0.0.1 Dynamic Symbols

With the new input modes, it is possible to modify a feature structure node using an arbitrary string, e.g., to store the surface form of a preprocessed token. To be able to do that in *cheap* , atomic types are created dynamically which may have no features and are direct subtypes of the topmost type. As a consequence, they only unify with themselves and `*top*`.

This functionality can also be used internally, say, to store arbitrary integer values in under a feature path.

At the moment, the dynamic symbols are removed when a new sentence is processed.

0.1 Input modes for *cheap*

0.1.1 YY-Mode

A token in YY input mode has the following form:

```
( <ID>, <STARTPOS>, <ENDPOS>,
  <PATHNR1> <PATHNR2> ... <PATHNRn>,
  "STEM" "SURFACE" ,
  <INFLECTION_POSITION>, "INFLR1" "INFLR2" ... "INFLRm",
  "POS1" <POS_PROB1> "POS2" <POS_PROB2> ... "POSk" <POS_PROBk> )
```

The characters "(", ")", and "," are part of the syntax, the tokens itself are separated by whitespace (Space, Tab).

`<ID>`, `<STARTPOS>`, `<ENDPOS>`, `<PATHNRi>` and `<INFLECTION_POSITION>` are *integer* values.

There are two ways of assigning numbers to the start and end positions, called the *positions* and the *counts* view.

In the positions view, you can imaging the words lying between increasingly numbered poles, so the start position of the first token is zero, its end position one, which is the starting position of the second token, and so on:

```
(3,0,1,1,"a",0,"null")(4,1,2,1,"good",0,"null")(5,2,3,1,"example",0,"null")
```

In this mode, start and end positions of one token may not be equal, because this would mean that the token has length zero. Tokens do not need to have length one, and there may also be gaps between the start and end position of adjacent tokens, which are automatically bridged unless there is another token spanning over the gap. But if the input encodes a word graph, it is important that all edges that are left-adjacent to some token have the same end position, because otherwise only the rightmost token will be connected to its neighbor.

The counts view is modeled after a situation were the component that produces the input labels the words with their character counts. The string "a good example" would therefore result in a list of tokens similar to the following (the gaps result from counting the white space characters, too):

```
(0,1,1,1,"a",0,"null")(1,3,6,1,"good",0,"null")(2,8,14,1,"example",0,"null")
```

```
(74, 12, 13,                (74, 12, 13, 1,             (93, 3, 6, 1,
 1,                          "Fische", 0,                "$NE_Loc" "New York",
 "fisch" "Fische", 0,        "null",                     0,
 "inflr-pl-nom-sg-dat",      "PROPER_NOUN" 0.91          "zero",
 "PROPER_NOUN" 0.91          "VERB_IMPERATIV" 0.09)      "PROPER_NOUN" 1.0)
 "VERB_IMPERATIV" 0.09)
```

fully analysed	do morphology and lexicon access	Named Entity with type name

Figure 1: Examples of YY input tokens

In this mode, the start position of the next right adjacent token always has to be bigger than the end position of its left neighbor, but, as in the first token, the start and end position of one token may be equal (which means the string has length one). Considering gaps and word graphs, the same remarks as above apply.

The POS tags are optional, if they are not given, the last comma (",") has to be omitted. The `<POS_PROBABILITYi>` values are C style (*double*) numbers.

The `<ID>` values have to be unique. `<STARTPOS>` and `<ENDPOS>` can get arbitrary integer values, as long as adjacency of the appropriate tokens is ensured. This input format is designed for word graphs, therefore, input tokens may overlap.

The `<PATHNRi>` values are use to encode a viable paths in the word graph, if this feature is not used, it is feasible to give it the same value for all tokens, say 1.

Tokens, whose `STEM` consists only of characters that are specified as `punctuation-characters` in the `.set` file are skipped by the parser.

The `"SURFACE"` string is optional. If this string is omitted, the double quotes should also be omitted.

There are special values for `"INFLR1"` that also imply $m = 1$:

`"zero"` No inflection

`"null"` Do the morphological analysis internally. In this case, the string in `"STEM"` is analysed, therefore, this string has to be the surface form. `"SURFACE"` may and should be omitted in this case.

To specify named entities or other pre-analyzed tokens, `"STEM"` should contain the HPSG type name instead, which has to start with the character specified as CLASS-NAME-CHAR in the settings, or, if this is not set, with '$'.

0.1.2 XML input mode

XML input mode is very similar to YY input mode. It allows you to specify only simple tokens that get analysed internally by *cheap* or to put all kinds of preprocessing information *cheap* can handle into the input directly, namely POS, morphology, lexicon lookup and multi-component entries.

It extends the YY mode in that it allows to have structured input tokens to provide a means to encode, say, named entities resulting from base tokens. It also allows to specify modifications to feature structures (coming from lexicon entries.

The example in figure 2 illustrates most of the available features. Tokens `W0` and `W1` are not analysed at all by *cheap* because the (boolean) `constant` attribute is `yes`. The default value of this attribute is `no`, which means that the token `W3` will be analysed by all of the activated preprocessing modules in *cheap* .

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE pet-input-chart
  SYSTEM "/home/cl-home/kiefer/src/pet/doc/pic.dtd">
<pet-input-chart>
  <w id="W0" cstart="1" cend="3" constant="yes">
    <surface>Kim</surface>
  </w>
  <w id="W1" cstart="5" cend="9" constant="yes">
    <surface>Novak</surface>
  </w>
  <ne id="NE0" prio="1.0">
    <ref id="W0">
    <ref id="W1">
    <pos tag="PN" prio="1.0">
    <typeinfo id="TNE0" baseform="no">
      <stem>$generic_name</stem>
      <fsmod path="SYNSEM.LOCAL.HEAD.FORM" value="Kim Novak"/>
    </typeinfo>
  </ne>
  <w id="W2" cstart="11" cend="16" constant="yes">
    <surface>sleeps</surface>
    <pos tag="VVFIN" prio="7.80000e-1"/>
    <pos tag="NN" prio="2.30000e-2"/>
    <typeinfo id="W1A1">
      <stem>sleep</stem>
      <infl name="$third_sg_fin_verb_infl_rule"/>
    </typeinfo>
    <typeinfo id="W1A2">
      <stem>sleep</stem>
      <infl name="$plur_noun_infl_rule"/>
    </typeinfo>
  </w>
  <w id="W3" cstart="18" cend="22">
    <surface>badly</surface>
    <pos tag="ADV" prio="1.00000e+1"/>
  </w>
</pet-input-chart>
```

Figure 2: Example of an XML input chart

Token `NE0` is an example of a complex token referencing a sequence of two base tokens. Its `typeinfo` directly gives the HPSG type name whose feature structure should be used as lexical item in *cheap* . While in YY mode this was triggered by a leading special character, in XML the attribute `baseform` decides if the string enclosed by the `<stem>` tag is to be interpreted as lexical base form or as type name. The default value of `baseform` is `yes`. In this token, the surface string is unified into the feature structure under path `SYNSEM.LOCAL.HEAD.FORM`, which is specified with the `<fsmod>` tag. The value of an `<fsmod>` may be an arbitrary string. *cheap* will add a dynamic symbol if the string is not a known type or symbol name.

Every `<typeinfo>` tag potentially generates a lexical item (if it leads to a valid lexical feature structure). Thus, there will be two readings for the token `W2` ("sleeps"), whereas internal analysis of the surface form has been inhibited. This need not be necessarily so. It is possible to provide external analyses and have a `<w>` token also being analysed internally if the `constant` flag is omitted or set to `no`.

The XML tag `<surface>` encloses the surface string, `<pos>` and `<path>` tags are analogous to YY mode; multiple `<infl>` rules in a `<typeinfo>` will have to be considered from first to last.

XML input mode can be used in two different ways, either by specifying a file name containing the XML data (preferably with correct XML header and DTD or DTD URL specification) or by giving the XML data directly.

If the XML data is put directly into the standard input, it *must* start with a valid XML header `<?xml version="1.0" ... ?>` with no leading whitespace, because recognition of the header triggers the reading of XML from standard input. The end of the data is marked by an empty line (two consecutive newline characters), therefore, the data itself, including an eventually given DTD, may not contain empty lines.

0.2 Settings And Options

0.2.1 Use of generic lexicon entries

If an input token did not generate a chart edge that is useful for syntax parsing, it normally triggers an error of the form "no lexicon entry for ...". If the DEFAULT-LES option was set, generic entries, selected on the basis of POS information, are tried to avoid this. The way they are selected is defined by the POSMAPPING setting.

If the input word has one more more POS tags associated to it, these are looked up in the POSMAPPING table: this table is a list of pairs (*tag*, *gle*) where *gle* is the name of a type or instance with a status that is in the GENERIC-LEXENTRY-STATUS-VALUES list. A non-empty POSMAPPING table will filter all generic entries that are not explicitly licensed by a POS tag.[1]

0.2.2 Packing and Restrictor

The restrictor is specified by giving a list of features or paths (features are just paths of length one). The structure at the end of a specified path will be replaced by a maximally underspecified appropriate feature structure. The paths list should not contain two paths where one is a sub-path of the other. Although this does not result in a program error, it may give wrong or unwanted results, and it does not seem to be a meaningful restrictor specification anyway.

The restrictor is only applied to rules and lexicon entries when the grammar is read in. It is not applied during parsing for efficiency reasons. If paths of length more than one are used, this may result in feature structures that contain paths which should in principle be deleted. This is due to the fact that these paths can be constructed using other, non-restricted sub-paths and coreferences.

[1]This paragraph is partly taken from the ERG settings file.

The PACKING option gets a bit-masked argument (add the appropriate value to set the bit):

bit 0 (= 1) equi packing: Only items with equivalent feature structures will be packed, which should make unification in the unpacking step unnecessary

bit 1 (= 2) proactive packing:

bit 2 (= 4) retroactive packing:

bit 7 (= 128) unpacking of results is disabled

For additional information about packing, see [**?**].

0.2.3 Quick Check Path Computation

There are three options to compute quick check paths: COMPUTE-QC computes quick check paths for unification and subsumption, if packing is enabled, while COMPUTE-QC-UNIF and COMPUTE-QC-SUBS only compute the paths for the appropriate operation. All three take an optional argument to determine the file containing the result. The default path for that file is `/tmp/qc.tdl`.

0.2.4 Tokenizer Selection

The tokenizer used in cheap is selected with the option TOK, which requires the tokenizer name as argument. Possible values (in version 0.99.5) are:

string the plain old string tokenizer, mainly tuned to English

yy YY input mode, see section 0.1.1

yy_counts YY input mode, using counts, see section 0.1.1

xml XML input mode, see section 0.1.2

xml_counts XML input mode, using counts, see section 0.1.2

If a wrong or no name is given, the string tokenizer is activated by default.

0.2.5 Result Selection

If no complete parse of the whole input could be found, *cheap* will normally produce no MRS output. With the option PARTIAL, *cheap* attempts to select a set of "best" partial results that cover the input as good as possible and produce MRS output for them.

The option RESULT gets a (required) integer argument that specifies the maximal number of results that will be produced if a full parse could be found.

0.2.6 Result Dumps

There are two file dump formats supported by cheap.

The first dumps the whole chart with feature structures after each parse in a form that was formerly used for input to Java programs, and which is activated with the option JXCHGDUMP. This option gets as required argument a directory where, for every input sentence, a file is created that contains the chart dump. The file name corresponds to the input sequence, with spaces replaced by underscores.

The second dumps `incr(tsdb[])` databases and is useful if external preprocessors are used together with either YY or XML input mode because `incr(tsdb[])` server mode only works from strings with internal preprocessors. The input sequence is reconstructed from the output of the tokenizer (using a shortest path algorithm on the input items), and may therefore be not exactly the same as those given to the preprocessor.

FAQ: Q: What is counted as words in PET? A: The base form entries (also completely filled multiwords) before inflection rules have been applied.

Appendix A

DTD of XML input mode

```
 <!DOCTYPE pet-input-chart [ <!ELEMENT
pet-input-chart ( w | ne )* >

  <!-- base input token -->
  <!ELEMENT w ( surface, path*, pos*, typeinfo* ) >
  <!ATTLIST w         id ID      #REQUIRED
                  cstart NMTOKEN #REQUIRED
                    cend NMTOKEN #REQUIRED
                    prio CDATA   #IMPLIED
                constant (yes | no) "no" >
  <!-- constant "yes" means: do not analyse, i.e., if the tag contains
       no typeinfo, no lexical item will be build by the token -->

  <!-- The surface string -->
  <!ELEMENT surface ( #PCDATA ) >

  <!-- numbers that encode valid paths through the input graph (optional) -->
  <!ELEMENT path EMPTY >
  <!ATTLIST path     num NMTOKEN #REQUIRED >

  <!-- every typeinfo generates a lexical token -->
  <!ELEMENT typeinfo ( stem, infl*, fsmod* ) >
  <!ATTLIST typeinfo   id ID     #REQUIRED
                     prio CDATA  #IMPLIED
                 baseform (yes | no) "yes" >
  <!-- Baseform yes: lexical base form; no: type name -->

  <!-- lexical base form or type name -->
  <!ELEMENT stem ( #PCDATA ) >

  <!-- type name of an inflection rule-->
  <!ELEMENT infl  EMPTY >
  <!ATTLIST infl    name CDATA   #REQUIRED >

  <!-- put type value under path into the lexical feature structure -->
  <!ELEMENT fsmod  EMPTY >
  <!ATTLIST fsmod   path CDATA   #REQUIRED
                   value CDATA   #REQUIRED >
  <!-- part-of-speech tags with priorities -->
  <!ELEMENT pos  EMPTY >
  <!ATTLIST pos      tag CDATA   #REQUIRED
                    prio CDATA   #IMPLIED >

  <!-- structured input items, mostly to encode named entities -->
  <!ELEMENT ne  ( ref+, pos*, typeinfo+ )  >
  <!ATTLIST ne        id ID      #REQUIRED
                    prio CDATA   #IMPLIED >

  <!-- reference to a base token -->
  <!ELEMENT ref  EMPTY >
  <!ATTLIST ref      dtr IDREF   #REQUIRED >
]>
```