[*International Conference on the State of Machine Translation in America, Asia and Europe*. Proceedings of IAI-MT86, 20-22 August 1986, Bürgerhaus, Dudweiler]

Poul Andersen

Presentation of EUROTRA-DK's ECS component

EUROTRA-DK
Njalsgade 80
DK-2300 Copenhagen

S U M M A R Y   S H E E T

Author : Poul Andersen, EUROTRA-DK.

Situation as of 28 May 1986.

Name of the system : EUROTRA-DK, ECS-level.

Status : research.

Type of system : transfer based / multilingual.

Translated languages : (only parsing of Danish).

Speed of the system : (no reliable data available).

Costs : financial costs not available
/ appr. 6 person-months of development work

Type of analysis output : configurational structure =

lexical and phrasal categories + word order

Dictionaries : 3.000-4.000 entries/wordforms = 500-600 words

Data bases with rules : appr. 40 grammar rules for analysis

Implementation language : C-prolog

Operating system : ULTRIX version 1.0 (appr. = UNIX BSD 4.2)

Type of hardware/equipment : VAX 11/750

INTRODUCTION

EUROTRA-DK's ECS-component is part of EUROTRA's analysis and generation module for Danish, which again is one of 7 analysis/ generation modules presently being built in the different EEC member states. These modules, together with 42 transfer modules, constitute the EUROTRA MT system, for which a small prototype is planned to be ready at the end of 1987 and an operational system covering one subject field is planned for 1989/90. The subject of this paper, consequently, is not a complete system, but one component of a large system. This is in perfect agreement with modularity being one of the fundamental principles behind EUROTRA. EUROTRA is conceived as a system, built of several, distinct components that can be developed - and described - independently of one another.

The Danish ECS component as here described was developed by three linguists, Hanne Jensen, Henrik Selsoe Soerensen and Poul Andersen, with help from the computer scientist Gunner Helweg Johansen and advice from the computational linguists Bente Maegaard and Ebbe Spang-Hanssen. This presentation is the sole responsibility of the author.

2. PRELIMINARIES

1. Coverage

EUROTRA MT can be said to be modular in at least three respects :

1. It is composed of an analysis/generation module for each language and a transfer module for each language pair.

2. Each analysis/generation module is split up in a number of representational levels. ECS is one of these levels.

3. Each representational level is developed in cycles.

The first level to be developed by all the language groups participating in EUROTRA is the ECS-level. The first cycle of developing ECS took place February – May 1986. The second cycle will probably take place appr. February - April 1987 and result in the ECS component of the small prototype (2.500 lexical units) which is planned for the end of 1987. During the following phase of the project, appr. 1988-1989, there will be an extension – in one or more cycles – of the system up to 20.000 lexical units. This makes it necessary to define partial goals for each cycle. The goal for the second cycle is the same as the goal for the present, second phase of the EUROTRA project: the language in the "ESPRIT-corpus" – some 100 pages of EC-Council decisions in the field of information technology, expected to yield appr. the mentioned 2.500 lexical units. The goal for the present, first cycle should represent 50% of the goal for the second cycle. At the same time it is highly desirable that all the language groups work on the same 50% in this first cycle.

This was the motivation for setting up a small group consisting of Bente Maegaard and Charlotte Toubro from EUROTRA-DK with the mandate to provide a 'euroversal' (i.e. universal within EUROTRA) definition of the grammars and lexicon of the first cycle. According to their definition, the grammar is restricted to main clauses + relative clauses without coordination on clause level, without personal and demonstrative pronouns (reference problems) and with VP's restricted to present and past tenses indicative. The Danish group tried to be as close as possible to the definition, which was found to correspond more or less to the vocabulary in the first of the texts in the ESPRIT corpus – 4 1/2 pages of text representing appr. 500 lexical units.

2. Definition of level

ECS is short for Eurotra Configurational Structure. Before we started to write a grammar and dictionary for the defined goal, we had to define exactly what kind of ECS level we wanted. The Danish ECS level is a purely configurational, surface syntactic level, where words and phrases are marked with their morphosyntactic category and properties that are relevant to their combination as morphosyntactic entities, and word order is preserved, but no other information is taken into account. This complies with a sound EUROTRA principle closely tied up with the above mentioned modularity principle, namely locality, meaning that each level should be defined independently of the other levels and only contain information relevant to that level. This is also in good accordance with The EUROTRA Reference Manual (below: RM), which is an internal document, among other things describing the representational levels into which the analysis/generation modules are split up. The description in RM is a starting point when defining a level in a specific language, but for several reasons RM cannot and should not contain an exhaustive definition of the lower levels at this stage, but should be evaluated by each language group with respect to how closely they can adhere to the given description and what they will have to provide themselves.

DESCRIPTION OF THE LEVEL

1. Tree geometry

Before we started working on ECS, we had experimented with implementing a complete mini-system, covering all representational levels, the so-called small scale experiment, also known affectionately as 'toy'. This work had given us some valuable experience:

- There is a great danger or solving problems on an ad hoc basis, writing very specific grammar rules. Already with a quite small corpus, this leads to an abundance of rules, as the number of possible combinations of constituents is very high, when word order is preserved. Apart from the theoretical drawback and the inconvenience of writing a lot of rules that take up a lot of space, it also means that the system has to look through a lot of rules that are hardly ever used. This problem becomes still more acute in cases where two rules start with the same constituents and only are distinguished by the presence or non-presence of some rightmost element. As the parser works from left to right, it will go along with the wrong rule till the very end of the sentence (or whatever entity is being parsed) and only then discover the missing or superfluous constituent. With very long runtimes, we became aware of this kind of problem at a very early stage.

The solution to this problem can be split up in two steps. The starting point is to have one, general rule for all sentences. After some discussion it was decided to build the sentence around the VP, defined as the constituent starting with the first finite verb and ending with possible particles going together with the verb, and having a constituent called XPP1 to the left of VP and a constituent called XPP2 to the right of VP. XPP1 and XPP2 are only being looked into at the next level of the tree, where VP also is being defined as possibly containing XPP3 and XPP4 apart from verbs and verbal particles. This description of VP mirrors the discontinuity of VP in Danish, which causes problems in a word order-preserving description. In Danish it is, however, much more restricted what can be inserted in the VP than in languages like German and Dutch, where our proposed solution might not be suitable.

Our overall derivation tree first looked like this:

```
                                          S
              ............................../|\.................................
           /                              |                                \
      XPP1                               VP                               (XPP2)
                 ..........................|..........................
               /        /                 |             \              \
          Vfinite    (XPP3)         (Vinfinite)      (XPP4)        (particle)
```

The different XPP's are rewritten on the next level of the tree as NP's, PP's, ADVP's and relative clauses separately and in combinations described by the grammar.

An example:

I 1981 har europaeiske firmaer omsider taget den udfordring op,
som ligger i dette emne.

(literal translation: In 1981 have European firms eventually taken
the challenge up, which lies in this subject.
-- > In 1981 European firms eventually took up the challenge posed by
this subject.)

The application of the S-rule + the VP-rule to this example yields:

XPP1 :	in 1981 (PP)
Vfinite :	have
XPP3 :	European firms eventually (NP + ADVP)
Vinfinite :	taken
XPP4 :	the challenge (NP)
particle :	up
XPP2 :	which lies in this subject (relative clause)

We may, however, push the generality one important step further, if we operate with one XPP instead of four different XPP's, and if we can rewrite this XPP as one NP, PP, ADVP, ADJP or relative clause and let these constituents combine freely in so far as they combine freely, in actual language. This is only possible if we can operate with optionality and iteration of constituents as described in RM. This possibility was not implemented in the official EUROTRA software available, but we implemented it locally in the Danish group. The result of this extension is the following, flatter derivation tree for a Danish declarative sentence, which at the same time is more general and carries more information:

```
                                          S
     ……………………………………/øØ………………………………
    /         /            /              ø          Ø            Ø          Ø
  /         /            /                ø          Ø            Ø          Ø
XPP*     Vfinite       (XPP)*        (Vinfinite)  (XPP)*   (particle)    (XPP)*
```

Brackets indicate optionality, Kleene star '*' indicates iteration, and the two together indicate that the constituent may be present 0, 1 or several times.

In traditional PS-rules the above tree together with the next level looks this way:

S => XPP* Vfinite (XPP)* (Vinfinite) (XPP)* (particle) (XPP)*

XPP => NP

XPP => PP

XPP => ADVP

XPP => ADJP

This model is not yet implemented in the Danish ECS. We have not split VP up on the level immediately under S, and relative clauses complicate the picture a little, as they do not occur as freely as the other constituents. They cannot be incorporated in NP, as they do not necessarily follow immediately after the NP that they are modifying. Furthermore, we found it necessary to introduce a bar level between XPP and NP, so our second rule is XPP => NPP. We shall return to this subject under II.3. Constructors.

2. Atoms

In EUROTRA, the terminal elements, the 'leaves' of the linguistic trees, are called atoms. The exact definition of 'atom' and its relationship to entities such as 'word', 'word form', 'lexical unit', 'dictionary entry' and 'reading' is not completely clarified within EUROTRA, and it cannot be the same for all representational levels. In Danish ECS an atom corresponds to a word form as it appears in the text, with the main exception that capital letters after full stop are changed to lowercase.

The format for Danish ECS-atoms is as follows:

```
<name> = (<lexical category>, <feature description>)
[<word form>]
```

An example:

```
forslag = (n, scat=no, gen=neu, num=sg, case=ge, def=df)
[forslagets]
```

The name of an atom is the uninflected form of a word. It is not unique, but common to the class of atoms made up by all the inflected forms of the word. The name together with the feature description contains the same information as the word form and can be thought of as a translation of the textual string / the word form into a more explicit representation. To write an atom for every word form may not seem very rational – it is both time-consuming and results in dictionaries some 5 to 10 times bigger than would be the case with 'atom = word' - dictionaries. However, the latter approach presupposes a morphological analysis of the textual words, and this morphological component, referred to as EMS (Eurotra Morphological Structure), has not yet been developed. As Danish does not have a very rich inflectional system, this drawback is more or less acceptable in our language, but in most other Western European languages apart from English the lack of EMS will make it far more difficult and time-consuming to build a credible prototype system, capable of parsing and translating sentences with constituents in varying tenses, numbers and persons. The development of EMS is consequently given high priority now. (cf. IV.1. Text <= => ECS).

It is not self-evident what information should be put into the feature description. It might be argued that we might as well put as much information into the description as possible, because even if this information is not needed immediately, it may turn out to be useful at some later stage in the development of this level or at some other representational level. We have, however, adhered to the abovementioned (in 1.2.) locality principle and only included information actually needed at the ECS level. As may be seen from the above example, the relevant information for nouns is: gender, number, case, definiteness. The description of verbs includes: mode, tense, valency, voice.

All lexical categories have an extra property called 'scat' (short for 'subcategorization'). This is available for other kinds of relevant information. We did not know the exact values of this property from the outset, but left it to the actual development work to supply these values. They are used to restrict the output from the grammar rules (cf. II.3. Constructors). At present, we operate with following scat's for nouns: abbr(eviation), rel(ative pronoun), currency, sing(ularia tantum), meas(ure), name, reflex(ive pronoun), specifier, part, year.

3. Constructors

The grammar rules that build linguistic trees out of atoms are called constructors. The format for the Danish ECS-constructors is as follows:

```
<constructor name> = (<phrasal     cat. construct>, <feature descr. construct>)[
                 (<phras/lex. cat.       argument/l>, <feature descr. argument/l>)
                 (<phras/lex. cat.       argument/2>, <feature descr. argument/2>)
                 (<phras/lex. cat.       argument/n>, <feature descr. argument/n>)]
```

An example:

```
np<detp adjp n> = (np, type=no, gen=G, num=N, case=C, def=D)[
        (detp, type=no, gen=G, num=N, case=nge, def=D)
        (adjp, type=no, gen=G, num=N, case=nge, def=D)
        (n, scat=no, gen=G, num=N, case=C, def=idf)]
```

The constructor name corresponds to a PS-rule, in the quoted example :

NP => DETP ADJP N

Capital letters in the feature descriptions represent variables. The example shows that when a Danish NP is composed of a determiner (e.g. an article), an adjective, and a noun, then the determiner and the adjective should be in non-genitive case, as only the last element in an NP is marked for case, and the noun should be without the definite ending, as the definite ending only can be present when there is no determiner in front of the noun. The two remaining properties, gender and number, should take the same value in all three arguments, and this value should be inserted in the feature description of the NP together with the case of the noun and the definiteness of the determiner and adjective.

At present we have written appr. 40 constructors. The number of constructors does not say anything about the coverage of the grammar – as pointed out in II.1. Tree geometry – because a small set of general constructors can analyse more different sentences than a lot of very specific constructors. Generality should, however, not be confused with 'unspecifiedness'. With unspecified constructors too many ungrammatical sentences will get an analysis and grammatical sentences will get too many analyses besides the correct one(s). By way of example, one of the Danish constructors looks like this :

```
npp <adjp detp cardp- aggr4- adjp* np> = (npp, type=S, case=nge, def=df, gen=G, num=N)[
    (adjp, type=tot, case=nge, deg=pos, def=df, gen=G, num=N)
    (detp, case=nge, def=df, gen=G, num=N)
    (cardp, num=pl)-
    (aggr4, def=df, num=N)-
    (adjp, type=no, case=nge, def=df, gen=G, num=N)*
    (np, type=S, case=nge, gen=G, num=N)]
```

This constructor is used for e.g.

```
næsten alle  de tre    af   Rådet        analyserede nyere   og    helt          nye forslag
almost  all  the three by   the Council analyzed     newer  and  completely new proposals
adv      adj  det card  prep det   n         prtc          adj    (coord) adv        adj     n
  \     /       |    |       |      \  /          /              |                   \      /        |
    |           |    |       |       NP        /               |                      |           |
    |           |    |        \      /        /                 |                      |           |
    |           |    |            PP        /                   |                      |           |
    |           |    |              \      /                    |                      |           |
 ADJP   DETP CARDP        AGGR4                 ADJP                   ADJP        NP
```

Although this constructor may look rather specific, it is very general thanks to several factors:

- CARDP and AGGR4 are optional,
- the second ADJP is optional and may be repeated,
- all constituents are in their turn described by other constructors; e.g. ADJP may consist of an adjective or of an adverb + an adjective.

We have introduced certain AGGR's, in the above example AGGR4, in order to make the parsing more efficient. The idea is to make the use of a constructor dependent on some distinctive leftmost element. As the parser starts from the left, it will only start trying this constructor, if it finds this leftmost element, in the case of AGGR4 a preposition, and then go on looking for an NP and a participle of a transitive verb.

The NP's in the NPP-constructor are built by one of the following NP-constructors :

```
np  <n  adjp* n> = (np, type=S, case=C, def=D, gen=G, num=N)[
         (n, scat=specifier, case=nge, def=D, gen=G, num=N)
         (adjp, type=no, case=nge, def=D, gen=G, num=N)*
         (n, scat=S, case=C)]

np <n> = (np, type=S, case=C, def=D, gen=G, num=N)[
         (n, scat=S, case=C, def=D, gen=G, num=N)]
```

The first of these are used for NPP's ending with e.g.

```
          antal        gode forslag
   (... number (of) good proposals)
```

This demonstrates the need for 'scat=specifier' for nouns like 'antal', 'flaske' ('flasket god vin' = 'bottles (of) good wine') etc.

The degree of specificity in the constructors should also be seen under another aspect, namely analysis vs. generation. If our grammar only was supposed to analyse grammatically correct input, we did not need specify e.g. the internal order of various categories of adjectives and determiners in front of the NP, as they already would be ordered in the input NPP. However, our grammar should also produce correct output in generation, and this makes it necessary to distinguish e.g. between adjectives with 'scat=tot(ality)' (like 'alle', 'hele', 'begge') and other adjectives, in order only to generate the NPP 'alle de gode forslag' and avoid NPP's like *'gode de alle forslag' or *'de alle gode forslag'.

METHOD OF WORK

1. Organization of work

A special UNIX-directory called 'ecs' was created with four subdirectories:

ecs/a	-	atom files
ecs/c	-	constructor files
ecs/r	-	test input files
ecs/o	-	test output files

The three linguistic implementors wrote atoms and constructors directly into files in subdirectory a and c, using a local version of the user language as shown above in II.2. and II.3. These files are named <filename>.u and they are transformed to prolog format by means of a locally developed preprocessor, which places the output in a file with the same name without '.u'. For instance, 'ecs/a/n.u' contains atoms for nouns in user language and is transformed to the file 'ecs/a/n' with the command 'eprep n.u'.

2. Special tools

It is of great importance, that the linguistic implementors are free to concentrate on linguistic or strategic issues and not hampered by a lot of routine work. This can, of course, be achieved by hiring supplementary staff for writing constructors and atoms according to instructions from the linguists. Another solution, which we chose, is to develop tools that maximally facilitate the tedious routine work. We used our editor, emacs, to define some special commands as described in ANNEX 1.

3. Testing

We attach great importance to testing atoms and constructors in parallel with the development work as it in practice is impossible to foresee where things will go wrong without actually running the grammar on a computer. This testing was done very efficiently in the following way :

A simple command script was written and put in a file in the ecs-directory. When this script, called 'run', is invoked with an argument, e.g. 'run test', the parsing and translation algorithm, written in prolog, is started and the argument is looked for as a file name in subdirectory ecs/r. The file, here called 'test', will typically contain a series of prolog commands adding the necessary atom and constructor files followed by a set of NP's and/or sentences to be tested. Other prolog commands may be inserted, e.g. 'tv' (= toggle verbose, i.e. no trace) and 'statistics' (printing out runtimes and resources/space occupied). In our case the testing just consisted in parsing at the ECS-level without any translation between levels – this is specified in the test file. The program is run as a background job with lowered priority, and the output produced by the parsing algorithm is written to a file with the same name as the input file, but located in the ecs/o subdirectory. Examples of a 'run'-script, a test input file and a test output file is found in ANNEX 2.

The advantages with this arrangement are several :

- a fixed set of NP's and sentences of different degrees of complexity up to the defined goal is tested regularly as atoms and constructors are added/corrected, and it is easy to check the adequateness and consistency of atoms and constructors at a given moment.

- the testing is run as a background job, demanding no human interference and leaving the terminal free for other jobs. This aspect is of special importance at the present stage, where we use an experimental software, sometimes giving runtimes up to several days for a set of 20-30 sentences.

- the typing of the test material is done once and for all.

- the test situation gets as close as possible to a real MT situation with a fully automatic system.

ECS IN RELATION TO OTHER LEVELS

Each analysis/generation module consists of a number of levels and ECS is just one of these levels. The output from level x serves as input to level y in analysis, whereas in generation the output from level y serves as input to level x. It is, consequently, not without interest how the levels relate to each other, although they, as mentioned in 1.2. (locality principle), should be defined independently of each other.

1. Text <= => ECS

The levels closer to the actual text is presently being explored. We do not know exactly how they are going to be implemented, but it may in a certain sense be regarded as less crucial for a MT system than the implementation of the 'deeper' levels, which are going to serve as the basis for transfer between different natural languages, the 'real' translation. In analysis, the input to ECS expected from the lower levels is now given manually as a sequence of word forms in the same format as they appear in the atoms, separated by commas, as may be seen from the example in ANNEX 2. In generation, the output from ECS similarly is presented as a sequence of word forms without punctuation, without capital letter in the beginning of a sentence etc. The transformation of real text into input to ECS and the transformation of output from ECS into real text can be regarded as a relatively minor, technical problem, and it is conceivable that this can be done within the ECS level.

What is left, is a proper morphological analysis, referred to as EMS (Eurotra Morphological Structure). The purpose of this level would be to compute the values that now are entered manually into the dictionary. To take the example above (in II.2. Atoms), the word form 'forslagets' with the description 'forslag = (n, scat=no, gen=neu, num=sg, case=ge, def=df)' would be analysed as composed of three morphemes yielding each their part of this description:

'forslag' --> 'name=forslag, cat=n, scat=no, gen=neu',
'et' --> 'cat=n, gen=neu, num=sg, def=df,
's' --> 'cat=n, case=ge'.
Developing a morphological module is, however, a comprehensive computational linguistic task, which will demand substantial manpower and time. As mentioned under II.2., this module might be left out or restricted in some way at the cost of having more ECS-atoms. For instance, many entries for verbs could be avoided by leaving out first and second person of verbs, if these forms are unlikely to appear in the text types the system is being built for.

Apart from flexion, morphology also includes derivation and compounding. This is not the right place to go into details on EMS, but it is not inconceivable that we for a certain period will restrict EMS to a subset of the phenomena that ideally should be treated in morphological analysis.

2. ECS <= => relational structure/deep syntax

Within EUROTRA, it is not conceivable to use a superficial constituent structure directly as basis for transfer, so we need one or two 'deeper' levels. The first of these is referred to as ERS (Eurotra Relational Structure). This level is now being investigated in the language groups. Certain problems can be envisaged when going from ECS to ERS, and their solution may lead to adaptations at the ECS level, notwithstanding the locality principle. By way of example, the translation from ECS to ERS of constituents with iteration may cause problems in cases where these constituents are not translated to constructs belonging to one and the same category at ERS.

Reference:
For further information on EUROTRA in general and ECS in particular, see the periodical Multilingua
(special issue on EUROTRA in course of publication).

ANNEX 1

How to write ECS-constructors and atoms with self-defined emacs-functions

(Explanations are written in UPPERCASE, manual input to the machine is underlined, and machine output is within ' '.

Example: How to write a constructor that builds an NP out of an optional determiner + an optional adjective or several adjectives + a noun.

```
EMACS-COMMAND: npc

OUTPUT:
'np<> = (np, type=no, gen=G, num=N, case=C, def=D)['

PROMPT:  'enter  arg-cat  :'

INPUT:  detp-

OUTPUT:
'np<detp-> = (np, type=no, gen=G, num=N, case=C, def=D)[
    (detp, type=no, gen=G, num=N, case=nge, def=D)-'

PROMPT:  'enter arg-cat  :'

INPUT:  adjp*

OUTPUT:
'np<detp- adjp*> = (np, type=no, gen=G, num=N, case=C, def=D)[
    (detp, type=no, gen=G, num=N, case=nge, def=D)-
    (adjp, type=no, gen=G, num=N, case=nge, def=D)*'

PROMPT:  'enter  arg-cat  :'

INPUT:  n

OUTPUT:
'np<detp- adjp* n> = (np, type=no, gen=G, num=N, case=C, def=D)[
    (detp, type=no, gen=G, num=N, case=nge, def=D)-
    (adjp, type=no, gen=G, num=N, case=nge, def=)*
    (n, scat=no, gen=G, num=N, case=C, def=D)'

PROMPT: 'enter arg-cat :'

INPUT: (carriage return)

OUTPUT :
'np<detp- adjp* n> = (np, type=no, gen=G, num=N, case=C, def=D)[
    (detp, type=no, gen=G, num=N, case=nge, def=D)-
    (adjp, type=no, gen=G, num=N, case=nge, def=D)*
    (n, scat=no, gen=G, num=N, case=C, def=D)]'
```

```
NB:     -   argument occurs 0 or 1 time           | extensions implemented
        * = argument occurs 0, 1 or more times  | locally in the parser
        + = argument occurs 1 or more times       |
```

Example: How to write the fullfledged paradigm (number = [sing, plur] X case = [nongenitive, genitive] X definiteness = [indefinite, definite]) for a noun of common gender.

EMACS-COMMAND: nca

PROMPT: 'enter lexical unit :'

INPUT: <u>meddelelse</u>

PROMPT: 'enter subcat :'

INPUT: no

OUTPUT:

```
'meddelelse = (n, scat=no, gen=com, num=sg, case=nge, def=idf)
[meddelelse ]

meddelelse = (n, scat=no, gen=com, num=sg, case=nge, def=df)
[meddelelsen]

meddelelse = (n, scat=no, gen=com, num=sg, case=ge, def=idf)
[meddelelses,]

meddelelse = (n, scat=no, gen=com, num=sg, case=ge, def=df)
[meddelelsens ]

meddelelse = (n, scat=no, gen=com, num=pl, case=nge, def=idf)
[meddelelser]

meddelelse = (n, scat=no, gen=com, num=pl, case=nge, def=df)
[meddelelserne]

meddelelse = (n, scat=no, gen=com, num=pl, case=ge, def=idf)
[meddelelsers]

meddelelse = (n, scat=no, gen=com, num=pl, case=ge, def=df)
[meddelelsernes]'
```

THE CURSOR STOPS AT THE FIRST ']' AND WAITS FOR THE APPROPRIATE ENDING.
THE CURSOR IS MOVED FORWARD TO THE NEXT ']' WHEN PRESSING 'CARRIAGE RETURN'.

(2

<u>'run'-script   :</u>

```
#   /?/?/?/run
#
# usage:  run file
#
# comment:  priority of the evm is reduced
#
#   action: evm is started,
#             input is taken from  ./r/file
#             output overwrites    ./o/file
#
/etc/renice +19 $$
(echo -n 'Job run ' ; echo -n $argv[l] ; echo -n ' started :: ' ; date ) >! o/$argv[l]
(evm < r/$argv[l] >> o/$argv[l] ; \
   (echo -n 'Job run ' ; echo -n $argv[l] ; echo -n ' finished :: ' ;
                 date ) \ >> o/$argv[l] ; echo  run $argv[l] ...done ) &
```

<u>test input file :</u>

```
tv.
a('a/n').
a('a/adj').
a('a/det').
a('c/np').
statistics.
pc(ecsdk,npp,[begyndelsen]).
statistics.
pc(ecsdk,npp,[det,andet,bidrag]).
statistics.
```

<u>test output file :</u>

```
Job run test started :: Fri May 23 12:31:08 MET 1986
C-Prolog version 1.5
[  Restoring file /usr/local/lib/srevm ]

yes
| ?- verbose(_0) off
yes
| ?- a/n consulted 424800 bytes 138.883 sec.
yes
| ?- a/adj consulted 55652 bytes 17.6167 sec.
yes
| ?- a/det consulted 5268 bytes 1.70003 sec.
yes
| ?- c/np consulted 17904 bytes 4.7501 sec.
yes
 | ?- atom space: 256K (in use: 133524, max. used: 133524)
aux. stack: 4K (in use: 0, max. used: 192)
trail: 12K (in use: 48, max. used: 256)
heap: 1500K (in use: 633176, max. used: 954036)
global stack: 256K (in use: 0, max. used: 10052)
local stack: 128K (in use: 300, max. used: 1928)
```

```
Runtime:   701.28 sec.

yes
 | ?-
Goal is       the sentence          :       :       begyndelsen
              parsed from           :       :       npp
              translated through  :       :       [ ]

Level ecsdk : parsing found

[npp <np>,[[npp,{case=nge,def=df,gen=com,num=sg,type=no}],
    [np <n aggr2->,[[np,(case=nge,def=df,gen=com,num=sg,type=no}],
        [begyndelse,[[n,{case=nge,def=df,gen=com,num=sg,scat=no}] ,
                                                        terminal(begyndelsen)]]]]]]

no
 | ?- atom space: 256K (in use: 133524, max. used: 133524)
aux. stack: 4K (in use: 0, max. used: 192)
trail: 12K (in use: 48, max. used: 480)
heap: 1500K (in use: 633176, max. used: 954036)
global stack: 256K (in use: 0, max. used: 10052)
local stack: 128K (in use: 300, max. used: 3292)
Runtime:   916.63 sec.

yes
 | ?-
 Goal is                    the sentence          :   :   det andet bidrag
                            parsed from           :   :   npp
                            translated through :      :   [ ]

Level ecsdk : parsing found

[npp <detp cardp- aggr4- adjp* np srel->,[[npp,{case=nge,def=df,
                                                gen=neu,num=sg,type=no}] ,
    [detp <det>,[[detp,{case=nge,def=df,dem=yes,gen=neu,num=sg,type=no} ] ,
        [x,[[det,{case=nge,def=df,dem=yes,gen=neu,num=sg,scat=no}],
                                                            terminal(det)]]]]
    [adjp <coord- advp* adj aggr3->,[[adjp,{case=nge,def=df,deg=_1173,
                                                gen=neu,num=sg,type=no}],
        [anden,[[adj,{case=nge,def=df,deg=_1173,gen=neu,num=sg,scat=no}],
                                                            terminal(andet)]]]],
     [np <n aggr2->,[[np,{case=nge,def=idf,gen=neu,num=sg,type=no}],
         [bidrag,[[n,{case=nge,def=idf,gen=neu,num=sg,scat=no}],
                                                        terminal(bidrag)]]]]]]
no
| ?- atom space: 256K (in use: 133524, max. used: 133524)
aux. stack: 4K (in use: 0, max. used: 192)
trail: 12K (in use: 48, max. used: 768)
heap: 1500K (in use: 633176, max. used: 954036)
global stack: 256K (in use: 0, max. used: 13780)
local stack: 128K (in use: 300, max. used: 5220)
Runtime:  1386.07 sec.

yes
| ?-
[ Prolog execution halted ]
Job run test finished :: Fri May 23 13:38:39 MET 1986
```