a Task Stack

During execution, instructions are popped off the top of the stack and then evaluated. As rules may be recursive and may pass formal parameters (arguments), each function is executed in the context of a frame. The current frame of execution is the last one to be pushed onto the stack. The bottom element of every stack is always a frame, so there is always a starting frame context. When another rule is called, a new frame is created with local values of the formal parameters. When execution is finished in a function and it returns to the caller, the local frame is removed, and the caller's frame becomes the active one. See figure A.1. For a detailed list of the data constituting a task, see section A.3.

```
defrule start.rule {                               STACK:
  rule_a('call0),
  store
}                                                    3) unify(<& arg0>, arg0)
defrule rule_a {                                    2) FRAME rule_a [arg0 = 'call0]
  <& arg0> ::= ?0   <---BREAK POINT 1) store
}                                                         0) FRAME start.rule [no args]
```

Figure A.1: The stack as it looks when execution has reached the indicated break point, starting from `start.rule`

A.2 Creating code strings for evaluation

Here follows a brief recapture of how the code strings were created. It is meant just as a reference to anyone who wants to recreate the tests.

```
#!/bin/bash
GRAM=/local/scarrie/rules.grm
CODE=/local/scarrie/corrie-codes
TEXT=/tmp/u1

head -25000 /corpora/scarrie/milan/unt9596.tag > $TEXT

LOG=/tmp/eval.log
DEF=/tmp/eval.def
echo "grammar=$GRAM $CODE" > $DEF

cd /home/staff/ljo/scarrie/corrie/corucp-linux
./corrietr webb_file.def +webbp.def +$DEF $TEXT -l $LOG

cat $LOG | sed 's/<[^>]*>//g' | sed 's/[^()]*$//'' | grep '^('
```

A.3 Key Data Structures

The data constituting a task:

<code>unsigned id;</code>	A unique task ID.
<code>unsigned creatorId;</code>	The ID of the creator task of this task.
<code>unsigned start;</code>	Start node of active edge.
<code>unsigned coll;</code>	End node of active edge, start node of passive edge.
<code>unsigned stop;</code>	End node of passive edge.
<code>struct _Filter *filter;</code>	Points to the action's filter.
<code>struct _PNode *frame;</code>	Points to the current frame on the stack.
<code>Stack *stack;</code>	Execution stack.
<code>FS fs;</code>	Feature structure.

The data constituting a vertex:

<code>unsigned id;</code>	A unique ID.
<code>struct { struct _Array *passive; struct _Array *active; } incoming;</code>	Incoming passive edges. Incoming active edges.
<code>struct { struct _Array *passive; struct _Array *active; } outgoing;</code>	Outgoing passive edges. Outgoing active edges.

The data constituting an edge:

<code>unsigned nibr;</code>	A unique ID.
<code>unsigned creatorId;</code>	ID of creator task.
<code>unsigned start;</code>	Index to start in vertex Array.
<code>unsigned stop;</code>	Index to stop in vertex Array.
<code>FS fs;</code>	Feature structure.
<code>BOOL active;</code>	True if this is an active edge.
<code>struct _Filter *filter;</code>	Current filter.
<code>Stack *action;</code>	Execution stack.

The data constituting the chart:

<code>struct _Array *vertices;</code>	Array of vertices, ordered.
<code>struct _Array *edges;</code>	Pointers to all edges in the chart.

A.4 Example Grammar

An example code file:

```
##### Article
:AL gen
AN neutrum !toy_np
AU utrum !toy_np

##### Noun
:N gen
NN neutrum !
NU utrum !
```

An example grammar:

```
defrule toy_np {
  <* word.cat> = 'AL,
  <& part :new> := <*>,
  <& phr.cat> := 'np,
  advance,
  <* word.cat> = 'N,
  <& part :last word.cat> = 'AL,
  <& part :last gen> = <* gen>,
  <& part :new> := <*>,
  <& part phr.cat> := 'np,
  store(<& part>)
}

defrule start.rule {
  failure
}
```

A.5 Errors

Invalid Characters A character not recognised by the lexical scanners will jam the parser (either the rule parser, code parser or the code string parser) and generate an error message. If this happens, it will most likely be in the initialisation phase when UCP Light parses the rules and codes. One way of dealing with such unusual characters is to edit the lexical scanners (`rulescan.l`, `codescan.l`, `sentscan.l`) and recompile. The rule file format 4 includes a quote operator which can be used to prefix unrecognised characters, which means no recompilation is necessary.

Undeclared Codes If the code string 4.5 contains a code that has not been declared in the code table 4.4, UCP Light will generate an error message and abort the parsing.

Call to an Undeclared Rule If a rule or code declaration contains a call to a rule which is not declared, UCP Light will generate an error message and abort the initialisation 2.2.1. There is one unfortunate exception to this however, due to the ambiguity in passing arguments to rule calls. It is legal to send the name of a rule as an argument, e.g. `PROC(NPDET)`. It is also legal to send a `PATH` atom as argument, e.g. `ASSIGN(CASE)`. The rule parser will not be able to tell the difference from these, as it does not know how the argument is used in the called rule. Hence it is possible to send the name of a rule as an argument, and in the called function use it to call the rule of this name. If there is no such rule, execution will be aborted.

Out of Memory A request to the operating system for a chunk of memory failed. Try shutting down a few applications or increasing your system's swap memory.

A.6 ScarCheck-specific Extensions

As UCP Light was developed to be used by ScarCheck, these extensions should be presented as well.

UCP Light will, after parsing has finished, analyse the chart. It will look for passive edges (longest span reported first) which contains error triplets. An error triplet is a certain construction found as value to `<* err>`, if such exists. In case many error triplets are to be added, they are the values of `<* err 1>`, `<* err 2>` etc. An error triplet looks like:

```
(code = error code
 start = start vertex numer
 stop = stop vertex number)
```

The error code is a symbol which should be declared by `deferror` in the grammar:

```
deferror error code ``error message``
```

This is the operator for pairing an error code with an error message. For a discussion on error codes, see section A.6. For example:

```
deferror gpvnmv01 "finite verb missing"
```

A.7 Compatibility Test

The set of edges between any two vertices should be the same. The feature structures of UCP2 may have additional `start`, `end` and `first` attributes, which replace the constant operators of UCP Light. Their creator numbers may also differ (see section 5.1). Note that they have different ways of printing the actions of active edges.

UCP2	UCP Light
1--1 Creator: 0 LR-Action: START.RULE;	1--1 Nbr: 0 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] start.rule [0 start.rule:0 EXEC_FRAME]
1--1 Creator: 2 LR-Action: ADVP;	1--1 Nbr: 1 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] advp [0 advp:0 EXEC_FRAME]
1--1 Creator: 2 LR-Action: ADJP_ADJ;	1--1 Nbr: 2 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] adjp_adj [0 adjp_adj:0 EXEC_FRAME]
1--1 Creator: 4 LR-Action: ADJP_ADVP;	1--1 Nbr: 149 Creator: 1 LR-Action: [1 start.rule:0 E_SUBCALL] adjp_advp [0 adjp_advp:0 EXEC_FRAME]
1--1 Creator: 4 LR-Action: CL.DECL_XP;	1--1 Nbr: 150 Creator: 1 LR-Action: [1 start.rule:0 E_SUBCALL] cl.decl_xp [0 cl.decl_xp:0 EXEC_FRAME]
1--1 Creator: 40 LR-Action: NP_ADJP;	1--1 Nbr: 147 Creator: 335 LR-Action: [1 start.rule:0 E_SUBCALL] np_adj [0 np_adj:0 EXEC_FRAME]
1--2 Creator: 2 Features: (* = (A-FORM = T DEGREE = POS CASE = BASIC FORM = INDEF NUMB = SING GENDER = NEUTR WORD.CAT = ADJ END = 2 START = 1 LEM = SAMTIDIG.AV FIRST = +))	1--2 Nbr: 3 Creator: 0 Features: (* = (word.cat = 'ADJ, gender = 'neutr, numb = 'sing, form = 'indef, case = 'basic, degree = 'pos, a-form = 't, lem = 'SAMTIDIG.AV))
1--2 Creator: 6 Features: (* = (START = 1 END = 2 PHR.CAT = ADJP ADJ1 = (WORD.CAT = ADJ LEM = SAMTIDIG.AV DEGREE = POS) GENDER = NEUTR NUMB = SING CASE = BASIC PROP = NIL A-FORM = T FUNC = NIL FORM = INDEF))	1--2 Nbr: 145 Creator: 2 Features: (* = (phr.cat = 'adjp, adj1 = (word.cat = 'ADJ, lem = 'SAMTIDIG.AV, degree = 'pos), gender = 'neutr, numb = 'sing, case = 'basic, prop = 'nil, a-form = 't, func = 'nil, form = 'indef))
1--2 Creator: 4 Features: (* = (START = 1 END = 2 PHR.CAT = ADVP FIRST = (WORD.CAT = ADJ LEM = SAMTIDIG.AV) SADV = NIL))	1--2 Nbr: 151 Creator: 1 Features: (* = (phr.cat = 'advp, first = (word.cat = 'ADJ, lem = 'SAMTIDIG.AV), sadv = 'nil))
1--2 Creator: 6 Features: (& = (PHR.CAT = ADJP ADJ1 = (WORD.CAT = ADJ LEM = SAMTIDIG.AV DEGREE = POS) GENDER = NEUTR NUMB = SING CASE = BASIC PROP = NIL A-FORM = T FUNC = NIL FORM = INDEF)) LR-Action: CONTINUE, ASSIGN.MAJORPROCESS(NP_ADJP), (<* WORD.CAT> = 'CONJ, NOT <* SUBJU>, ASSIGN.MAJORPROCESS(ADJP.COORD_ADJP)/ <* WORD.CAT> = 'SEP, (<* LEM> = 'COMMA.SR, ASSIGN.MAJORPROCESS (ADJP.COORD_ADJP)/ <* LEM> = 'HYPCONT.SR, ASSIGN.MAJORPROCESS (NP.HYPCONT))/ <* PHR.CAT> = 'ADJP, NOT <* 1>, NOT <* ADJ1 LEM> = 'EGEN.AV, NOT <* ADJ1 LEM> = 'VISS.AV, (IF <& ADJ1 WORD.CAT> = 'PART, <& ADJ1 PART.TYPE> = 'PAST THEN NOT <& A-FORM> = 'A, NOT <* A-FORM> = 'A ELSE CONTINUE), (IF <* ADJ1 WORD.CAT> = 'PART, <* ADJ1 PART.TYPE> = 'PAST THEN NOT <* A-FORM> = 'A ELSE CONTINUE), <& ADJ2 PHR.CAT> := 'ADJP, (ASSIGN (GENDER)/ NOT <* SUP>, ASSIGN.ERR('GPAPAG01)), (ASSIGN (A-FORM)/ NOERROR, NOT <& A-FORM> = 'T, ASSIGN.ERR('GPAPAG01)), STORE);	1--2 Nbr: 146 Creator: 2 Features: (& = (phr.cat = 'adjp, adj1 = (word.cat = 'ADJ, lem = 'SAMTIDIG.AV, degree = 'pos), gender = 'neutr, numb = 'sing, case = 'basic, prop = 'nil, a-form = 't, func = 'nil, form = 'indef)) LR-Action: [1 adjp_adj:173 EXEC_FRAME] [0 adjp_adj:0 EXEC_FRAME]

<p>1--2 Creator: 48 Features: (& = (PHR.CAT = NP NUMB = SING GENDER = NEUTR A-FORM = T ATTR = (WORD.CAT = ADJ LEM = SAMTIDIG.AV DEGREE = POS FORM = INDEF PROP = NIL) INIT = +)) LR-Action: NP.ADJ_NOUN;</p>	<p>1--2 Nmbr: 148 Creator: 337 Features: (& = (phr.cat = 'np, numb = 'sing, gender = 'neutr, a-form = 't, attr = (word.cat = 'ADJ, lem = 'SAMTIDIG.AV, degree = 'pos, form = 'indef, prop = 'nil'), init = '+)) LR-Action: [2 np_adj:426 E_SUBCALL] np.adj_noun [1 np_adj:426 EXEC_FRAME] [0 np_adj:0 EXEC_FRAME]</p>
<p>1--2 Creator: 4 Features: (& = (PHR.CAT = ADVP FIRST = (WORD.CAT = ADJ LEM = SAMTIDIG.AV) SADV = NIL)) LR-Action: CONTINUE, NOT <* LEM> = 'HELLER.AB, NOT <* LEM> = 'VAD.AB, NOT <* QUEST> = '+, (<* WORD.CAT> = 'ADV/ <* WORD.CAT> = 'ADJ, <* A-FORM> = 'T, NOT <* ADV> = '-/ <* WORD.CAT> = 'DT), <& SEC> := '+, STORE, ADVANCE, NOT <* LEM> = 'HELLER.AB, (<* WORD.CAT> = 'ADV/ <* WORD.CAT> = 'ADJ, <* A-FORM> = 'T, NOT <* ADV> = '-), <& THIRD> := '+, STORE;</p>	<p>1--2 Nmbr: 152 Creator: 1 Features: (& = (phr.cat = 'advp, first = (word.cat = 'ADJ, lem = 'SAMTIDIG.AV), sadv = 'nil)) LR-Action: [1 advp:86 EXEC_FRAME] [0 advp:0 EXEC_FRAME]</p>
<p>1--2 Creator: 36 Features: (& = (FUND = (PHR.CAT = ADVP) PHR.CAT = CL.DECL.FRAG INV = +)) LR-Action: VFIN.INV, STORE;</p>	<p>1--2 Nmbr: 153 Creator: 350 Features: (& = (fund = (phr.cat = 'advp), phr.cat = 'cl.decl.frag, inv = '+)) LR-Action: [3 cl.decl_xp:1380 E_SUBCALL] vfin.inv [1 cl.decl_xp:1380 EXEC_FRAME] [0 cl.decl_xp:0 EXEC_FRAME]</p>
<p>1--2 Creator: 34 Features: (& = (FUND = (PHR.CAT = ADVP) PHR.CAT = CL.DECL.FRAG INV = +)) LR-Action: (CONTINUE, <* PHR.CAT> = 'NP, <* HEAD WORD.CAT> = 'PRON, NOT <* CASE> = 'ACC, NOT <* CASE> = 'GEN, NOT <* HEAD LEM> = 'EN.PN, NOT <* HEAD LEM> = 'DET.PN, NOT <* ERR>, ASSIGN.ERR ('GPWOIN02), <& SUBJ> := <*>, STORE;</p>	<p>1--2 Nmbr: 154 Creator: 349 Features: (& = (fund = (phr.cat = 'advp), phr.cat = 'cl.decl.frag, inv = '+)) LR-Action: [1 cl.decl_xp:1382 EXEC_FRAME] [0 cl.decl_xp:0 EXEC_FRAME]</p>
<p>1--2 Creator: 14 Features: (& = (PHR.CAT = ADJP MOD = SAMTIDIG.AV)) LR-Action: CONTINUE, <* PHR.CAT> = 'ADJP, (IF <& MOD> = 'SÅ.AB THEN NOT <* ADJ1 LEM> = 'RÅKNA.PC ELSE CONTINUE), NOT <* SUP>, <& ADJ1 LEM> := <* ADJ1 LEM>, <& ADJ1 WORD.CAT> := <* ADJ1 WORD.CAT>, ASSIGN(A-FORM), ASSIGN(GENDER), ASSIGN(NUMB), STORE, MAJORPROCESS(NP_ADJP);</p>	<p>1--2 Nmbr: 155 Creator: 346 Features: (& = (phr.cat = 'adjp, mod = 'SAMTIDIG.AV)) LR-Action: [1 adjp_adv:203 EXEC_FRAME] [0 adjp_adv:0 EXEC_FRAME]</p>
<p>2--2 Creator: 32 LR-Action: VP_VP;</p>	<p>2--2 Nmbr: 4 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] vp_vp [0 vp_vp:0 EXEC_FRAME]</p>
<p>2--4 Creator: 32 Features: (* = (PHRASE = + INFF = FIN WORD.CAT = VP END = 4 START = 2))</p>	<p>2--4 Nmbr: 5 Creator: 0 Features: (* = (word.cat = 'VP, inff = 'fin, phrase = '+))</p>
<p>4--4 Creator: 44 LR-Action: NP_NOUN;</p>	<p>4--4 Nmbr: 6 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_noun [0 np_noun:0 EXEC_FRAME]</p>
<p>4--4 Creator: 44 LR-Action: HYPCONT;</p>	<p>4--4 Nmbr: 7 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] hypcont [0 hypcont:0 EXEC_FRAME]</p>
<p>4--5 Creator: 44 Features: (* = (CASE = BASIC FORM = INDEF NUMB = SING GENDER = UTR WORD.CAT = NOUN END = 5 START = 4 LEM = VERNISSAGE.NN))</p>	<p>4--5 Nmbr: 8 Creator: 0 Features: (* = (word.cat = 'NOUN, gender = 'utr, numb = 'sing, form = 'indef, case = 'basic, lem = 'VERNISSAGE.NN))</p>

<pre>4--5 Creator: 64 Features: (* = (START = 4 END = 5 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) NUMB = SING))</pre>	<pre>4--5 Nmr: 137 Creator: 322 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), numb = #3))</pre>
<pre>4--5 Creator: 54 Features: (& = (1 = VERNISSAGE.NN)) LR-Action: CONTINUE, <* WORD.CAT> = 'SEP, <* LEM> = 'HYPCONT.SR, ADVANCE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU>, <& :NEW> := <* LEM>, ADVANCE, (<* WORD.CAT> = 'NOUN, <& WORD.CAT> := 'NOUN, <& HYPCONT> := '+, ASSIGN(CASE), ASSIGN(NUMB), ASSIGN (FORM), ASSIGN(GENDER), <& :NEW> := <* LEM>, (<* FORM> = 'DEF, ASSIGN.MAJORPROCESS(NP_NOUN)/ CONTINUE)/ <* WORD.CAT> = 'ADJ, <& :NEW> := <* LEM>, <& PHR.CAT> := 'ADJP, ASSIGN(A-FORM), ASSIGN(NUMB), ASSIGN(GENDER), ASSIGN(FUNC), ASSIGN(FORM)), STORE;</pre>	<pre>4--5 Nmr: 136 Creator: 5 Features: (& = (1 = 'VERNISSAGE.NN)) LR-Action: [1 hypcont:348 EXEC_FRAME] [0 hypcont:0 EXEC_FRAME]</pre>
<pre>4--5 Creator: 62 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN))) LR-Action: NOUN.TAIL;</pre>	<pre>4--5 Nmr: 138 Creator: 321 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] noun.tail [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</pre>
<pre>4--5 Creator: 60 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN))) LR-Action: NP.COORD.CONJ;</pre>	<pre>4--5 Nmr: 144 Creator: 320 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] np.coord.conj [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</pre>
<pre>4--7 Creator: 250 Features: (* = (START = 4 END = 7 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</pre>	<pre>4--7 Nmr: 139 Creator: 331 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), post.attr = (phr.cat = 'pp), numb = #3))</pre>

<p>4--9 Creator: 446 Features: (* = (START = 4 END = 9 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>4--9 Nmbr: 140 Creator: 330 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>4--10 Creator: 490 Features: (* = (START = 4 END = 10 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>4--10 Nmbr: 141 Creator: 329 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>4--13 Creator: 628 Features: (* = (START = 4 END = 13 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>4--13 Nmbr: 142 Creator: 328 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>4--16 Creator: 704 Features: (* = (START = 4 END = 16 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = VERNISSAGE.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>4--16 Nmbr: 143 Creator: 326 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'VERNISSAGE.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>5--5 Creator: 56 LR-Action: PP_PREP;</p>	<p>5--5 Nmbr: 9 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] pp_prep [0 pp_prep:0 EXEC_FRAME]</p>
<p>5--6 Creator: 56 Features: (* = (PART = + WORD.CAT = PREP END = 6 START = 5 LEM = I.PP))</p>	<p>5--6 Nmbr: 10 Creator: 0 Features: (* = (word.cat = 'PREP, part = '+, lem = 'I.PP))</p>
<p>5--6 Creator: 78 Features: (& = (PREP = (LEM = I.PP))) LR-Action: PP.PREP_NP;</p>	<p>5--6 Nmbr: 119 Creator: 270 Features: (& = (prep = (lem = 'I.PP))) LR-Action: [2 pp_prep:1172 E_SUBCALL] pp.prep_np [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>5--7 Creator: 194 Features: (* = (START = 5 END = 7 PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = HERRGÅRD.NN GENDER = UTR) PHR.CAT = PP))</p>	<p>5--7 Nmbr: 134 Creator: 309 Features: (* = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'HERRGÅRD.NN, gender = 'utr), phr.cat = 'pp))</p>

<p>5--13 Creator: 582 Features: (* = (START = 5 END = 13 PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP))</p>	<p>5--13 Nmbr: 126 Creator: 292 Features: (* = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil'), phr.cat = 'pp))</p>
<p>5--13 Creator: 664 Features: (* = (START = 5 END = 13 PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = HERRGÅRD.NN GENDER = UTR) PHR.CAT = PP))</p>	<p>5--13 Nmbr: 128 Creator: 296 Features: (* = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'HERRGÅRD.NN, gender = 'utr'), phr.cat = 'pp))</p>
<p>5--13 Creator: 582 Features: (& = (PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>5--13 Nmbr: 127 Creator: 292 Features: (& = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil'), phr.cat = 'pp)) LR-Action: [2 pp.prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>5--13 Creator: 664 Features: (& = (PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = HERRGÅRD.NN GENDER = UTR) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>5--13 Nmbr: 129 Creator: 296 Features: (& = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'HERRGÅRD.NN, gender = 'utr'), phr.cat = 'pp)) LR-Action: [2 pp.prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>5--16 Creator: 702 Features: (* = (START = 5 END = 16 PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP))</p>	<p>5--16 Nmbr: 124 Creator: 289 Features: (* = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil'), phr.cat = 'pp))</p>
<p>5--16 Creator: 702 Features: (& = (PREP = (LEM = I.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>5--16 Nmbr: 125 Creator: 289 Features: (& = (prep = (lem = 'I.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil'), phr.cat = 'pp)) LR-Action: [2 pp.prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>6--6 Creator: 70 LR-Action: NP_NOUN;</p>	<p>6--6 Nmbr: 11 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_noun [0 np_noun:0 EXEC_FRAME]</p>
<p>6--6 Creator: 128 LR-Action: NP_NP.COORD.NP;</p>	<p>6--6 Nmbr: 103 Creator: 225 LR-Action: [1 start.rule:0 E_SUBCALL] np_np.coord.np [0 np_np.coord.np:0 EXEC_FRAME]</p>
<p>6--7 Creator: 70 Features: (* = (CASE = BASIC FORM = DEF NUMB = SING GENDER = UTR WORD.CAT = NOUN END = 7 START = 6 LEM = HERRGÅRD.NN))</p>	<p>6--7 Nmbr: 12 Creator: 0 Features: (* = (word.cat = 'NOUN, gender = 'utr, numb = 'sing, form = 'def, case = 'basic, lem = 'HERRGÅRD.NN))</p>
<p>6--7 Creator: 88 Features: (* = (START = 6 END = 7 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN) NUMB = SING))</p>	<p>6--7 Nmbr: 97 Creator: 211 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN), numb = #3))</p>

<p>6--7 Creator: 86 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN))) LR-Action: NOUN.TAIL;</p>	<p>6--7 Nmbr: 98 Creator: 210 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] noun.tail [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</p>
<p>6--7 Creator: 84 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN))) LR-Action: NP.COORD.CONJ;</p>	<p>6--7 Nmbr: 102 Creator: 209 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] np.coord.conj [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</p>
<p>6--7 Creator: 188 Features: (& = (COORD = + PHR.CAT = NP GNPAG03 = NIL)) LR-Action: CONTINUE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU> = '+, ADVANCE, <* PHR.CAT> = 'NP, PROPAGATE.ERR, STORE, ADVANCE(DET.REL.TAIL);</p>	<p>6--7 Nmbr: 107 Creator: 229 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [1 np_np.coord.np:990 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]</p>
<p>6--8 Creator: 246 Features: (& = (COORD = + PHR.CAT = NP GNPAG03 = NIL)) LR-Action: CONTINUE, <* PHR.CAT> = 'NP, PROPAGATE.ERR, STORE, ADVANCE(DET.REL.TAIL);</p>	<p>6--8 Nmbr: 108 Creator: 241 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [1 np_np.coord.np:993 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]</p>
<p>6--9 Creator: 324 Features: (* = (START = 6 END = 9 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>6--9 Nmbr: 99 Creator: 222 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>6--9 Creator: 302 Features: (* = (START = 6 END = 9 COORD = + PHR.CAT = NP GNPAG03 = NIL))</p>	<p>6--9 Nmbr: 117 Creator: 254 Features: (* = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil))</p>
<p>6--9 Creator: 370 Features: (& = (COORD = + PHR.CAT = NP GNPAG03 = NIL)) LR-Action: CONTINUE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU> = '+, ADVANCE, <* PHR.CAT> = 'NP, PROPAGATE.ERR, STORE, ADVANCE(DET.REL.TAIL);</p>	<p>6--9 Nmbr: 106 Creator: 230 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [1 np_np.coord.np:990 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]</p>
<p>6--9 Creator: 302 Features: (& = (COORD = + PHR.CAT = NP GNPAG03 = NIL)) LR-Action: DET.REL.TAIL;</p>	<p>6--9 Nmbr: 118 Creator: 254 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [2 np_np.coord.np:997 E_SUBCALL] det.rel.tail [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]</p>

6--10 Creator: 466 Features: (* = (START = 6 END = 10 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))	6--10 Nmbr: 100 Creator: 221 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN), post.attr = (phr.cat = 'pp), numb = #3))
6--10 Creator: 352 Features: (* = (START = 6 END = 10 COORD = + PHR.CAT = NP GPNPAG03 = NIL))	6--10 Nmbr: 113 Creator: 255 Features: (* = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil))
6--10 Creator: 494 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: CONTINUE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU> = '+, ADVANCE, <* PHR.CAT> = 'NP, PROPAGATE.ERR, STORE, ADVANCE (DET.REL.TAIL);	6--10 Nmbr: 105 Creator: 231 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [1 np_np.coord.np:990 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
6--10 Creator: 352 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: DET.REL.TAIL;	6--10 Nmbr: 114 Creator: 255 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [2 np_np.coord.np:997 E_SUBCALL] det.rel.tail [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
6--13 Creator: 642 Features: (* = (START = 6 END = 13 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = DEF HEAD = (ADJ = NIL FORM = DEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = HERRGÅRD.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))	6--13 Nmbr: 101 Creator: 220 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'def, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'HERRGÅRD.NN), post.attr = (phr.cat = 'pp), numb = #3))
6--13 Creator: 504 Features: (* = (START = 6 END = 13 COORD = + PHR.CAT = NP GPNPAG03 = NIL))	6--13 Nmbr: 109 Creator: 256 Features: (* = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil))
6--13 Creator: 504 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: (DET.REL.TAIL/ <* PHR.CAT> = 'NP, <* DEF> = 'DEF, <* CASE> = 'BASIC, PROCESS(CL_REL_NO.PRON)/ <* PHR.CAT> = 'ADVP, <* FIRST LEM> = 'DÅR.AB, PROCESS(CL_REL));	6--13 Nmbr: 104 Creator: 232 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [1 np_np.coord.np:990 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
6--13 Creator: 534 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: DET.REL.TAIL;	6--13 Nmbr: 110 Creator: 256 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [2 np_np.coord.np:997 E_SUBCALL] det.rel.tail [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]

6--13 Creator: 652 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: CONTINUE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU> = '+, ADVANCE, <* PHR.CAT> = 'NP, PROPAGATE.ERR, STORE, ADVANCE(DET.REL.TAIL);	6--13 Nmbr: 115 Creator: 264 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [4 det.rel.tail:2135 E_SUBCALL] det.rel.tail [2 det.rel.tail:2135 EXEC_FRAME] [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
6--16 Creator: 674 Features: (* = (START = 6 END = 16 COORD = + PHR.CAT = NP GPNPAG03 = NIL))	6--16 Nmbr: 111 Creator: 261 Features: (* = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil))
6--16 Creator: 674 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: (DET.REL.TAIL/ <* PHR.CAT> = 'NP, <* DEF> = 'DEF, <* CASE> = 'BASIC, PROCESS(CL_REL_NO.PRON)/ <* PHR.CAT> = 'ADVP, <* FIRST LEM> = 'DÄR.AB, PROCESS(CL_REL));	6--16 Nmbr: 112 Creator: 261 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [4 det.rel.tail:2135 E_SUBCALL] det.rel.tail [2 det.rel.tail:2135 EXEC_FRAME] [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
6--16 Creator: 676 Features: (& = (COORD = + PHR.CAT = NP GPNPAG03 = NIL)) LR-Action: ((DET.REL.TAIL/ <* PHR.CAT> = 'NP, <* DEF> = 'DEF, <* CASE> = 'BASIC, PROCESS(CL_REL_NO.PRON)/ <* PHR.CAT> = 'ADVP, <* FIRST LEM> = 'DÄR.AB, PROCESS(CL_REL))/ <* PHR.CAT> = 'NP, <* DEF> = 'DEF, <* CASE> = 'BASIC, PROCESS(CL_REL_NO.PRON)/ <* PHR.CAT> = 'ADVP, <* FIRST LEM> = 'DÄR.AB, PROCESS(CL_REL));	6--16 Nmbr: 116 Creator: 265 Features: (& = (coord = '+, phr.cat = 'np, gpnpag03 = 'nil)) LR-Action: [6 det.rel.tail:2135 E_SUBCALL] det.rel.tail [4 det.rel.tail:2135 EXEC_FRAME] [2 det.rel.tail:2135 EXEC_FRAME] [1 np_np.coord.np:997 EXEC_FRAME] [0 np_np.coord.np:0 EXEC_FRAME]
7--7 Creator: 82 LR-Action: ADVP;	7--7 Nmbr: 13 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] advp [0 advp:0 EXEC_FRAME]
7--7 Creator: 82 LR-Action: CL_CONJ;	7--7 Nmbr: 15 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] cl_conj [0 cl_conj:0 EXEC_FRAME]
7--7 Creator: 82 LR-Action: PP_PREP;	7--7 Nmbr: 17 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] pp_prep [0 pp_prep:0 EXEC_FRAME]
7--7 Creator: 82 LR-Action: VPF;	7--7 Nmbr: 19 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] vpf [0 vpf:0 EXEC_FRAME]
7--7 Creator: 96 LR-Action: NP_ADV;	7--7 Nmbr: 92 Creator: 8 LR-Action: [1 start.rule:0 E_SUBCALL] np_adv [0 np_adv:0 EXEC_FRAME]
7--7 Creator: 96 LR-Action: ADJP_ADV;	7--7 Nmbr: 93 Creator: 8 LR-Action: [1 start.rule:0 E_SUBCALL] adjp_adv [0 adjp_adv:0 EXEC_FRAME]
7--8 Creator: 82 Features: (* = (GRADE = + WORD.CAT = ADV END = 8 START = 7 LEM = FÖR.AB))	7--8 Nmbr: 14 Creator: 0 Features: (* = (word.cat = 'ADV, grade = '+, lem = 'FÖR.AB))
7--8 Creator: 82 Features: (* = (WORD.CAT = CONJ END = 8 START = 7 LEM = FÖR.CN))	7--8 Nmbr: 16 Creator: 0 Features: (* = (word.cat = 'CONJ, lem = 'FÖR.CN))
7--8 Creator: 82 Features: (* = (PART = + WORD.CAT = PREP END = 8 START = 7 LEM = FÖR.PP))	7--8 Nmbr: 18 Creator: 0 Features: (* = (word.cat = 'PREP, part = '+, lem = 'FÖR.PP))
7--8 Creator: 82 Features: (* = (IMP = + TENSE = PRES VERB.TYPE = MAIN INFF = FIN DIAT = ACT WORD.CAT = VERB END = 8 START = 7 LEM = FÖRA.VB))	7--8 Nmbr: 20 Creator: 0 Features: (* = (word.cat = 'VERB, diat = 'act, inff = 'fin, verb.type = 'main, tense = 'pres, imp = '+, lem = 'FÖRA.VB))

<p>7--8 Creator: 114 Features: (* = (START = 7 END = 8 INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN)))</p>	<p>7--8 Nmbr: 67 Creator: 23 Features: (* = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verb1 = (lem = #1, verb.type = 'main)))</p>
<p>7--8 Creator: 96 Features: (* = (START = 7 END = 8 PHR.CAT = ADVP FIRST = (WORD.CAT = ADV LEM = FÖR.AB) SADV = NIL))</p>	<p>7--8 Nmbr: 94 Creator: 8 Features: (* = (phr.cat = 'advp, first = (word.cat = 'ADV, lem = 'FÖR.AB), sadv = 'nil))</p>
<p>7--8 Creator: 168 Features: (& = (INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN))) LR-Action: (INF_COMPL);</p>	<p>7--8 Nmbr: 68 Creator: 101 Features: (& = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verb1 = (lem = #1, verb.type = 'main))) LR-Action: [2 vp:f:1756 E_SUBCALL] inf_compl [1 vp:f:1756 EXEC_FRAME] [0 vp:f:0 EXEC_FRAME]</p>
<p>7--8 Creator: 166 Features: (& = (INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN))) LR-Action: (VPF.VBFIN_VERB);</p>	<p>7--8 Nmbr: 72 Creator: 100 Features: (& = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verb1 = (lem = #1, verb.type = 'main))) LR-Action: [2 vp:f:1756 E_SUBCALL] vpf.vbfin_verb [1 vp:f:1756 EXEC_FRAME] [0 vp:f:0 EXEC_FRAME]</p>
<p>7--8 Creator: 158 Features: (& = (PREP = (LEM = FÖR.PP))) LR-Action: PP.PREP_NP;</p>	<p>7--8 Nmbr: 73 Creator: 113 Features: (& = (prep = (lem = 'FÖR.PP))) LR-Action: [2 pp_prep:1172 E_SUBCALL] pp.prep_np [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>7--8 Creator: 156 Features: (& = (PREP = (LEM = FÖR.PP))) LR-Action: (CONTINUE, (* WORD.CAT>, (* LEM> = 'ATT.IE, MAJORPROCESS(CL.INF_CONJ)/ (* PHR.CAT>, FÖR.SEDAN));</p>	<p>7--8 Nmbr: 80 Creator: 112 Features: (& = (prep = (lem = 'FÖR.PP))) LR-Action: [1 pp_prep:1174 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>7--8 Creator: 102 Features: (& = (CONJ = +)) LR-Action: CONTINUE, PROCESS(CL.DECL_XP), PROCESS(CL.DECL_NP);</p>	<p>7--8 Nmbr: 83 Creator: 11 Features: (& = (conj = '+)) LR-Action: [1 cl_conj:1464 EXEC_FRAME] [0 cl_conj:0 EXEC_FRAME]</p>
<p>7--8 Creator: 96 Features: (& = (PHR.CAT = ADVP FIRST = (WORD.CAT = ADV LEM = FÖR.AB) SADV = NIL)) LR-Action: CONTINUE, NOT (* LEM> = 'HELLER.AB, NOT (* LEM> = 'VAD.AB, NOT (* QUEST> = '+', (* WORD.CAT> = 'ADV/ (* WORD.CAT> = 'ADJ, (* A-FORM> = 'T, NOT (* ADV> = '-/ (* WORD.CAT> = 'DT), <& SEC> := '+, STORE, ADVANCE, NOT (* LEM> = 'HELLER.AB, (* WORD.CAT> = 'ADV/ (* WORD.CAT> = 'ADJ, (* A-FORM> = 'T, NOT (* ADV> = '-), <& THIRD> := '+, STORE;</p>	<p>7--8 Nmbr: 95 Creator: 8 Features: (& = (phr.cat = 'advp, first = (word.cat = 'ADV, lem = 'FÖR.AB), sadv = 'nil)) LR-Action: [1 advp:86 EXEC_FRAME] [0 advp:0 EXEC_FRAME]</p>
<p>7--8 Creator: 150 Features: (& = (PHR.CAT = ADJP MOD = FÖR.AB)) LR-Action: CONTINUE, (* PHR.CAT> = 'ADJP, (IF <& MOD> = 'SÄ.AB THEN NOT (* ADJ1 LEM> = 'RÄKNA.PC ELSE CONTINUE), NOT (* SUP>, <& ADJ1 LEM> := (* ADJ1 LEM>, <& ADJ1 WORD.CAT> := (* ADJ1 WORD.CAT>, ASSIGN(A-FORM), ASSIGN(GENDER), ASSIGN(NUMB), STORE, MAJORPROCESS(NP_ADJP);</p>	<p>7--8 Nmbr: 96 Creator: 200 Features: (& = (phr.cat = 'adjp, mod = 'FÖR.AB)) LR-Action: [1 adjp_advdp:203 EXEC_FRAME] [0 adjp_advdp:0 EXEC_FRAME]</p>

<p>7--9 Creator: 218 Features: (* = (START = 7 END = 9 INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN) NP = (START = 8 END = 9 PHR.CAT = NP HEAD = (WORD.CAT = PRON LEM = EN.PN) NUMB = SING CASE = BASIC GENDER = UTR)))</p>	<p>7--9 Nnbr: 71 Creator: 104 Features: (* = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verbl = (lem = #1, verb.type = 'main), np = (phr.cat = 'np, head = (word.cat = 'PRON, lem = 'EN.PN), numb = 'sing, case = 'basic, gender = 'utr)))</p>
<p>7--9 Creator: 286 Features: (* = (START = 7 END = 9 PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = EN.PN GENDER = UTR) PHR.CAT = PP))</p>	<p>7--9 Nnbr: 78 Creator: 126 Features: (* = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'EN.PN, gender = 'utr), phr.cat = 'pp))</p>
<p>7--9 Creator: 286 Features: (& = (PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = EN.PN GENDER = UTR) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>7--9 Nnbr: 79 Creator: 126 Features: (& = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'EN.PN, gender = 'utr), phr.cat = 'pp)) LR-Action: [2 pp.prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>7--10 Creator: 358 Features: (* = (START = 7 END = 10 INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN) NP = (START = 8 END = 10 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC)))</p>	<p>7--10 Nnbr: 70 Creator: 105 Features: (* = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verbl = (lem = #1, verb.type = 'main), np = (phr.cat = 'np, def = #2:'indef, form = #2, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #3:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #3, form = #2), case = 'basic)))</p>
<p>7--10 Creator: 420 Features: (* = (START = 7 END = 10 PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = UTSTÄLLNING.NN GENDER = UTR) PHR.CAT = PP))</p>	<p>7--10 Nnbr: 76 Creator: 122 Features: (* = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'UTSTÄLLNING.NN, gender = 'utr), phr.cat = 'pp))</p>
<p>7--10 Creator: 420 Features: (& = (PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = UTSTÄLLNING.NN GENDER = UTR) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>7--10 Nnbr: 77 Creator: 122 Features: (& = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'UTSTÄLLNING.NN, gender = 'utr), phr.cat = 'pp)) LR-Action: [2 pp.prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>

<p>7--10 Creator: 362 Features: (& = (PREP = (LEM = FÖR.PP))) LR-Action: (((FÖR.NP_SEDAN/ <* PHR.CAT> = 'VP.FRAG, NOT <* VP VERB WORD.CAT> = 'PART, NOT <* SUP> = '+, (<* INFF> :=: 'INF/ <* INF> = '+), NOT <* VP HEAD LEM> = 'SINA.VB, NOT <* VP HEAD LEM> = 'HELA.VB, NOT <* VP HEAD LEM> = 'SÅ.VB, NOT <* VP HEAD LEM> = 'ÄKTA.VB, NOT <* VP HEAD LEM> = 'TVÅ.VB, NOT <* VP HEAD LEM> = 'BÅDA.VB, NOT <* VP HEAD LEM> = 'KORTA.VB, NOT <* VP HEAD LEM> = 'LÅGA.VB, ASSIGN.ERR('GPVVIP02), <& PHR.CAT> :=: 'PP, PROPAGATE.ERR, STORE));</p>	<p>7--10 Nmbr: 82 Creator: 133 Features: (& = (prep = (lem = 'FÖR.PP))) LR-Action: [5 för.sedan:1185 E_SUBCALL] för.np_sedan [3 för.sedan:1185 EXEC_FRAME] [1 pp_prep:1176 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>7--13 Creator: 540 Features: (* = (START = 7 END = 13 INFF = FIN FIN = NIL INF = NIL DIAT = ACT PHR.CAT = VP.FRAG MAIN = (LEM = FÖRA.VB) VERB1 = (LEM = FÖRA.VB VERB.TYPE = MAIN) NP = (START = 8 END = 13 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC POST.ATTR = (PHR.CAT = PP))))</p>	<p>7--13 Nmbr: 69 Creator: 106 Features: (* = (inff = 'fin, fin = 'nil, inf = 'nil, diat = 'act, phr.cat = 'vp.frag, main = (lem = #1:'FÖRA.VB), verb1 = (lem = #1, verb.type = 'main), np = (phr.cat = 'np, def = #2:'indef, form = #2, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #3:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #3, form = #2), case = 'basic, post.attr = (phr.cat = 'pp))))</p>
<p>7--13 Creator: 612 Features: (* = (START = 7 END = 13 PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = UTSTÄLLNING.NN GENDER = UTR) PHR.CAT = PP))</p>	<p>7--13 Nmbr: 74 Creator: 118 Features: (* = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'UTSTÄLLNING.NN, gender = 'utr), phr.cat = 'pp))</p>
<p>7--13 Creator: 612 Features: (& = (PREP = (LEM = FÖR.PP) POBJ = (PHR.CAT = NP LEM = UTSTÄLLNING.NN GENDER = UTR) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>7--13 Nmbr: 75 Creator: 118 Features: (& = (prep = (lem = 'FÖR.PP), pobj = (phr.cat = 'np, lem = 'UTSTÄLLNING.NN, gender = 'utr), phr.cat = 'pp)) LR-Action: [2 pp_prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>7--13 Creator: 544 Features: (& = (PREP = (LEM = FÖR.PP))) LR-Action: (((FÖR.NP_SEDAN/ <* PHR.CAT> = 'VP.FRAG, NOT <* VP VERB WORD.CAT> = 'PART, NOT <* SUP> = '+, (<* INFF> :=: 'INF/ <* INF> = '+), NOT <* VP HEAD LEM> = 'SINA.VB, NOT <* VP HEAD LEM> = 'HELA.VB, NOT <* VP HEAD LEM> = 'SÅ.VB, NOT <* VP HEAD LEM> = 'ÄKTA.VB, NOT <* VP HEAD LEM> = 'TVÅ.VB, NOT <* VP HEAD LEM> = 'BÅDA.VB, NOT <* VP HEAD LEM> = 'KORTA.VB, NOT <* VP HEAD LEM> = 'LÅGA.VB, ASSIGN.ERR('GPVVIP02), <& PHR.CAT> :=: 'PP, PROPAGATE.ERR, STORE));</p>	<p>7--13 Nmbr: 81 Creator: 134 Features: (& = (prep = (lem = 'FÖR.PP))) LR-Action: [5 för.sedan:1185 E_SUBCALL] för.np_sedan [3 för.sedan:1185 EXEC_FRAME] [1 pp_prep:1176 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>8--8 Creator: 122 LR-Action: NP_QUANT;</p>	<p>8--8 Nmbr: 21 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_quant [0 np_quant:0 EXEC_FRAME]</p>
<p>8--8 Creator: 122 LR-Action: NP_PRON;</p>	<p>8--8 Nmbr: 23 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_pron [0 np_pron:0 EXEC_FRAME]</p>
<p>8--8 Creator: 174 LR-Action: CL.DECL_XP;</p>	<p>8--8 Nmbr: 84 Creator: 138 LR-Action: [1 start.rule:0 E_SUBCALL] cl.decl_xp [0 cl.decl_xp:0 EXEC_FRAME]</p>

8--8 Creator: 174 LR-Action: CL.DECL_NP;	8--8 Nmbr: 85 Creator: 138 LR-Action: [1 start.rule:0 E_SUBCALL] cl.decl_np [0 cl.decl_np:0 EXEC_FRAME]
8--9 Creator: 122 Features: (* = (DET.TYPE = QUANT DEF = INDEF FORM = INDEF NUMB = SING GENDER = UTR WORD.CAT = ART END = 9 START = 8 LEM = EN.AL))	8--9 Nmbr: 22 Creator: 0 Features: (* = (word.cat = 'ART, gender = 'utr, numb = 'sing, form = #1:'indef, def = #1, det.type = 'quant, lem = 'EN.AL))
8--9 Creator: 122 Features: (* = (PRON.TYPE = PERS CASE = BASIC NUMB = SING GENDER = UTR WORD.CAT = PRON END = 9 START = 8 LEM = EN.PN))	8--9 Nmbr: 24 Creator: 0 Features: (* = (word.cat = 'PRON, gender = 'utr, numb = 'sing, case = 'basic, pron.type = 'pers, lem = 'EN.PN))
8--9 Creator: 178 Features: (* = (START = 8 END = 9 PHR.CAT = NP HEAD = (WORD.CAT = PRON LEM = EN.PN) NUMB = SING CASE = BASIC GENDER = UTR))	8--9 Nmbr: 60 Creator: 27 Features: (* = (phr.cat = 'np, head = (word.cat = 'PRON, lem = 'EN.PN), numb = 'sing, case = 'basic, gender = 'utr))
8--9 Creator: 214 Features: (* = (START = 8 END = 9 PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 9 PHR.CAT = NP HEAD = (WORD.CAT = PRON LEM = EN.PN) NUMB = SING CASE = BASIC GENDER = UTR))	8--9 Nmbr: 90 Creator: 146 Features: (* = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, head = (word.cat = 'PRON, lem = 'EN.PN), numb = 'sing, case = 'basic, gender = 'utr)))
8--9 Creator: 228 Features: (& = (PHR.CAT = NP HEAD = (WORD.CAT = PRON LEM = EN.PN) NUMB = SING CASE = BASIC GENDER = UTR)) LR-Action: NP.PRON_PP;	8--9 Nmbr: 61 Creator: 76 Features: (& = (phr.cat = 'np, head = (word.cat = 'PRON, lem = 'EN.PN), numb = 'sing, case = 'basic, gender = 'utr)) LR-Action: [2 np_pron:1014 E_SUBCALL] np.pron_pp [1 np_pron:1014 EXEC_FRAME] [0 np_pron:0 EXEC_FRAME]
8--9 Creator: 212 Features: (& = (PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR)) LR-Action: NP.DET_ADJP.OR.NOUN;	8--9 Nmbr: 62 Creator: 82 Features: (& = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = 'utr)) LR-Action: [2 np_quant:1120 E_SUBCALL] np.det_adjp.or.noun [1 np_quant:1120 EXEC_FRAME] [0 np_quant:0 EXEC_FRAME]
8--9 Creator: 210 Features: (& = (PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR)) LR-Action: NP_SEL;	8--9 Nmbr: 66 Creator: 81 Features: (& = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = 'utr)) LR-Action: [2 np_quant:1121 E_SUBCALL] np_sel [1 np_quant:1121 EXEC_FRAME] [0 np_quant:0 EXEC_FRAME]
8--9 Creator: 214 Features: (& = (PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 9 PHR.CAT = NP HEAD = (WORD.CAT = PRON LEM = EN.PN) NUMB = SING CASE = BASIC GENDER = UTR)) LR-Action: CONTINUE, CL.NP_VFIN;	8--9 Nmbr: 91 Creator: 146 Features: (& = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, head = (word.cat = 'PRON, lem = 'EN.PN), numb = 'sing, case = 'basic, gender = 'utr))) LR-Action: [1 cl.decl_np:1316 EXEC_FRAME] [0 cl.decl_np:0 EXEC_FRAME]

<pre> 8--10 Creator: 310 Features: (* = (START = 8 END = 10 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC)) </pre>	<pre> 8--10 Nmbr: 63 Creator: 93 Features: (* = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic)) </pre>
<pre> 8--10 Creator: 354 Features: (* = (START = 8 END = 10 PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 10 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC)) </pre>	<pre> 8--10 Nmbr: 88 Creator: 147 Features: (* = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic')) </pre>
<pre> 8--10 Creator: 310 Features: (& = (PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC)) LR-Action: ((CONTINUE, (<* PHR.CAT> = 'PP, <& POST.ATTR PHR.CAT> := 'PP/ <* PHR.CAT> = 'PAP, <& CASE> := 'BASIC, <& POST.ATTR PHR.CAT> := 'PAP/ <* PHR.CAT> = 'CL.SUB.FRAG, <* SUBJU> = 'OM.SN, <& POST.ATTR PHR.CAT> := 'CL.SUB.FRAG), PROPAGATE.ERR, STORE)); </pre>	<pre> 8--10 Nmbr: 64 Creator: 93 Features: (& = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic)) LR-Action: [3 det_noun:672 EXEC_FRAME] [2 np.det_adjp.or.noun:582 EXEC_FRAME] [1 np_quant:1120 EXEC_FRAME] [0 np_quant:0 EXEC_FRAME] </pre>
<pre> 8--10 Creator: 354 Features: (& = (PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 10 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC)) LR-Action: CONTINUE, CL.NP_VFIN; </pre>	<pre> 8--10 Nmbr: 89 Creator: 147 Features: (& = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic)) LR-Action: [1 cl.decl_np:1316 EXEC_FRAME] [0 cl.decl_np:0 EXEC_FRAME] </pre>
<pre> 8--13 Creator: 506 Features: (* = (START = 8 END = 13 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC POST.ATTR = (PHR.CAT = PP)) </pre>	<pre> 8--13 Nmbr: 65 Creator: 97 Features: (* = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic, post.attr = (phr.cat = 'pp)) </pre>

<p>8--13 Creator: 536 Features: (* = (START = 8 END = 13 PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 13 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC POST.ATTR = (PHR.CAT = PP))))</p>	<p>8--13 Nnbr: 86 Creator: 148 Features: (* = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic, post.attr = (phr.cat = 'pp))))</p>
<p>8--13 Creator: 536 Features: (& = (PHR.CAT = CL.DECL.FRAG SUBJ = (START = 8 END = 13 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = SING QUANT = (WORD.CAT = ART LEM = EN.AL) GENDER = UTR HEAD = (LEM = UTSTÄLLNING.NN GENDER = UTR FORM = INDEF) CASE = BASIC POST.ATTR = (PHR.CAT = PP)))) LR-Action: CONTINUE, CL.NP_VFIN;</p>	<p>8--13 Nnbr: 87 Creator: 148 Features: (& = (phr.cat = 'cl.decl.frag, subj = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'sing, quant = (word.cat = 'ART, lem = 'EN.AL), gender = #2:'utr, head = (lem = 'UTSTÄLLNING.NN, gender = #2, form = #1), case = 'basic, post.attr = (phr.cat = 'pp)))) LR-Action: [1 cl.decl_np:1316 EXEC_FRAME] [0 cl.decl_np:0 EXEC_FRAME]</p>
<p>9--9 Creator: 186 LR-Action: NP_NOUN;</p>	<p>9--9 Nnbr: 25 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_noun [0 np_noun:0 EXEC_FRAME]</p>
<p>9--9 Creator: 186 LR-Action: HYPCONT;</p>	<p>9--9 Nnbr: 26 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] hypcont [0 hypcont:0 EXEC_FRAME]</p>
<p>9--10 Creator: 186 Features: (* = (CASE = BASIC FORM = INDEF NUMB = SING GENDER = UTR WORD.CAT = NOUN END = 10 START = 9 LEM = UTSTÄLLNING.NN))</p>	<p>9--10 Nnbr: 27 Creator: 0 Features: (* = (word.cat = 'NOUN, gender = 'utr, numb = 'sing, form = 'indef, case = 'basic, lem = 'UTSTÄLLNING.NN))</p>
<p>9--10 Creator: 292 Features: (* = (START = 9 END = 10 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = UTSTÄLLNING.NN) NUMB = SING))</p>	<p>9--10 Nnbr: 56 Creator: 71 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'UTSTÄLLNING.NN), numb = #3))</p>
<p>9--10 Creator: 234 Features: (& = (1 = UTSTÄLLNING.NN)) LR-Action: CONTINUE, <* WORD.CAT> = 'SEP, <* LEM> = 'HYPCONT.SR, ADVANCE, <* WORD.CAT> = 'CONJ, NOT <* SUBJU>, <& :NEW> :=: <* LEM>, ADVANCE, (<* WORD.CAT> = 'NOUN, <& WORD.CAT> :=: 'NOUN, <& HYPCONT> :=: '+, ASSIGN(CASE), ASSIGN(NUMB), ASSIGN (FORM), ASSIGN(GENDER), <& :NEW> :=: <* LEM>, (<* FORM> = 'DEF, ASSIGN.MAJORPROCESS(NP_NOUN)/ CONTINUE)/ <* WORD.CAT> = 'ADJ, <& :NEW> :=: <* LEM>, <& PHR.CAT> :=: 'ADJP, ASSIGN(A-FORM), ASSIGN(NUMB), ASSIGN(GENDER), ASSIGN(FUNC), ASSIGN(FORM)), STORE;</p>	<p>9--10 Nnbr: 55 Creator: 29 Features: (& = (1 = 'UTSTÄLLNING.NN)) LR-Action: [1 hypcont:348 EXEC_FRAME] [0 hypcont:0 EXEC_FRAME]</p>

<p>9--10 Creator: 290 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = UTSTÄLLNING.NN))) LR-Action: NOUN.TAIL;</p>	<p>9--10 Nbr: 57 Creator: 70 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'UTSTÄLLNING.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] noun.tail [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</p>
<p>9--10 Creator: 288 Features: (& = (PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = UTSTÄLLNING.NN))) LR-Action: NP.COORD.CONJ;</p>	<p>9--10 Nbr: 59 Creator: 69 Features: (& = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'UTSTÄLLNING.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] np.coord.conj [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</p>
<p>9--13 Creator: 508 Features: (* = (START = 9 END = 13 PHR.CAT = NP GENDER = UTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = SING WORD.CAT = NOUN GENDER = UTR TITLE = NIL LEM = UTSTÄLLNING.NN) POST.ATTR = (PHR.CAT = PP) NUMB = SING))</p>	<p>9--13 Nbr: 58 Creator: 73 Features: (* = (phr.cat = 'np, gender = #1:'utr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'sing, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'UTSTÄLLNING.NN), post.attr = (phr.cat = 'pp), numb = #3))</p>
<p>10--10 Creator: 236 LR-Action: PP.PREP;</p>	<p>10--10 Nbr: 28 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] pp_prep [0 pp_prep:0 EXEC_FRAME]</p>
<p>10--11 Creator: 236 Features: (* = (PART = + WORD.CAT = PREP END = 11 START = 10 LEM = MED.PP))</p>	<p>10--11 Nbr: 29 Creator: 0 Features: (* = (word.cat = 'PREP, part = '+, lem = 'MED.PP))</p>
<p>10--11 Creator: 342 Features: (& = (PREP = (LEM = MED.PP))) LR-Action: PP.PREP_NP;</p>	<p>10--11 Nbr: 52 Creator: 62 Features: (& = (prep = (lem = 'MED.PP))) LR-Action: [2 pp_prep:1172 E_SUBCALL] pp.prep_np [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>10--13 Creator: 478 Features: (* = (START = 10 END = 13 PREP = (LEM = MED.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP))</p>	<p>10--13 Nbr: 53 Creator: 65 Features: (* = (prep = (lem = 'MED.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil), phr.cat = 'pp))</p>
<p>10--13 Creator: 478 Features: (& = (PREP = (LEM = MED.PP) POBJ = (PHR.CAT = NP LEM = NIL GENDER = NIL) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>10--13 Nbr: 54 Creator: 65 Features: (& = (prep = (lem = 'MED.PP), pobj = (phr.cat = 'np, lem = 'nil, gender = 'nil), phr.cat = 'pp)) LR-Action: [2 pp_prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>11--11 Creator: 344 LR-Action: NP_QUANT;</p>	<p>11--11 Nbr: 30 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_quant [0 np_quant:0 EXEC_FRAME]</p>

<p>11--13 Creator: 344 Features: (* = (LEM = EN_DEL.QP DET.TYPE = QUANT NUMB = NIL WORD.CAT = QP END = 13 START = 11))</p>	<p>11--13 Nmbr: 31 Creator: 0 Features: (* = (word.cat = 'QP, numb = 'nil, det.type = 'quant, lem = 'EN_DEL.qp))</p>
<p>11--13 Creator: 380 Features: (* = (START = 11 END = 13 PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = PLUR QUANT = (WORD.CAT = QP LEM = EN_DEL.QP) GENDER = NIL))</p>	<p>11--13 Nmbr: 49 Creator: 31 Features: (* = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'plur, quant = (word.cat = 'QP, lem = 'EN_DEL.qp), gender = 'nil))</p>
<p>11--13 Creator: 440 Features: (& = (PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = PLUR QUANT = (WORD.CAT = QP LEM = EN_DEL.QP) GENDER = NIL)) LR-Action: NP.DET_ADJP.OR.NOUN;</p>	<p>11--13 Nmbr: 50 Creator: 52 Features: (& = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'plur, quant = (word.cat = 'QP, lem = 'EN_DEL.qp), gender = 'nil)) LR-Action: [2 np_quant:1120 E_SUBCALL] np.det_adjp.or.noun [1 np_quant:1120 EXEC_FRAME] [0 np_quant:0 EXEC_FRAME]</p>
<p>11--13 Creator: 438 Features: (& = (PHR.CAT = NP DEF = INDEF FORM = INDEF NUMB = PLUR QUANT = (WORD.CAT = QP LEM = EN_DEL.QP) GENDER = NIL)) LR-Action: NP_SEL;</p>	<p>11--13 Nmbr: 51 Creator: 51 Features: (& = (phr.cat = 'np, def = #1:'indef, form = #1, numb = 'plur, quant = (word.cat = 'QP, lem = 'EN_DEL.qp), gender = 'nil)) LR-Action: [2 np_quant:1121 E_SUBCALL] np_sel [1 np_quant:1121 EXEC_FRAME] [0 np_quant:0 EXEC_FRAME]</p>
<p>13--13 Creator: 382 LR-Action: PP_PREP;</p>	<p>13--13 Nmbr: 32 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] pp_prep [0 pp_prep:0 EXEC_FRAME]</p>
<p>13--14 Creator: 382 Features: (* = (PART = + WORD.CAT = PREP END = 14 START = 13 LEM = AV.PP))</p>	<p>13--14 Nmbr: 33 Creator: 0 Features: (* = (word.cat = 'PREP, part = '+, lem = 'AV.PP))</p>
<p>13--14 Creator: 484 Features: (& = (PREP = (LEM = AV.PP))) LR-Action: PP.PREP_NP;</p>	<p>13--14 Nmbr: 46 Creator: 44 Features: (& = (prep = (lem = 'AV.PP))) LR-Action: [2 pp_prep:1172 E_SUBCALL] pp_prep_np [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>13--16 Creator: 658 Features: (* = (START = 13 END = 16 PREP = (LEM = AV.PP) POBJ = (PHR.CAT = NP LEM = ARBETE.NN GENDER = NEUTR) PHR.CAT = PP))</p>	<p>13--16 Nmbr: 47 Creator: 47 Features: (* = (prep = (lem = 'AV.PP), pobj = (phr.cat = 'np, lem = 'ARBETE.NN, gender = 'neutr), phr.cat = 'pp))</p>
<p>13--16 Creator: 658 Features: (& = (PREP = (LEM = AV.PP) POBJ = (PHR.CAT = NP LEM = ARBETE.NN GENDER = NEUTR) PHR.CAT = PP)) LR-Action: (CONTINUE, <* PHR.CAT> = 'ADJP, <* ADJ1 WORD.CAT> = 'PART, MAJORPROCESS(ADJP_PP));</p>	<p>13--16 Nmbr: 48 Creator: 47 Features: (& = (prep = (lem = 'AV.PP), pobj = (phr.cat = 'np, lem = 'ARBETE.NN, gender = 'neutr), phr.cat = 'pp)) LR-Action: [2 pp_prep_np:1220 EXEC_FRAME] [1 pp_prep:1172 EXEC_FRAME] [0 pp_prep:0 EXEC_FRAME]</p>
<p>14--14 Creator: 444 LR-Action: NP_POSS;</p>	<p>14--14 Nmbr: 34 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_poss [0 np_poss:0 EXEC_FRAME]</p>

14--15 Creator: 444 Features: (* = (NUMB = NIL GENDER = NIL WORD.CAT = PS END = 15 START = 14 LEM = DERAS.PN))	14--15 Nbr: 35 Creator: 0 Features: (* = (word.cat = 'PS, gender = 'nil, numb = 'nil, lem = 'DERAS.PN))
14--15 Creator: 486 Features: (& = (NUMB = NIL GENDER = NIL POSS = (WORD.CAT = PS LEM = DERAS.PN) PHR.CAT = NP)) LR-Action: NP.POSS_NP;	14--15 Nbr: 43 Creator: 33 Features: (& = (numb = 'nil, gender = 'nil, poss = (word.cat = 'PS, lem = 'DERAS.PN), phr.cat = 'np)) LR-Action: [2 np_poss:952 E_SUBCALL] np.poss_np [1 np_poss:952 EXEC_FRAME] [0 np_poss:0 EXEC_FRAME]
14--16 Creator: 626 Features: (* = (START = 14 END = 16 NUMB = PLUR GENDER = NEUTR POSS = (WORD.CAT = PS LEM = DERAS.PN) PHR.CAT = NP HEAD = (LEM = ARBETE.NN)))	14--16 Nbr: 44 Creator: 40 Features: (* = (numb = 'plur, gender = 'neutr, poss = (word.cat = 'PS, lem = 'DERAS.PN), phr.cat = 'np, head = (lem = 'ARBETE.NN)))
14--16 Creator: 626 Features: (& = (NUMB = PLUR GENDER = NEUTR POSS = (WORD.CAT = PS LEM = DERAS.PN) PHR.CAT = NP HEAD = (LEM = ARBETE.NN))) LR-Action: (NP.COORD.CONJ);	14--16 Nbr: 45 Creator: 40 Features: (& = (numb = 'plur, gender = 'neutr, poss = (word.cat = 'PS, lem = 'DERAS.PN), phr.cat = 'np, head = (lem = 'ARBETE.NN))) LR-Action: [3 np_poss_np:972 E_SUBCALL] np.coord.conj [2 np_poss_np:972 EXEC_FRAME] [1 np_poss:952 EXEC_FRAME] [0 np_poss:0 EXEC_FRAME]
15--15 Creator: 488 LR-Action: NP_NOUN;	15--15 Nbr: 36 Creator: 0 LR-Action: [1 start.rule:0 E_SUBCALL] np_noun [0 np_noun:0 EXEC_FRAME]
15--16 Creator: 488 Features: (* = (CASE = BASIC FORM = INDEF NUMB = PLUR GENDER = NEUTR WORD.CAT = NOUN END = 16 START = 15 LEM = ARBETE.NN))	15--16 Nbr: 37 Creator: 0 Features: (* = (word.cat = 'NOUN, gender = 'neutr, numb = 'plur, form = 'indef, case = 'basic, lem = 'ARBETE.NN))
15--16 Creator: 570 Features: (* = (START = 15 END = 16 PHR.CAT = NP GENDER = NEUTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = PLUR WORD.CAT = NOUN GENDER = NEUTR TITLE = NIL LEM = ARBETE.NN) NUMB = PLUR))	15--16 Nbr: 40 Creator: 37 Features: (* = (phr.cat = 'np, gender = #1:'neutr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = #3:'plur, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'ARBETE.NN), numb = #3))
15--16 Creator: 568 Features: (& = (PHR.CAT = NP GENDER = NEUTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = PLUR WORD.CAT = NOUN GENDER = NEUTR TITLE = NIL LEM = ARBETE.NN))) LR-Action: NOUN.TAIL;	15--16 Nbr: 41 Creator: 36 Features: (& = (phr.cat = 'np, gender = #1:'neutr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'plur, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'ARBETE.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] noun.tail [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]

<p>15--16 Creator: 566 Features: (& = (PHR.CAT = NP GENDER = NEUTR CASE = BASIC DEF = INDEF HEAD = (ADJ = NIL FORM = INDEF NUMB = PLUR WORD.CAT = NOUN GENDER = NEUTR TITLE = NIL LEM = ARBETE.NN))) LR-Action: NP.COORD.CONJ;</p>	<p>15--16 Nnbr: 42 Creator: 35 Features: (& = (phr.cat = 'np, gender = #1:'neutr, case = 'basic, def = #2:'indef, head = (adj = 'nil, form = #2, numb = 'plur, word.cat = 'NOUN, gender = #1, title = 'nil, lem = 'ARBETE.NN))) LR-Action: [2 np_noun:821 E_SUBCALL] np.coord.conj [1 np_noun:821 EXEC_FRAME] [0 np_noun:0 EXEC_FRAME]</p>
<p>16--17 Creator: 520 Features: (* = (LEM = STOP.SR TYPE = MAJOR WORD.CAT = SEP END = 17 START = 16))</p>	<p>16--17 Nnbr: 38 Creator: 0 Features: (* = (word.cat = 'SEP, type = 'major, lem = 'STOP.SR))</p>
<p>17--18 Creator: 572 Features: (* = (LEM = EOS.SR TYPE = MAJOR WORD.CAT = SEP END = 18 START = 17))</p>	<p>17--18 Nnbr: 39 Creator: 0 Features: (* = (word.cat = 'SEP, type = 'major, lem = 'EOS.SR))</p>