Technical Report no. 2007-10

The Extract Tool

Markus Forsberg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2007

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2007-10
ISSN 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Göteborg, Sweden, 2007

1 Introduction

This technical report describes *Extract v2.0*, a tool for extracting linguistic information from raw text data, in particular inflectional information on words, based on the word forms appearing in the text data. This document is a revision of the *Extract manual* describing *Extract v1.0*, a previous version of the tool [1]. The main differences between v1.0 and v2.0 are the addition of Constraint Grammar together with a new data format, and an upgrade of the regular expression engine. The new engine has changed the semantics of the regular expressions: `letter`, `upper`, and `lower` no longer refer to Unicode letters, but to English letters. However, this small loss of functionality is well compensated by the improved efficiency.

The input of Extract is a file containing a, possibly unannotated, corpus and a file containing Extract rules. Each rule provides a *search template* for some linguistic information, such as regular nouns of English. If a rule's search template is applicable to a set of word forms in the text data then the tool outputs the *head* of the rule. The head of the rule specifies the rule identifier and the output word forms.

The tool's output is a list of analyses, each analysis consists of a sequence of words annotated with an identifier. The identifier states some linguistic information, e.g. `regNoun hat` may encode that the word `hat` is a regular noun, and `v2 eat` may encode that `eat` is a transitive verb.

The previous version of the tool viewed the input data as a set of words with no contextual information. The focus was solely on annotating words with paradigm identifiers, hence the rules were marked with the keyword `paradigm`. The new version of Extract, presented in this document, allows contextual information in the rules, expressed with a variant of the *Constraint Grammar* formalism [3]. This addition allows more involved linguistic information to be extracted, such as subcategorization frames for verbs. This change is reflected through the change of the keyword `paradigm` to the more generic keyword `rule`. The old keyword is still usable for backward compatibility reasons.

2 Lexicon Extraction

We start our discussion with *lexicon extraction* — how *citation forms* marked with *inflectional information* can be extracted from a corpus. We will present our tool incrementally, trying to motivate its different features by presenting problems that need to be resolved.

Citation forms, or dictionary forms, are those word forms typically found in a normal dictionary. They represent a word, or a word's inflection table, and are usually the word forms that are the most unmarked, i.e. perceived as most neutral, or the most characteristic.

Paradigms are abstractions from inflection tables. A paradigm identifier together with the citation forms is enough to produce the complete inflection table of a word.

The objective is to extract citation forms annotated with identifiers.

A first approach is to search for all word forms of a paradigm, where the stem is replaced with a variable. We then traverse the input data searching for instantiations of the variable. This idea is illustrated in the syntax of Extract with the paradigm of Swedish first declension noun as example.

```
rule decl1 =
 x+"a"
 { x+"a"  & x+"as"  & x+"an"   & x+"ans" &
   x+"or" & x+"ors" & x+"orna" & x+"ornas" } ;
```

The rules of Extract consists of an *identifier*, a *body* and a *head*. The body consists of a search template inside curly brackets. Given that all word forms are found in the search template for some string `x`, then the head `x+"a"` will be output tagged with the identifier `decl1`.

Stated more concretely, if we have the inflection table of the Swedish word *smula* (Eng. 'crumb') in our input data, that is, the word forms *smula*, *smulas*, *smulan*, *smulans*, *smulor*, *smulors*, *smulorna* and *smulornas*, then the output would be `decl1 smula`.

If the rule `decl1` is defined in a file `rules`, and if we collect some Swedish text and put it in a file `Swedish_text`, then Extract is runnable with those files as arguments (see Fig. 1). Two additional flags are supplied, `-u`, for no duplicates, and `-utf8`, for UTF-8 encoding. The text we are using in this example is the complete set of word forms for the words *smula* (Eng. 'crumb'), *mnniska* (Eng. 'human') and *flicka* (Eng. 'girl'), all first declension nouns.

The result of the run is, as expected, the three words in the input data.

This looks straightforward enough, but unfortunately, words in a language with non-trivial morphology, such as Swedish, rarely occur in all word forms. Furthermore, it is often difficult to select a subset of word forms that are the most representative for a particular paradigm, if at all possible. For example, in the rule `decl1`, it does not matter if a word form is in the nominative or genitive case, since the case inflection is the same for all declensions. But, restricting the search to just one of the cases would

```
$ extract -u -utf8 rules Swedish_text

*****************************************
*           Lexicon Extraction          *
*                 with                  *
*           Constraint Grammar          *
*****************************************
* (c) Markus Forsberg & Aarne Ranta 2007 *
* under GNU General Public License.      *
*****************************************

1 rule read from 'rules'.

Reading raw text data from 'Swedish_text'...

decl1 flicka
decl1 mnniska
decl1 smula

Unique tokens     : 24
Corpus Usage      : 100.00%
Words Extracted   : 3
```

Figure 1: Example run

$$
\begin{array}{lcl}
\langle \mathit{Logic} \rangle & ::= & \langle \mathit{Logic} \rangle \ \texttt{\&} \ \langle \mathit{Logic} \rangle \\
& | & \langle \mathit{Logic} \rangle \ | \ \langle \mathit{Logic} \rangle \\
& | & \langle \mathit{Logic} \rangle \\
& | & \texttt{\~{}} \ \langle \mathit{Logic} \rangle \\
& | & \langle \mathit{Pattern} \rangle \\
& | & \texttt{(} \ \langle \mathit{Logic} \rangle \ \texttt{)}
\end{array}
$$

Figure 2: Propositional logic

be completely arbitrary. The tool supports propositional logic in the constraints to allow more fine-grained descriptions, such as using disjunction for the case distinction. The use of propositional logic in Extract is described in more detail in Sec. 2.1.

In the rule `decl1`, there is no control over which substrings may be associated to the variable `x`. We may, for example, want to state that `x` should at least be monosyllabic to avoid spurious outputs. Sec. 2.2 describes how the tool improves this situation by allowing variables to be associated with a regular expression.

The stem in rule `decl1` is the same for all word forms. This is not the case for many paradigms, such as paradigms with umlaut. In Sec. 2.3 we describes how such paradigms can be defined by the use of multiple variables.

2.1 Propositional Logic

Propositional logic is used in the body of a rule to enable a more fine-grained description of which word forms the tool should look for. The basic unit is a *pattern*, corresponding to a word form, which are the atoms of a propositional logic formula. The formula is referred to as a *search template*. A formula is built with three connectives with their conventional meanings: *conjunction* (`&`), *disjunction* (`|`) and *negation* (`~`). The syntax is given in Fig. 2, without precedence and associativity information. For a complete reference, see Sec. 10.

Note that negation in a search template is true when no counter-example is found, i.e. it is *negation as failure*. We try to prove p, and when we fail, we conclude `~`p. The use of negation in lexicon extraction makes it *non-monotonic* — the addition of more data may lead to a smaller result set.

The rule in Sec. 2 can be rewritten with propositional logic to reflect that it is sufficient to find a word form either in the nominative or genitive case by the use of the disjunctive operator.

```
rule decl1 =
      x+"a"
      {( x+"a"  | x+"an"  | x+"or"  | x+"orna") &
       ( x+"as" | x+"ans" | x+"ors" | x+"ornas")} ;
```

The word forms in the head are not necessary in the input data, they may be constructed from the instantiated variables and constant strings.

2.2 Regular Expressions

In Sec. 2 we mentioned that it is necessary to increase the control over what substrings a rule's variables may be associated with — allowing a variable to be associated to any substring may seriously degrade the performance of the tool. For example, in Swedish it is necessary to require that the variable corresponding to the stem contains, at least, one vowel to avoid many false positives.

The solution Extract provides is to enable variables to be associated to regular expressions describing which strings the variable can match. An unannotated variable matches any string, i.e. its regular expression is Kleene star over all characters.

Let us define the stems of German nouns, exemplified with the rule `noun`. A German noun begins with an uppercase letter, which we express easily with a regular expression. In this example, it is necessary that the word forms in the rule are capitalized to avoid capturing verbs.

```
regexp GermanUpper = "" | "" | "" | "" | upper ;
regexp GermanLower = "" | "" | "" | lower ;
regexp UpperWord   = GermanUpper GermanLower*;

rule noun [x:UpperWord] =
          { x+"e" & x+"en" } ;
```

The syntax of the tool's regular expression is given in Fig. 3, with the normal connectives: union, concatenation, set minus, Kleene's star, Kleene's plus and optionality. *eps* refers to the empty string, *digit* to $0-9$, *letter* to an alphabetic character (a-z, A-Z), *lower* and *upper* to lowercase (a-z) and uppercase (A-Z), and *char* to any character. A regular expression can also contain a double quoted string, which is interpreted as the concatenation of the characters in the string.

⟨*Reg* ⟩ ::= ⟨*Reg* ⟩ | ⟨*Reg* ⟩
| ⟨*Reg* ⟩ − ⟨*Reg* ⟩
| ⟨*Reg* ⟩ ⟨*Reg* ⟩
| ⟨*Reg* ⟩ `*`
| ⟨*Reg* ⟩ +
| ⟨*Reg* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| ⟨*String* ⟩
| `(` ⟨*Reg* ⟩ `)`

Figure 3: Regular expression syntax

2.3 Multiple Variables

Not all paradigms are as neat as the initial example — phenomena like *umlaut* require increased control over the variable part. The solution the tool provides is to allow multiple variables, i.e. that a pattern can have more than one variable. This is best explained with an example, here with the rules of two German nouns.

```
regexp GermanUpper = "" | "" | "" | "" | upper ;
regexp GermanLower = "" | "" | "" | lower ;
regexp Pre         = GermanUpper GermanLower ;
regexp Whatever    = GermanLower* ;

rule n2 [F:Pre, ll:Whatever] =
     F+"a"+ll
    {F+"a"+ll & F+""+ll+"e"} ;

rule n3 [W:Pre, rt:Whatever] =
     W+"o"+rt
    {W+"o"+rt & W+""+rt+"er"} ;
```

The use of multiple variables did previously reduce the performance of the tool. This is no longer true in Extract v2.0, as long as no back reference is used, due to a new regular expression back end.

It is not required that all variables occur in every pattern, but the tool performs an initial match only on patterns that contains all variables. The reason for this is efficiency — the tool only considers one word at a time, and if the word matches one of the patterns, it searches for all other patterns with the variables instantiated by the initial match. For obvious reasons, an initial match is never performed under a negation (this would imply that the tool searches for something it does not want to find!).

2.4 Multiple Output Patterns

The head of a rule may have multiple output patterns, which support more abstract rules. An example is Swedish nouns, where many nouns can be correctly classified by just analyzing the word forms in nominative singular and nominative plural. The first and second declensions are handled with the same paradigm function, and the head consists of two output patterns. The constraints are omitted.

```
rule regNoun =
        flick+"a" flick+"or"
        {...} ;

rule regNoun =
        pojk+"e" pojk+"ar"
        {...} ;
```

3 Structured Input Data

The input data of Extract v2.0 can either be raw text data or structured data. Structured data consists of a list of sequences of tokens, *chunks*, where chunks correspond to a meaningful context. A token is a word form associated with a list of *terms*, referred to as the *ambiguity class*. A term is a non-atomic structured information unit, consisting of labels and applications of labels. Here is an example of two chunks: the first one consisting of two unambiguous tokens, and the second one of four tokens, two unambiguous tokens, one ambiguous (`sticka`), and one token without analysis (`fingret`).

```
{ ("hej",in)
  ("!",spec) }
{ ("sticka",nn sg indef nom u|vb inf aktiv|vb imper)
  ("i",pr)
  ("fingret",)
  (".",spec) }
```

A chunk may be a sentence, a phrase or some other appropriate unit. Raw text data is also divided into chunks, where the dividers are major punctuations. Although a naive approach, it works quite well, since the garbage generated by, e.g. abbreviations, does not normally give rise to any spurious analyses.

A chunk is used to generate the context of a token, which can be referred to with Constraint Grammar (CG) constructs, explained Sec. 4. If no CG constructs appear in any rule, then the chunks are left unused, and the tool acts in the same manner as before.

A morphological lexicon typically includes more information about the word forms than just part of speech, e.g. inflectional parameters, such as number or case, and inherent parameters, such as gender. Because of this, it is natural to allow non-atomic class labels that can be partially specified. This is done by a simple term language. When we refer to non-atomic class labels, we can use a *don't care* symbol _ to state that a part of a class label is irrelevant.

If a lexical resource in FM is available, then structured input data can be produced with the following command. We use the FM implementation of Latin in this example to produce our structured data.

```
$ cat raw_text_data.txt | ./morpho_lat -pos > structured_data.txt
```

This produces a file `structured_data.txt`, consisting of a sequence of chunks, where the tokens have been annotated with the analyses provided by the FM implementation.

4 Constraint Grammar

4.1 Introduction to Constraint Grammar

Constraint grammar (CG) [3] [4] was first introduced by F. Karlsson, presented as a facility for performing disambiguation and light parsing. Karlsson's Constraint Grammar framework assumes *total knowledge*, i.e. that all possible analyses of a word is already known, and the objective of a Constraint Grammar is to reduce ambiguity. Yet another assumption, related to total knowledge, is the *Sherlock Holmes assumption*, stating that if we remove all possible analyses except one, then that one must be the correct one, no matter how improbable.

The input word forms of a CG have an *ambiguity class* associated to them. The ambiguity classes consist of lists of readings, and the goal of the rules of a CG is to reduce these ambiguity classes.

A CG rule consists of a *domain*, a *target*, an *operator* and a *context*.

The *domain* provides a typing of the rule, declaring which of the tokens that are affected. For example, `@w` declares that all tokens are affected. The *target* declares a result of the rule, its reading, and together with the *operator* of the rule, defines the outcome of the rule if the input word form satisfies the rule's *context*.

Two examples are taken from F. Karlsson [3]. The first rule states that for any word (the domain `@w`) preceded by a word with the reading `TO` (the context), then the target `VFIN` is discarded (stated by the operator `=0`). The second rule contains a operator `=!`, stating that the target reading `"<REL>"` is unambiguous, if the input word form `that` (the domain), if it is preceded by `NOMHEAD` and followed by `VFIN`.

```
1. (@w=0 VFIN (-1 TO))

2. ("that" =! "<REL>" (-1 NOMHEAD) (1 VFIN))
```

4.2 Constraint Grammar in Extract Rules

The total knowledge assumption is a reasonable assumption with a large dictionary, but in a lexicon extraction setting it is invalid, since it is the unknown word forms that are interesting. Theoretically, we could fix this by associating all words with all possible analyses, but this would destroy all information provided by our dictionary.

Constraint Grammar in Extract differs from the traditional definition, since the task is no longer to reduce ambiguity, but to classify unknown words.

There is only one, implicit, operator used in the rules, which is not quite the same as any of the operators in CG, since it selects a reading that, possibly, is not part of the ambiguity class. This is natural since the words that we are aiming for are outside the lexicon.

The constraints in the rules are defined with a propositional formula, which already existed in Karlsson's framework in an indirect fashion. Conjunction is implicit in the rules, what he refers to as 'polarity' corresponds to negation, and disjunction can be defined by translating it to a list of rules.

A *context element* is a triple, consisting of a position, a regular expression and a term, as illustrated in Fig. 4. Karlsson's CG lacked the possibility to refer to words in the context with regular expressions, which is natural since the assumption is that the ambiguity class contains the correct class, so we normally only need to refer to the classes. Here, we need to be able

(⟨*Position* ⟩ , ⟨*Reg* ⟩ , ⟨*Unique* ⟩ ⟨*Patt* ⟩)

Figure 4: An atom of a constraint

to refer to the actual words, since we cannot assume that the correct class is included.

An example is given below where we are interested in identifying nouns. The constraint is in square brackets and states that the word in question must be preceded by an article and followed by a verb in present tense. We use the don't care symbol _ for positions that we are not interested in. We also use the uniqueness operator `!` to express that we are only interested in words where the preceding and following words is unambiguous.

```
rule noun x [x:Word] =
  x
  {
   x [(-1,_, ! article _ _) &
      (1, _, ! verb present _ _)]
  }
```

Wild cards (_) can be used anywhere except for positions. A wild card can be matched with anything.

4.3 Positions in CG

Positions in a context element of CG are either *absolute*, *unbounded*, *relative* or *unbounded relative*. A position identifies a token in the context of the current token.

Positions are referred to with integers, where the current token is at position 0, the ones to the left have increasingly negative numbers, and the ones to the right have increasingly positive numbers. An absolute position is an integer referring to a token, much in the same way as indexing an array.

Unbounded positions, marked by a star '*', refer to the first hit starting at an absolute position within the current chunk. An unbounded position may furthermore be labelled so the resulting position can be referred to by a relative position. As an example, consider the following two context elements. The first element contains an unbounded position, where the label `t` will be assigned the value of the position of the first occurrence of `the`, if any, starting to the left of the current token. The second element contains a relative position, stating that the word to the right of the first `the` should be `black`.

```
(t@-1*,"the",_) & (t+1,"black",_)
```

Relative positions may also be unbounded. We could change our example of a relative position to `t+1*` to express that the word `black` should be somewhere on the right of `the`, and moreover, we could label it so its position becomes referable: `p@t+1*`.

4.4 Changes in the Algorithm

Here is a modified version of the algorithm (without CG) given in Forsberg et al. [1], in pseudo-code notation:

```
let L be the empty lexicon.
let R be the set of extraction rules.
let W be all word types in the corpus.
let S be the list of chunks.
for each w : W
 for each r : R
  for each constraint r(C) with which w matches r(C)
   if W,S satisfies instantiated r(C) with the result H,
    add H to L
   endif
  end
 end
end
```

The algorithm is initialized by reading the word types of the corpus into an array W. A word w **matches** a constraint of a rule $r(C)$, if it can match any of the patterns in the rule's constraint that contains all variables occurring in the constraint. The result of a successful match is an **instantiated constraint**, i.e. a logical formula with words as atomic propositions. The corpus W **satisfies** an instantiated constraint $r(C)$ if the formula is true, where the truth of an atomic proposition a means that the word a occurs in W. And finally, the store S, containing the contextual information, **satisfies** an instantiated constraint $r(C)$ if, for a Constraint Grammar cg_i associated to an instantiated pattern $a \in r(C)$, there exists at least one chunk $ch \in S$, in which a appears, which **satisfies** cg_i.

5 The Implementation

An important consideration with the implementation is to be able to deal with as much data as possible. It is space, not speed, which is the critical

issue (within reason, of course), since the result typically improves with larger amounts of data.

The representation of strings in Haskell is a bit wasteful, and if one is not careful, one soon runs out of memory. There are other string types with more efficient representations, but these types are not as convenient to program with. We solved this by implementing *sharing*, i.e. a word form is only fully represented once in memory, and all other occurrences are pointers to the full representation. This solution gives the tool a reasonable space behavior.

6 Experiments

An experiment with the French Hansard corpus has been conducted and furthermore described in Extract's FinTAL article [1].

Some initial experiments have been tried using CG and structured data while extracting Swedish words, where the starting point is a morphology implemented in Functional Morphology (FM) [2]. FM supports the analysis of a text into Extract's data format, where the chunks are divided, in a naive fashion, at major punctuations. No disambiguation is performed, i.e. FM augments all word forms with all possible interpretations.

In these experiments we realized that without any form of disambiguation, the information provided by the existing lexicon is of little use, at least for Swedish. It is not necessary to perform full disambiguation: it is sufficient to focus on classes mentioned in the constraints. For example, consider the word *att* in Swedish, which is a function word that is either an infinitive marker or a subjunction (*that*). If we knew that *att* is an infinitive marker, it would be a good indicator for identifying verbs, but with no disambiguation, we are not helped.

It is our experience that it is more difficult than we previously thought to use the context consisting of either unannotated word forms or word forms annotated with ambiguous information for improving the extraction in a substantial way. Our hypothesis is that to be able to use CG in Extract efficiently, we need to augment the word forms with unambiguous information. Ambiguous information is useful for filtering out spurious outputs, but difficult to use in a positive sense, i.e. to locate and extract new words. However, it is too early to draw any conclusions.

7 Compiling Extract

The source code is downloadable at Extract's homepage[1].

Extract requires the GHC compiler[2] to be built. Since Extract is a command-line program, it should work on all platforms supported by the GHC compiler.

1. `tar xvfz Extract_2.0.tgz`
2. `cd Extract_2.0`
3. `make`
4. This produces a binary: `extract`

8 Running Extract

Extract is command-line program, which is run with a *rule file*, a *text file*, and possibly, some flags.

```
$ extract [Flag(s)] rule_file  text_file
```

The text file `text_file` may either be structured, in the sense described in Sec. 3, or unformatted. Extract is able to automatically detect which of the different types it is.

If a lexical resource in Functional Morphology is available, then it can be used to generate structured data. See the technical report of Functional Morphology for details.

9 Command-line Options

All command-line flags available in Extract are printed by calling Extract with the flag `-h`. The output is given in Fig. 5. Most of the flags are self-explanatory, but we will still give some clarifying comments for all of them.

[1] `http://www.cs.chalmers.se/~markus/Extract`
[2] `http://www.haskell.org/ghc`

```
$ extract -h

*****************************************
*           Lexicon Extraction          *
*                 with                  *
*           Constraint Grammar          *
*****************************************
* (c) Markus Forsberg & Aarne Ranta 2007 *
* under GNU General Public License.      *
*****************************************

 Help message:
 extract [Option(s)] rule_file  corpus_file
 Options:
  -h       Display this message
  -utf8    Use UTF-8 encoding
  -nobad   Keep only the analysed words
  -uncap   Transform uppercase to lowercase
  -nocap   Remove all words in uppercase
  -e       Print evidence as comment
  -u       Print no duplicates
  -id      Print identifier
  -r       Reverse words
```

Figure 5: Help command

9.1 Character Encodings

Extract v1.0 supported the character encoding ISO-8859, which restricts the use of the tool to a particular set of language, or forces the use of some ad-hoc encoding. Extract v2.0 acknowledges this problem by introducing UTF-8 encoding, activated with the flag `-utf8`. If the UTF-8 mode is activated, then both the input data file and the rule file must be in that encoding.

9.2 Data Preprocessing

There are two commands for data preprocessing: (`-uncap`), which transforms the word forms to lowercase and `-nocap` which removes all capitalized words.

9.3 Output Control

The word forms that do not give rise to any output are by default printed with dashes (`--`) in front of them. To avoid printing these, use the flag `-nobad`.

Extract traverses all word forms and tries to instantiate the rule's constraint. This, in turn, typically means duplicated output, since every word form gives rise to its own instantiations. This is avoided by giving the uniqueness flag, `-u`, which means that a history of previous results is kept and no duplicates are output. Extract is typically run with the uniqueness flag, or, if preferred, with the Unix command `sort -u`.

A constraint of a rule is fulfilled by a set of word forms, but since a constraint may contain disjunctive patterns, it is not always clear which word forms were used. Since this information may be useful, it is possible to obtain it with the evidence flag `-e`. Every output will then be marked with the word forms used.

Multiple rules may have the same identifier, and it is sometimes useful to know exactly which of the rules were used: if for nothing else, so for rule debugging purposes. The identify flag `-id` annotates the output with an integer corresponding to a rule in the rule file.

9.4 Dictionary

One of the data structures of Extract is a trie, which is a deterministic automaton, i.e. it is prefix-minimal but not suffix-minimal. If a language is suffix-heavy, it may save memory to reverse the string, which can be done with the flag `-r`.

10 BNFC Documentation of Extract

10.1 The Language Extract

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Extract

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String* ⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals ⟨*Int* ⟩ are nonempty sequences of digits.

Character literals ⟨*Char* ⟩ have the form `'`c`'`, where c is any single character.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Extract are the following:

```
char      context  digit
eps       letter   lower
paradigm  regexp   rule
upper
```

The symbols used in Extract are the following:

```
;   =   {
}   [   ]
:   ,   -
(   )   +
&   |   ~
!   @   −
*   ?
```

Comments

Single-line comments begin with $--$.
Multiple-line comments are enclosed with $\{-$ and $-\}$.

The syntactic structure of Extract

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Grammar* ⟩ ::= ⟨*ListDef* ⟩

⟨*ListDef* ⟩ ::= ϵ
| ⟨*Def* ⟩ `;` ⟨*ListDef* ⟩

⟨*Def* ⟩ ::= `paradigm` ⟨*Ident* ⟩ ⟨*Env* ⟩ = ⟨*Head* ⟩ { ⟨*Logic* ⟩ }
| `rule` ⟨*Ident* ⟩ ⟨*Env* ⟩ = ⟨*Head* ⟩ { ⟨*Logic* ⟩ }
| `regexp` ⟨*Ident* ⟩ = ⟨*Reg* ⟩
| `context` ⟨*Ident* ⟩ = ⟨*CLogic* ⟩

⟨*Env* ⟩ ::= `[` ⟨*ListBinding* ⟩ `]`
| ϵ

⟨*Binding* ⟩ ::= ⟨*Ident* ⟩ `:` ⟨*Reg* ⟩

⟨*ListBinding* ⟩ ::= ϵ
| ⟨*Binding* ⟩
| ⟨*Binding* ⟩ `,` ⟨*ListBinding* ⟩

⟨*Pattern* ⟩ ::= ⟨*ListItem* ⟩ ⟨*Constraint* ⟩

⟨*Item* ⟩ ::= ⟨*String* ⟩
| ⟨*Ident* ⟩

⟨*Patt1* ⟩ ::= _
| ⟨*Ident* ⟩
| (⟨*Patt* ⟩)

⟨*Patt* ⟩ ::= ⟨*Ident* ⟩ ⟨*ListPatt1* ⟩
| ⟨*Patt1* ⟩

⟨*ListPatt1* ⟩ ::= ⟨*Patt1* ⟩
| ⟨*Patt1* ⟩ ⟨*ListPatt1* ⟩

⟨*Head* ⟩ ::= ⟨*ListPattern* ⟩

⟨*ListPattern* ⟩ ::= ⟨*Pattern* ⟩
| ⟨*Pattern* ⟩ ⟨*ListPattern* ⟩

⟨*ListItem* ⟩ ::= ⟨*Item* ⟩
| ⟨*Item* ⟩ + ⟨*ListItem* ⟩

⟨*Constraint* ⟩ ::= [⟨*CLogic* ⟩]
| [⟨*Ident* ⟩]
| ϵ

⟨*CLogic* ⟩ ::= ⟨*CLogic* ⟩ & ⟨*CLogic1* ⟩
| ⟨*CLogic* ⟩ | ⟨*CLogic1* ⟩
| ⟨*CLogic1* ⟩

⟨*CLogic1* ⟩ ::= ~ ⟨*CLogic1* ⟩
| _
| (⟨*Position* ⟩ , ⟨*Reg* ⟩ , ⟨*Unique* ⟩ ⟨*Patt* ⟩)
| (⟨*CLogic* ⟩)

⟨*Unique* ⟩ ::= !
| ϵ

⟨*Position* ⟩ ::= _
| ⟨*Pos* ⟩
| ⟨*Ident* ⟩ @ ⟨*Pos* ⟩
| ⟨*Ident* ⟩ @ ⟨*Ident* ⟩ ⟨*Pos* ⟩
| ⟨*Ident* ⟩ ⟨*Pos* ⟩

⟨*Pos* ⟩ ::= ⟨*Integer* ⟩
| + ⟨*Integer* ⟩
| − ⟨*Integer* ⟩
| + ⟨*Integer* ⟩ *
| − ⟨*Integer* ⟩ *
| *

⟨*Logic* ⟩ ::= ⟨*Logic* ⟩ `&` ⟨*Logic1* ⟩
| ⟨*Logic* ⟩ | ⟨*Logic1* ⟩
| ⟨*Logic1* ⟩

⟨*Logic1* ⟩ ::= `~` ⟨*Logic1* ⟩
| `_`
| ⟨*Pattern* ⟩
| `(` ⟨*Logic* ⟩ `)`

⟨*Reg* ⟩ ::= ⟨*Reg* ⟩ | ⟨*Reg1* ⟩
| ⟨*Reg1* ⟩ − ⟨*Reg1* ⟩
| ⟨*Reg1* ⟩

⟨*Reg1* ⟩ ::= ⟨*Reg1* ⟩ ⟨*Reg2* ⟩
| ⟨*Reg2* ⟩

⟨*Reg2* ⟩ ::= ⟨*Reg2* ⟩ `*`
| ⟨*Reg2* ⟩ +
| ⟨*Reg2* ⟩ `?`
| `eps`
| ⟨*Char* ⟩
| `[` ⟨*String* ⟩ `]`
| `digit`
| `letter`
| `upper`
| `lower`
| `char`
| `_`
| ⟨*String* ⟩
| ⟨*Ident* ⟩
| `(` ⟨*Reg* ⟩ `)`

10.2 The Language Data

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of Data

Identifiers

Identifiers ⟨*Ident* ⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

String literals ⟨*String* ⟩ have the form `"`x`"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Data are the following:

There are no reserved words in Data.

The symbols used in Data are the following:

```
{  }  (
,  )  |
```

Comments

There are no single-line comments in the grammar.
There are no multiple-line comments in the grammar.

The syntactic structure of Data

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$\langle Input \rangle \quad ::= \quad \langle ListData \rangle$

$\langle Data \rangle \quad ::= \quad \{ \langle ListTokD \rangle \}$

$\langle TokD \rangle \quad ::= \quad (\langle String \rangle \ , \langle ListPattern \rangle)$

$\langle Pattern \rangle \quad ::= \quad \langle Ident \rangle \langle ListPattern1 \rangle$
$\quad | \quad \langle Pattern1 \rangle$

$\langle Pattern1 \rangle \quad ::= \quad \langle Ident \rangle$
$\quad | \quad$ `(` $\langle Pattern \rangle$ `)`

$\langle ListData \rangle \quad ::= \quad \epsilon$
$\quad | \quad \langle Data \rangle \langle ListData \rangle$

$\langle ListTokD \rangle \quad ::= \quad \epsilon$
$\quad | \quad \langle TokD \rangle \langle ListTokD \rangle$

$\langle ListPattern1 \rangle \quad ::= \quad \langle Pattern1 \rangle$
$\quad | \quad \langle Pattern1 \rangle \langle ListPattern1 \rangle$

$\langle ListPattern \rangle \quad ::= \quad \epsilon$
$\quad | \quad \langle Pattern \rangle$
$\quad | \quad \langle Pattern \rangle \ | \ \langle ListPattern \rangle$

References

[1] M. Forsberg, H. Hammarström, and A. Ranta. Morphological lexicon extraction from raw text data. *FinTAL 2006, LNAI 4139*, pages 488–499, 2006.

[2] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2007.

[3] F. Karlsson. Constraint grammar as a framework for parsing running text. *13th International Conference of Computational Linguistics*, 3:168–173, 1990.

[4] A. Voutilainen, J. Heikkil, and A. Anttila. Constraint grammar of english, a performance-oriented introduction. *University of Helsinki, publication no. 21*, 1992.