LANGUAGE-BASED ENVIRONMENT FOR NATURAL LANGUAGE PARSING

Lehtola, A., Jäppinen, H., Nelimarkka, E.
Sitra Foundation (*) and
Helsinki University of Technology
Helsinki, Finland

ABSTRACT

This paper introduces a special programming environment for the definition of grammars and for the implementation of corresponding parsers. In natural language processing systems it is advantageous to have linguistic knowledge and processing mechanisms separated. Our environment accepts grammars consisting of binary dependency relations and grammatical functions. Well-formed expressions of functions and relations provide constituent surroundings for syntactic categories in the form of two-way automata. These relations, functions, and automata are described in a special definition language.

In focusing on high level descriptions a linguist may ignore computational details of the parsing process. He writes the grammar into a DPL-description and a compiler translates it into efficient LISP-code. The environment has also a tracing facility for the parsing process, grammar-sensitive lexical maintenance programs, and routines for the interactive graphic display of parse trees and grammar definitions. Translator routines are also available for the transport of compiled code between various LISP-dialects. The environment itself exists currently in INTERLISP and FRANZLISP. This paper focuses on knowledge engineering issues and does not enter linguistic argumentation.

INTRODUCTION

Our objective has been to build a parser for Finnish to work as a practical tool in real production applications. In the beginning of our work we were faced with two major problems. First, so far there was no formal description of the Finnish grammar. Second difficulty was that Finnish differs by its structure greatly from the Indoeuropean languages. Finnish has relatively free word order and syntactico-semantic knowledge in a sentence is often expressed in the inflections of the words. Therefore existing parsing methods for Indoeuropean languages (eg. ATN, DCG, LFG etc.) did not seem to grasp the idiosyncracies of Finnish.

The parser system we have developed is based on functional dependency. Grammar is specified by a family of two-way finite automata and by dependency function and relation definitions. Each automaton expresses the valid dependency context of one constituent type. In abstract sense the working storage of the parser consists of two constituent stacks and of a register which holds the current constituent (Figure 1).

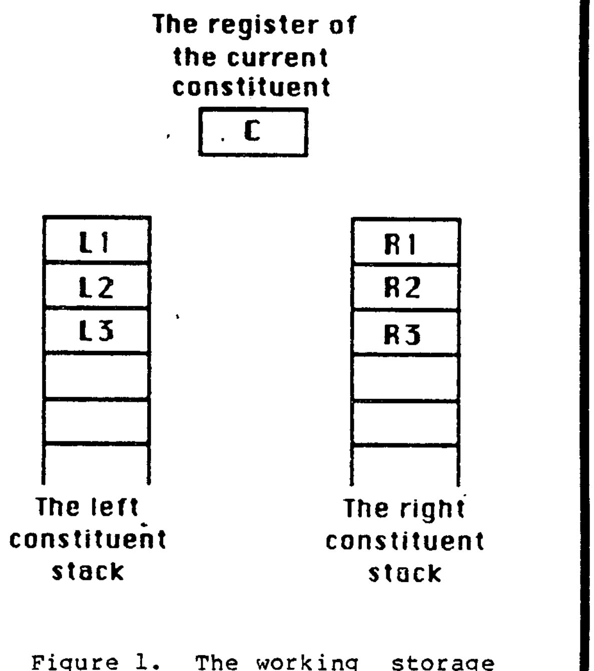

The register of the current constituent
C
L1
L2
L3
R1
R2
R3
The left constituent stack
The right constituent stack

Figure 1. The working storage of DPL-parsers

==========
(*) SITRA Foundation
P.O. Box 329, SF-00121 Helsinki, Finland

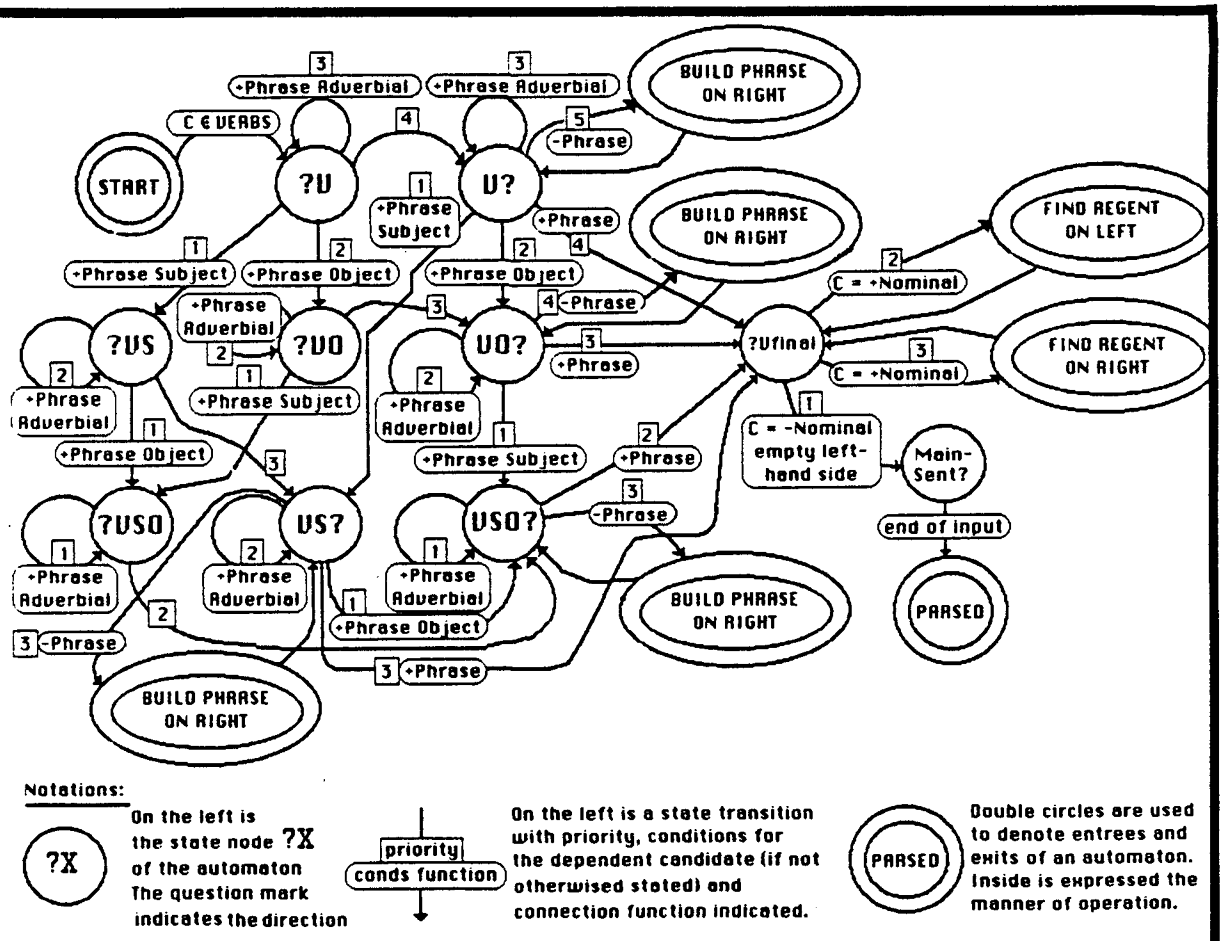


Figure 2. A two-way automaton for Finnish verbs

The two stacks hold the right and left contexts of the current constituent. The parsing process is always directed by the expectations of the current constituent. Dynamic local control is realized by permitting the automata to activate one another. The basic decision for the automaton associated with the current constituent is to accept or reject a neighbor via a valid syntactico-semantic subordinate relation. Acceptance subordinates the neighbor, and it disappears from the stack. The structure an input sentence receives is an annotated tree of such binary relations.

An automaton for verbs is described in Figure 2. When a verb becomes the current constituent for the first time it will enter the automaton through the START node. The automaton expects to find a dependent from the left (?V). If the left neighbor has the constituent feature +Phrase, it will be tested first for Subject and then for Object. When a function test succeeds, the neighbor will be subordinated and the verb advances to the state indicated by arcs. The double circle states denote entry and exit points of the automaton.

If completed constituents do not exist as neighbors, an automaton may defer decision. In the Figure 2 states labelled "BUILD PHRASE ON RIGHT" and "FIND REGENT ON RIGHT" push the verb to the left stack and pop the right stack for the current constituent. When the verb is activated later on, the control flow will continue from the state expressed in the deactivation command.

There are two distinct search strategies involved. If a single parse is sufficient, the graphs (i.e. the automata) are searched depth first following the priority numbering. A full search is also possible.

The functions, relations and automata are expressed in a special conditional expression formalism DPL (for Dependency Parser Language). We believe that DPL might find applications in other inflectional languages as well.

DPL-DESCRIPTIONS

The main object in DPL is a constituent. A grammar specification opens with the structural descriptions of constituents and the allowed property names and property values. User may specify simple properties, features or categories. The structures of the lexical entries are also defined at the beginning. The syntax of these declarations can be seen in Figure 3.

All properties of constituents may be referred in a uniform manner using their values straight. The system automatically takes into account the computational details associated to property types. For example, the system is automatically tuned to notice the inheritance of properties in their hierarchies. Extensive support to multidimensional analysis has been one of the central objectives in the design of the DPL-formalism. Patterning can be done in multiple dimensions and the property set associated to constituents can easily be extended.

An example of a constituent structure and its property definitions is given in Figure 4. The description states first that each constituent contains Function, Role, ConstFeat, PropOfLexeme and MorphChar. The next two following definitions further specify ConstFeat and PropOfLexeme. In the last part the definition of a category tree SemCat is given. This tree has sets of property values associated with nodes. The DPL-system automatically takes care of their inheritances. Thus for a constituent that belongs to the semantic category Human the system automatically associates feature values +Hum, +Anim, +Countable, and +Concr.

The binary grammatical functions and relations are defined using the syntax in Figure 5. A DPL-function returns as its value the binary construct built from the current constituent (C) and its dependent candidate (D), or it returns NIL. DPL-relations return as their values the pairs of C and D constituents that have passed the associated predicate filter.

By choosing operators a user may vary a predication between simple equality (=) and equality with ambiguity elimination (=:=). Operators := and :- denote replacement and insertion, respectively. In predicate expressions angle brackets signal the scope of an implicit OR-operator and parentheses that of an

```
<constituent structure> ::= ( CONSTITUENT:   <list of properties>.. )
<subtree of constituent>::= ( SUBTREE:       <glue node>
                                             <list of properties> ) |
                            ( LEXICON-ENTRY: <glue node>
                                             <list of properties> )
<list of properties>      ::= ( <list of properties>.. ) |
                              ( <property name>.. )
<property name>           ::= <type name> | <glue node name>
<type name>               ::= <unique lisp atom>
<glue node name>          ::= <unique lisp atom>
<glue node>               ::= <glue node name in upper level>

<property declaration>    ::= ( PROPERTY: <type name> <possible values> ) |
                              ( FEATURE:  <type name> <possible values> ) |
                              ( CATEGORY: <type name> < <node definition>.. > )
<possible values>         ::= < <default value> <unique lisp atom>.. >
<default value  >         ::= NoDefault | <unique lisp atom>
<node definition>         ::= ( <node name> <feature set> <father node> )
<node name>               ::= <unique lisp atom>
<feature set>             ::= ( <feature value> ) | <empty>
<father node>             ::= / <name of an already defined node> | <empty>
<empty>                   ::=
```

Figure 3. The syntax of constituent structure and property definitions

```
(CONSTITUENT:     (Function Role ConstFeat PropOfLexeme Morphchar))

(LEXICON-ENTRY:   PropOfLexeme
                  ( (SyntCat SyntFeat)
                    (SemCat SemFeat)
                    (FrameCat LexFrame)
                    AKO ))

(SUBTREE:         MorphChar
                  ( Polar Voice Modal Tense Comparison
                    Number Case PersonN PersonP Clit1 Clit2))

(CATEGORY:        SemCat
                < ( Entity )
                  ( Concrete ( +Concr ) / Entity )
                      ( Animate  ( +Anim +Countable ) / Concrete )
                           ( Human ( +Hum ) / Animate )
                           ( Animals / Animate )
                      ( NonAnim  / Concrete )
                           ( Matter ( -Countable ) / NonAnim )
                           ( Thing ( +Countable ) / NonAnim ) >
)
```

Figure 4. An example of a constituent structure specification and the definition of an category tree

implicit AND-operator. An arrow triggers defaults on: the elements of expressions to the right of an arrow are in the OR-relation and those to the left of it are in the AND-relation. Two kinds of arrows are in use. A simple arrow (->) performs <u>all</u> operations on the right and a double arrow (=>) terminates the execution at the first successful operation.

In Figure 6 is an example of how one may define Subject. If the relation RecSubj holds between the regent and the dependent candidate the latter will be labelled Subject and subordinated to the former. The relational expression RecSubj defines the property patterns the constituents should match.

A grammar definition ends with the context specifications of constituents expressed as two-way automata. The automata are described using the notation shown in somewhat simplified form in Figure 7. An automaton can refer up to three constituents to the right or left using indexed names: L1, L2, L3, R1, R2 or R3.

```
<function>           ::= ( FUNCTION: <function name> <operation expr> )
<relation>           ::= ( RELATION: <relation name> <operation expr> )
<operation expr>     ::= ( <predicate expr>.. <impl> <operation expr>.. ) ¦
                         <predicate expr> ¦
                         <relation name> ¦
                         ( DEL <constituent label> )
<predicate expr>     ::= < <predicate expr> > ¦
                         ( <predicate expr> ) ¦
                         ( <constituent pointer> <operator> <value expr>)
<impl>               ::= -> ¦ =>
<constituent label>::= C ¦ D
<operator>           ::= = ¦ := ¦ :- ¦ =:=
<value expr>         ::= < <value expr>.. > ¦
                         ( <value expr>.. ) ¦
                         <value of some property> ¦
                         '<lexeme> ¦
                         ( <property name> <constituent label> )
```

Figure 5. The syntax of DPL-functions and DPL-relations

```
(FUNCTION:        Subject
                  ( RecSubj  -> (D := Subject))
)

(RELATION:        RecSubj
                  ((C = Act < Ind Cond Pot Imper >) (D = -Sentence +Nominal)
                    -> ((D = Nom)
                          -> (D = PersPron (PersonP C) (PersonN C))
                            ((D = Noun) (C = 3P) -> ((C = S) (D = SG))
                                                    ((C = P) (D = PL))))
                       ((D = Part) (C = S 3P)
                          -> ((C = 'OLLA)
                               => (C :- +Existence))
                            ((C = -Transitive +Existence))))
)
```

Figure 6. A realisation of Subject

```
<state in autom.>::= ( STATE: <state name> <direction> <state expr>.. )
<direction>       ::= LEFT ! RIGHT
<state expr>      ::= ( <lhs of s. expr> <impl> <state expr>.. ) !
                      ( <lhs of s. expr> <impl> <state change> ) !
<lhs of s. expr> ::= <function name> ! <predicate expr>..
<state change>    ::= ( C := <name of next state> ) !
                      ( FIND-REG-ON     <direction> <sstate ch.> ) !
                      ( BUILD-PHRASE-ON <direction> <sstate ch.> ) !
                      ( PARSED )
<state change>    ::= <work sp. manip.> <state change>
<sstate ch.>      ::= ( C := <name of return state> )
<work sp. manip.>::= ( DEL <constituent label>        ) !
                      ( TRANSPOSE <constituent label>
                                  <constituent label> )
```

Figure 7. Simplified syntax of state specifications

```
(STATE:           V? RIGHT
                  ((D = +Phrase) -> (Subject   -> (C := VS?))
                                    (Object    -> (C := VO?))
                                    (Adverbial -> (C := V?))
                                    (T         => (C := ?VFinal)))
                  ((D = -Phrase) -> (BUILD-PHRASE-ON RIGHT (C := V?)))
)
```

Figure 8. The expression of V? in Figure 2.

The direction of a state (see Figure 2.) selects the dependent candidate normally as L1 or R1. A switch of state takes place by an assignment in the same way as linguistic properties are assigned. As an example the node V? of Figure 2 is defined formally in Figure 8.

More linguistically oriented argumentation of the DPL-formalism appears elsewhere (Nelimarkka, 1984a, and Nelimarkka, 1984b).

THE ARCHITECTURE OF THE DPL-ENVIRONMENT

The architecture of the DPL-environment is described schematically in Figure 9. The main parts are highlighted by heavy lines. Single arrows represent data transfer; double arrows indicate the production of data structures. All modules have been implemented in LISP. The realisations do not rely on specifics of underlying LISP-environments.

The DPL-compiler

A compilation results in executable code of a parser. The compiler produces highly optimized code (Lehtola, 1984). Internally data structures are only partly dynamic for the reason of fast information fetch. Ambiguities are expressed locally to minimize redundant search. The principle of structure sharing is followed whenever new data structures are built. In the manipulation of constituent structures there exists a special service routine for each combination of property and predication types. These routines take special care of time and memory consumption. For instance with regard replacements and insertions the copying includes physically only the path from the root of the list structure to the changed sublist. The logically shared parts will be shared also physically. This stipulation minimizes memory usage.

In the state transition network level the search is done depth first. To handle ambiquities DPL-functions and -relations process all alternative interpretations in parallel. In fact the alternatives are stored in the stacks and in the C-register as trees of alternants.

In the first version of the DPL-compiler the generation rules were intermixed with the compiler code. The maintenance of the compiler grew harder when we experimented with new computational features. We

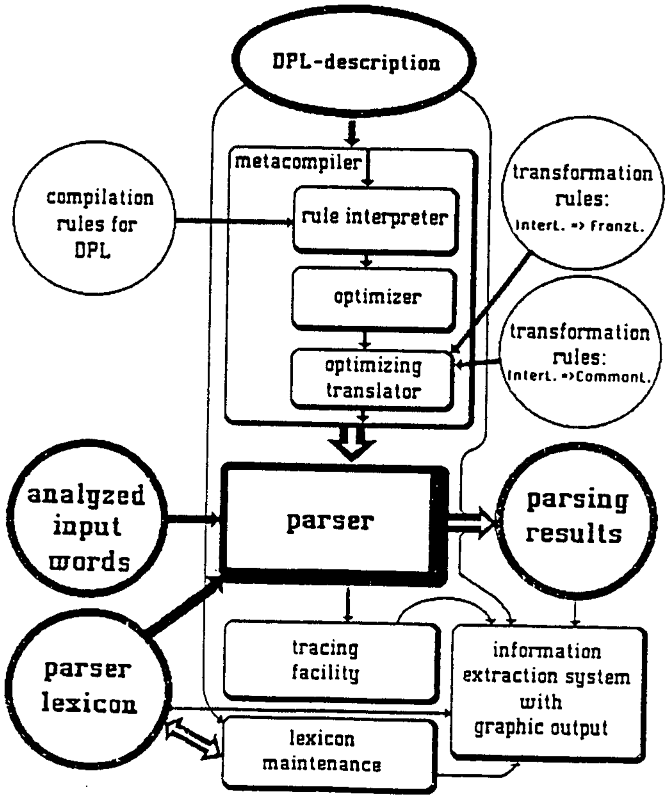


Figure 9. The architecture of the DPL-environment

therefore started to develop a metacompiler in which compilation is defined by rules. At moment we are testing it and soon it will be in everyday use. The amount of LISP-code has greatly reduced with the rule based approach, and we are now planning to install the DPL-environment into IBM PC.

Our parsers were aimed to be practical tools in real production applications. It was hence important to make the produced programs transferable. As of now we have a rule-based translator which converts parsers between LISP dialects. The translator accepts currently INTERLISP, FranzLISP and Common Lisp.

Lexicon and its Maintenance

The environment has a special maintenance program for lexicons. The program uses video graphics to ease updating and it performs various checks to guarantee the consistency of the lexical entries. It also co-operates with the information extraction system to help the user in the selection of properties.

The Tracing Facility

The tracing facility is a convenient tool for grammar debugging. For example, in Figure 10 appears the trace of the parsing of the sentence "Poikani tuli illalla kentältä heittämästä kiekkoa." (= "My son

```
_(T POIKANI TULI ILLALLA KENTÄLTÄ HEITTÄMÄSTÄ KIEKKOA .)

383 conses
.03 seconds
0.0 seconds, garbage collection time
PARSED
_PATH()

=>  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
(POIKA)  <=  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  N?
=>  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
(##)  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  NIL
(POIKA)  =>  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?V
=>  ((POIKA) TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?VS
((POIKA) TULLA)  <=  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA)  =>  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
((POIKA) TULLA)  (ILTA)  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  N?
((POIKA) TULLA)  =>  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
((POIKA) TULLA)  <=  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA))  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA))  =>  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
((POIKA) TULLA (ILTA))  (KENTTÄ)  <=  (HEITTÄÄ)  (KIEKKO)  N?
((POIKA) TULLA (ILTA))  =>  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
((POIKA) TULLA (ILTA))  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA) (KENTTÄ))  <=  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA) (KENTTÄ))  =>  (HEITTÄÄ)  (KIEKKO)  ?V
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  <=  (KIEKKO)  V?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  =>  (KIEKKO)  ?N
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  (KIEKKO)  <=  N?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  =>  (KIEKKO)  ?NFinal
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  <=  (KIEKKO)  V?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ (KIEKKO))  <=  VO?
((POIKA) TULLA (ILTA) (KENTTÄ))  =>  (HEITTÄÄ (KIEKKO))  ?VFinal
((POIKA) TULLA (ILTA) (KENTTÄ))  <=  (HEITTÄÄ (KIEKKO))  VS?
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  VS?
=>  ((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  ?VFinal
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  MainSent?
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  MainSent? OK
DONE
```

Figure 10. A trace of parsing process

came back in the evening from the stadium where he had been throwing the discus."). Each row represents a state of the parser before the control enters the state mentioned on the right-hand column. The thus-far found constituents are shown by the parenthesis. An arrow head points from a dependent candidate (one which is subjected to dependency tests) towards the current constituent.

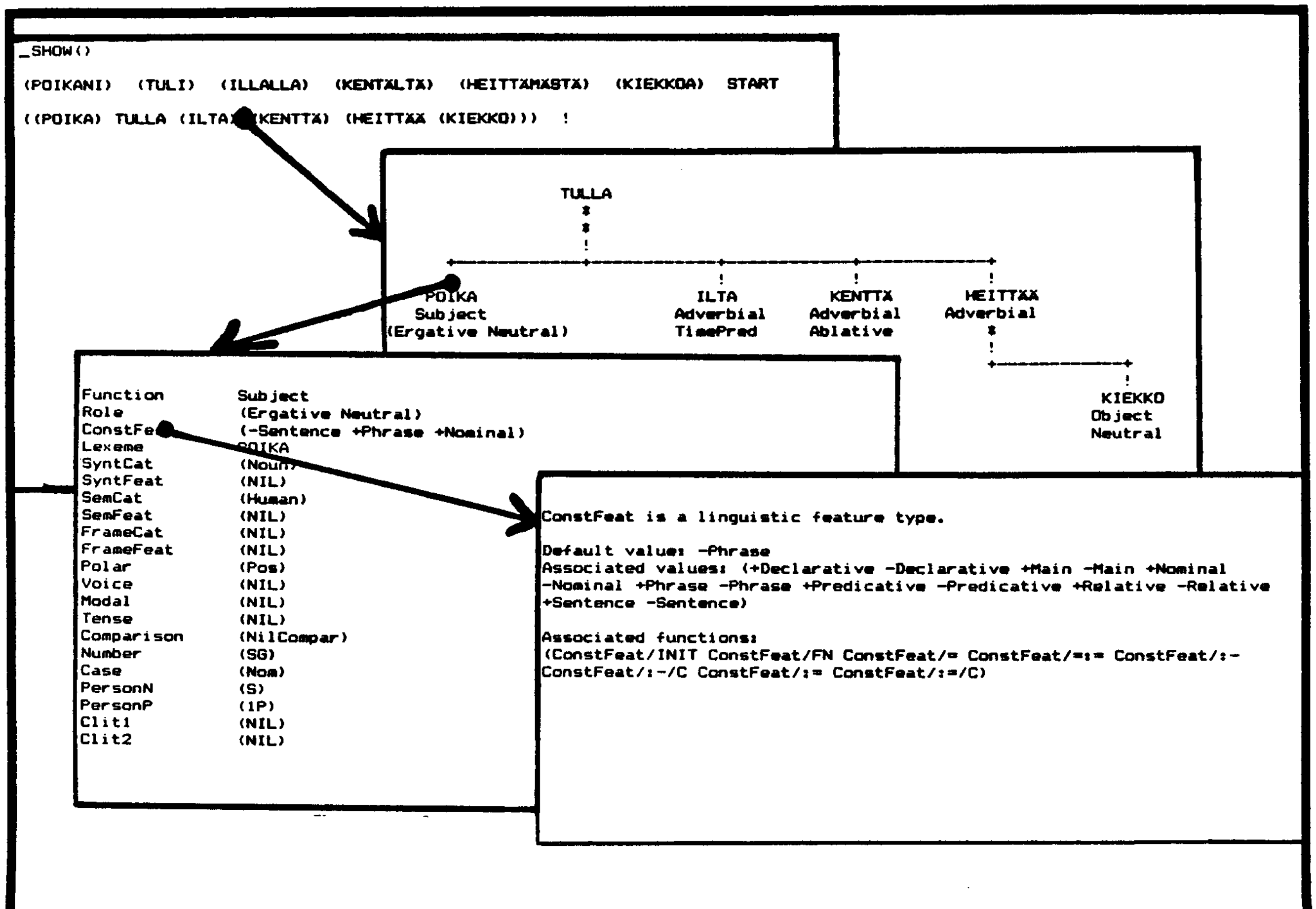
The tracing facility gives also the consumed CPU-time and two quality indicators: search efficiency and connection efficiency. Search efficiency is 100%, if no useless state transitions took place in the search. This figure is meaningless when the system is parameterized to full search because then all transitions are tried.

Connection efficiency is the ratio of the number of connections remaining in a result to the total number of connections attempted for it during the search. We are currently developing other measuring tools to extract statistical information, eg. about the frequency distribution of different constructs. Under development is also automatic book-keeping of all sentences input to the system. These will be divided into two groups: parsed and not parsed. The first group constitutes growing test material to ensure monotonic improvement of grammars: after a non trivial change is done in the grammar, a new compiled parser runs all test sentences and the results are compared to the previous ones.

Information Extraction System

In an actual working situation there may be thousands of linguistic symbols in the work space. To make such a complex manageable, we have implemented an information system that for a given symbol pretty-prints all information associated with it.

The environment has routines for the graphic display of parsing results. A user can select information by pointing with the cursor. The example in Figure 11 demonstrates the use of this facility. The command SHOW() inquires the results of



```
_SHOW()

(POIKANI)  (TULI)  (ILLALLA)  (KENTÄLTÄ)  (HEITTÄMÄSTÄ)  (KIEKKOA)  START

((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  !
```

```
                 TULLA
                   *
                   *
                   !
        +----------+----------+----------+----------+
        !                     !          !          !
      POIKA                 ILTA       KENTTÄ     HEITTÄÄ
     Subject             Adverbial   Adverbial   Adverbial
(Ergative Neutral)        TimePred    Ablative       *
                                                     !
                                                     +----------+
                                                                !
                                                             KIEKKO
                                                             Object
                                                             Neutral
```

```
Function      Subject
Role          (Ergative Neutral)
ConstFeat     (-Sentence +Phrase +Nominal)
Lexeme        POIKA
SyntCat       (Noun)
SyntFeat      (NIL)
SemCat        (Human)
SemFeat       (NIL)
FrameCat      (NIL)
FrameFeat     (NIL)
Polar         (Pos)
Voice         (NIL)
Modal         (NIL)
Tense         (NIL)
Comparison    (NilCompar)
Number        (SG)
Case          (Nom)
PersonN       (S)
PersonP       (1P)
Clit1         (NIL)
Clit2         (NIL)
```

```
ConstFeat is a linguistic feature type.

Default value: -Phrase
Associated values: (+Declarative -Declarative +Main -Main +Nominal
-Nominal +Phrase -Phrase +Predicative -Predicative +Relative -Relative
+Sentence -Sentence)

Associated functions:
(ConstFeat/INIT ConstFeat/FN ConstFeat/= ConstFeat/=:= ConstFeat/:-
ConstFeat/:-/C ConstFeat/:= ConstFeat/:=/C)
```

Figure 11. An example of information extraction utilities

the parsing process described in Figure 10. The system replies by first printing the start state and then the found result(s) in compressed form. The cursor has been moved on top of this parse and CTRL-G has been typed. The system now draws the picture of the tree structure. Subsequently one of the nodes has been opened. The properties of the node POIKA appear pretty-printed. The user has furthermore asked information about the property type ConstFeat. All these operations are general; they do not use the special features of any particular terminal.

CONCLUSION

The parsing strategy applied for the DPL-formalism was originally viewed as a cognitive model. It has proved to result practical and efficient parsers as well. Experiments with a non-trivial set of Finnish sentence structures have been performed both on DEC-2060 and on VAX-11/780 systems. The analysis of an eight word sentence, for instance, takes between 20 and 600 ms of DEC CPU-time in the INTERLISP-version depending on whether one wants only the first or, through complete search, all parses for structurally ambiguous sentences. The MacLISP-version of the parser runs about 20 % faster on the same computer. The NIL-version (Common Lisp compatible) is about 5 times slower on VAX. The whole environment has been transferred also to FranzLISP on VAX. We have not yet focused on optimality issues in grammar descriptions. We believe that by rearranging the orderings of expectations in the automata improvement in efficiency ensues.

REFERENCES

1. Lehtola, A., Compilation and Implementation of 2-way Tree Automata for the Parsing of Finnish. M.S. Thesis, Helsinki University of Technology, Department of Physics, 1984, 120 p. (in Finnish)
2. Nelimarkka, E., Jäppinen, H. and Lehtola A., Two-way Finite Automata and Dependency Theory: A Parsing Method for Inflectional Free Word Order Languages. Proc. COLING84/ACL, Stanford, 1984a, pp. 389-392.
3. Nelimarkka, E., Jäppinen, H. and Lehtola A., Parsing an Inflectional Free Word Order Language with Two-way Finite Automata. Proc. of the 6th European Conference on Artificial Intelligence, Pisa, 1984b, pp. 167-176.
4. Winograd, T., Language as a Cognitive Process. Volume I: Syntax, Addison-Wesley Publishing Company, Reading, 1983, 640 p.