KATR

Software for morphological studies in computational linguistics

---

PROJECT

A project submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the University of Kentucky

By

Lei Shen

Lexington, Kentucky

Director: Dr. Raphael Finkel, Professor of

Department of Computer Science, Lexington, Kentucky

1999

ACKNOWLEDGMENTS

I want to express my sincere gratitude and appreciation to Dr. Raphael Finkel for his consistent guidance, suggestion, encouragement, and support. Also, I wish to thank my project committee members − Dr. Ken Calvert and Dr. Victor Marek, for their patience in reviewing this work and their valuable comments and suggestions.

I am grateful to Dr. Greg Stump of Department of English for his help and valuable examples.

I am eternally thankful to my wife, my parents, my sisters and my friends for their unconditional love, understanding, and support.

ABSTRACT OF PROJECT

KATR
SOFTWARE FOR MORPHOLOGICAL STUDIES IN COMPUTATIONAL LINGUISTICS

This project is to design and implement KATR, an extension of DATR. DATR is a language for defining nonmonotonic inheritance networks with path/value equations. It was designed specifically for lexical knowledge representation by Roger Evans and Gerald Gazdar in 1989. Since then, DATR has gained considerable success in the area of linguistic analysis, especially in describing the morphological structures of nature languages. We implement the full DATR language using the free, widely available and platform−independent language Java, and we enhance DATR with a new set construct that makes DATR more appropriate for certain linguistic analyses.

Chapter 1 DATR Fundamentals

Introduction

KATR is an implementation of an enhancement of DATR. DATR is a formal representation of default inheritance hierarchies used in morphological analysis of natural languages. The morphological structure of a natural language describes how words are modified to represent their grammatical position in a sentence. For example, Latin verbs have different forms to indicate the number and person of the subject, the tense, the mood, and the voice. The verb representing "I will love" is different in all these regards from the verb "She might have been loved." In any natural language, words with different base meanings (such as "love" and "hear") tend to have very similar morphological forms to indicate other aspects (such as "I will").

DATR allows the similarities in morphological formation to be abstracted out to a node high in an inheritance tree (such as a node called "VERB"). Classes of similar words might be grouped together at a lower level, such as "verbs with stem ending in e" versus "verbs with stem ending in a." Within each group, individual words may be placed. Queries can then ask how to construct a particular form of a word, such as "the first person singular future indicative active of the verb to love." In DATR, such a query might look like this:

Love:<mor 1 singular future indicative active>

This query would be answered by recourse to rules in the node Love, its parent node (perhaps "Verbs ending in e"), and its parent (perhaps "Verb"). Any exceptional formations would be found as rules in the first nodes in this list, which would be the lowest (most concrete) in the tree of abstraction.

Basic notions

We now turn to a description of how DATR represents morphological information. The best way to learn DATR, like any other language, is through examples.

```
Example 1-1: A simple node in DATR
    LovePresentParticiple:
        <syntactic category>  == verb
        <syntactic form>      == present participle
        <morphological form>  == love ing.
```

This short program defines a node, which describes the present participle form of the verb `love` using DATR. A ***node*** in DATR usually corresponds to a word, for example `love`, a lexeme, for example `all verbs`, or a class of lexemes, such as `all words`. In this particular example, we have a node `LovePresentParticiple` that has three rules. A ***rule*** is an equation in the DATR program. The left−hand side of the equation should always be enclosed in pointy brackets with a sequence of atoms inside. This left−hand side item is called a ***path***. The ***length of the path*** is the number of atoms in the path. The value associated with the path is the right−hand side of the equation. This value is a list of ***terms***, which can take various forms.

```
Example 1−2: Rule in DATR
    <syntactic form> == present participle
```

In example 1−2, the right−hand side value consists of two terms, both of which are atoms. We will see more detailed information about other forms later.

By default, the node name should start with an upper−case letter, and an atom should start with any non upper−case letter. This constraint can be explicitly overridden by an atom or node clause. However, we will assume no overrides in the rest of this paper unless otherwise noted for simplicity.

The DATR program of example 1.1 tells us that there is a node named `LovePresentParticiple` whose syntactic category is `verb`, whose syntactic form is `present participle`, and whose morphological form is `love ing`. That is how the DATR language describes the present participle form of the word `love`. A group (network) of nodes in DATR represents a body of knowledge called a ***theory***. The principle behind the DATR language is that we can categorize information hierarchically, represent it, then query the theory or generate the whole theory.

Suppose we already have a theory that consists of several nodes. (In example 1−1, we only have one node.) The theory is a body of knowledge, and the DATR engine is an automated tool to find the information we need. In DATR, a ***query*** is a question presented to the engine in the form `NodeName:path`, where `path` is in the form `<atom1 atom2>`.

The engine computes the answer to a query by following a set of steps within the specified node. The first step of this function is to find a ***matching rule***. If a matching rule is available, the DATR engine takes the right−hand side of the matching rule as the temporary result. Next, the engine checks whether this temporary result contains only atoms. A temporary result that contains atoms only is called a ***flattened result***, and this flattened result becomes the final

result. The DATR engine tries to ***flatten*** the temporary result, treating it as a new query, and keeps doing so until it gets a final result that contains atoms only. If a rule uses = instead of ==, then flatten is not to be called, and the right−hand side must already be flattened.

Let's talk about the matching algorithm first. Suppose we have a query NodeName:<atom1 atom2> and want to find a matching rule in the theory for this query. The engine first locates the node in the theory with the same name as the node name in the query. After that, it finds the rule whose left−hand side is the longest prefix of the path in the query. We say that path A is path B's ***prefix*** if A's first atom is B's first atom and A's second atom is B's second atom and so on. ***Length*** is measured in the number of atoms in the path.

Example 1−3: Longest Prefix for Matching Algorithm

```
Theory:  Node:
     <atom1> == value1
     <atom1 atom2> == value2.
Query:   Node:<atom1 atom2 atom3>
```

All rules' left−hand sides are prefixes of the query's path. In this case, however, the matching rule is <atom1 atom2> == value2, because <atom1 atom2> has length two, which happens to be the longest.

The right−hand side of the rule is a ***value***. A value consists of terms, which can be atoms or more complex structures shown now.

Example 1−4: Basic Form of DATR Term.

```
    VERB:
      <syntactic category>  == verb.

    LovePresentParticiple:
      <syntactic category>  == VERB:<syntactic category>
      <syntactic form>      == present participle
      <morphological form>  == love ing.

    LovePassiveParticiple:
      <syntactic category>  == VERB:<syntactic category>
      <syntactic form>      == present participle
      <morphological form>  == love ed.
```

Both node LovePresentParticiple and node LovePassive−Participle have value VERB:<syntactic category> as the value of their first rule. This value contains one term of the form ***Node:path***.

It is permissible to abbreviate the term by either omitting the node name and ":" if the desired node happens to be the current node, or by omitting ":" and <atom1 atom2> if that path is the same as the left−hand side's path.

```
Example 1−5: Abbreviated Form of the Term.
    VERB:
      <syntactic category>  == verb.

    Love:
      <syntactic category>  == VERB
      <morphological root>  == love
      <first gender>        == <morphological root>.
```

In the first rule of node Love, we specify the value as VERB, which has the same meaning as VERB:<syntactic category>. We omit <syntactic category> on the right−hand side because it is the same as the left−hand side of the rule. The last rule of node Love follows the same idea. We specify the value as <morphological root>, which is the same as Love:<morphological root>, since the desired node name is the same as the current node name.

A typical default rule looks like <> == ParentNode, in which the value of the default rule is a node name. A default rule, in which the path on the left is empty, has the DATR engine refer to the related node to find information not in the current node. In example 1−6, if the DATR engine cannot find information about the node Love, it tries to find it in node Verb. By such inheritance, a programmer can abstract common information in some high−level node and let all low−level nodes inherit it from the high−level nodes instead of repeating the information individually. This abstraction saves a lot of work and is an important DATR programming idiom.

```
Example 1−6: Inheritance by Default Rule.
    Verb:
      <> ==
      <syntactic category> == verb.

    Love:
      <> == Verb
      <root> == love.
```

The right−hand side of the default rule <> == in the node Verb is blank. This rule is valid and indicates that the default rule has null replacement.

Another form of a term is "Node:path". This form uses a pair of quote marks and is called a ***global term***, whereas Node:path is called a ***local***

term. Abbreviations apply to global terms as well as local terms; either the Node or the path may be omitted. Therefore, we have seven different forms of terms: `atom`, `Node`, `"Node"`, `path`, `"path"`, `Node:path` and `"Node:path"`. The detailed meaning of a global term will be fully explained later in the flatten algorithm.

All forms of terms mentioned so far are called ***basic forms***. In addition to these basic forms, DATR allows recursion in defining a term. A recursive term definition has the form: `Node:<term1 term2>`.

We need to strengthen the notation here. The distinction between `Node:path` and `Node:<term1 term2>` is that `path` can only be of format `<atom1 atom2>`, while `term1` and `term2` can be any valid term. Therefore, `Node:path` is an example of `Node:<term1 term2>`.

```
Example 1−7: Recursively Defined Term
    Node:<Node1:<atom1> atom2 "Node2">
```

Sanskrit example

A real−world Sanskrit example (Figure 1−1) provided by Dr. Greg Stump at the University of Kentucky) is listed here to demonstrate inheritance

The purpose of this program is to generate case forms of specific nouns and adjectives in Sanskrit. The node NOMINALS lists declensional properties that are (at least by default) associated with all of the nouns and adjectives at the bottom of the tree. These include (for example) the property of forming the accusative singular through the suffixation of −m and the property of forming the instrumental plural through the suffixation of −bhis. The node VOWEL_STEM_NOMINALS contains properties that are typical only of nouns and adjectives whose stems end with vowels; these include (for example) the property of forming the accusative plural through the suffixation of −n. The node I_OR_U_STEM_NOMINALS contains properties that are typical only of nouns and adjectives whose stems end in i or u; these include (for example) the property of forming the nominative dual without a suffix. Thus, the program and tree diagram account for the fact that Sanskrit nouns and adjectives fall into nested classes, where members of the same class share certain declensional properties while members of distinct classes exhibit distinct declensional properties.[1]

[1]Source: http://www.cs.uky.edu/~gstump/indicfragments/epicfragment
Result: http://www.cs.uky.edu/~gstump/indicfragments/epic00.html

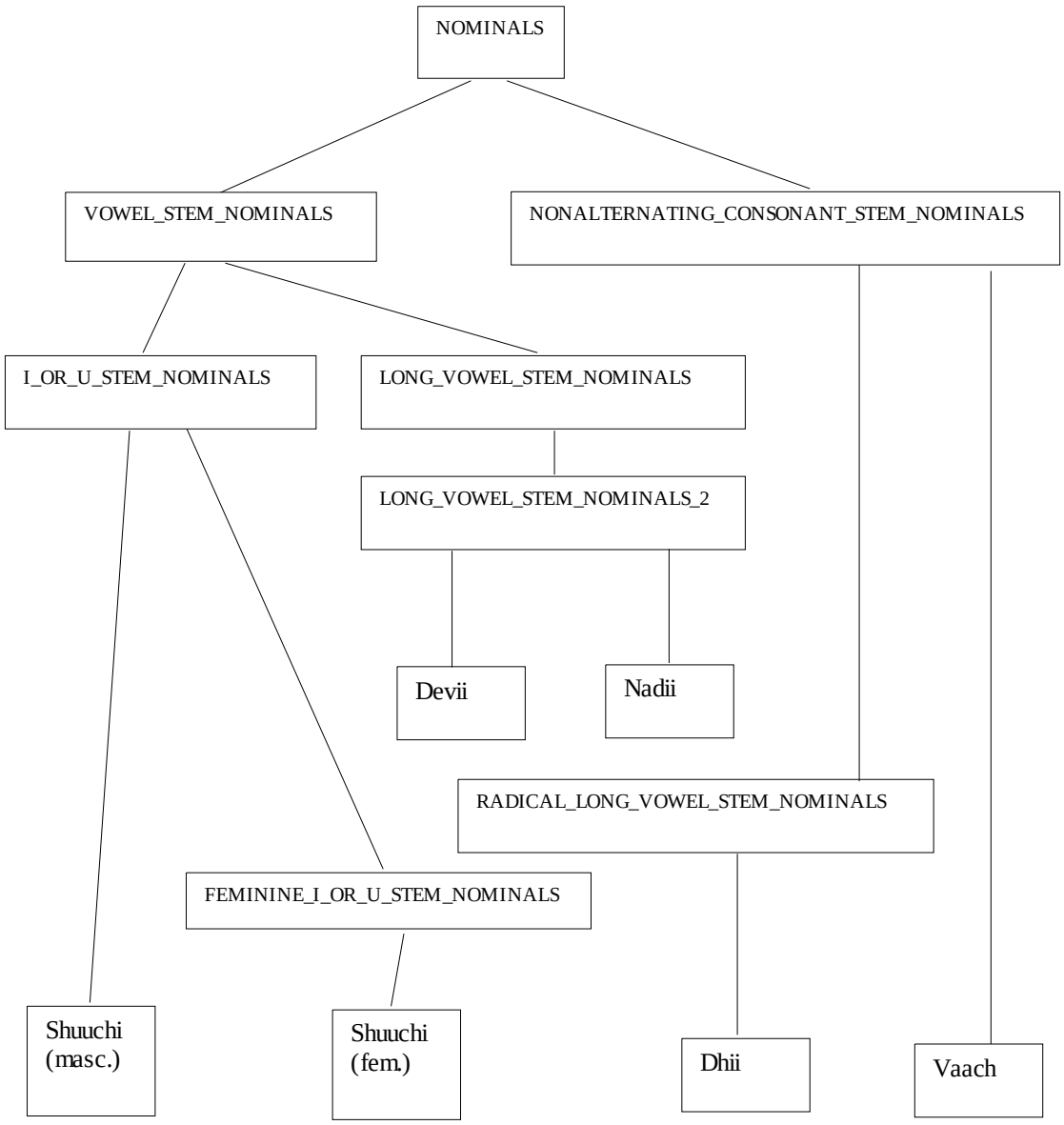


Figure 1−1

An ACL example

DATR is intended for describing morphological forms of natural languages. It can also be used in other realms. For example, the rights pertaining to files in an operating system are sometimes described by *access control lists (ACLs).* We can formulate ACLs by using DATR rules:

```
Superuser:
 <read> == true
 <write> == true
 <exec> == true
 <delete> == true.

Developer:
 <> == Root
 <write> == false
 <delete> == false.

User:
 <> == Developer
 <write user's source> == true.
 <delete user's source> == true.
```

The flatten algorithm

There are several methods available for modeling the flatten algorithm; we choose to use the local/global environment model here.

For every new query, DATR starts with two fresh pairs of environments: local and global. Both environments contain (Node, path) and are initialized based on the query. We will use LE to represent local environment and GE for global environment for simplicity.

During the flatten phase, the DATR engine tries to flatten every term of the temporary result. Because flattening has no side effects, it makes no difference in what order DATR flattens the term if it has several parts that need flattening. If the term being flattened is not an atom but is one of the basic forms, such as `Node:path`, `"Node:path"`, or their abbreviations, the local/global environment is updated according to following rules:

1 if term is `Node`, update term to `Node:LE.path`
   LE = `(Node, no change)`
   GE is unchanged
2 if term is `path`, update term to `LE.Node:path`
   LE = `(no change, path)`
   GE is unchanged
3 if term is `Node:path`,
   LE = `(Node, path)`
   GE is unchanged
4 if term is `"Node"`, update term to `Node:GE.path`
   LE = `(Node, GE.path)`
   GE = `(Node, no change)`
5 if term is `"path"`, update term to `GE.Node:path`

LE = (GE.Node, path)
GE = (no change, path)

6 if term is Node:path
LE = (Node, path)
GE = (Node, path)

In some cases, the path is supplemented, but we will discuss that later.

After the engine updates the term and the local and global environment, the updated term is in the form Node:path. The engine now takes this Node:path as a new query. Therefore, the query is broken into several small new queries. The engine keeps doing so until the query is flattened. Let's look at an example:

```
Example 1−8: Description of the DATR Engine.
    Verb:
     <present participle> == "<root>" ing.

    Love:
     <> == Verb
     <root> == love.

    Query: Love:<present participle>
```

In example 1−8, both local and global environments are initialized to (Love, <present participle>).

step a)
Matching rule: Love:<> == Verb.
Temporary result: Verb
LE: (Love, <present participle>)
GE: (Love, <present participle>)

step b)
Update term to: Verb:<present participle>
Update LE: (Verb, <present participle>)
GE is unchanged: (Love, <present participle>)

step c)
Take Verb:<present participle> as new query
Matching rule:
Verb:<present participle> == "<root>" ing.
Temporary result : "<root>" ing.

step d) flatten term "<root>"
Update "<root>" to Love:<root> according to rule 5
Update LE: (Love, <root>)

Update GE: (Love, <root>)

step e)
Take Love:<root> as a new query,
Matching rule: Love:<root> == love
Temporary result: love ing.

step f)
All terms of the temporary result are atoms now at this step; the final result is love ing.

Although the rules behind the DATR engine are simple, it is lengthy to write down every single step of its behavior. In the previous example, we see how the engine deals with the seven basic forms in the temporary result. For the recursive term, the fundamental idea is the same. The engine first breaks down the original problem into pieces and then uses a depth−first algorithm to flatten the recursion to arrive at a final answer.

```
Example 1−9: Flatten Complicated Term
    VERB:
      <mor form> == "<mor "<syn form>">"
      <mor present participle> == "<mor root>" s

    Love:
      <> == VERB
      <mor root> == love.

    LovePP:
      <> == Love
      <syn form> == present participle

  Query: LovePP:<mor form>.
```

The Analysis:
Initialize both LE and GE to: (LovePP, <mor form>)

Step a)
LovePP:<mor form> matches the rule <>==Love in node LovePP.
The temporary result is Love.

Step b)
Update Love to: Love:<mor form>
Update LE to: (Love,<mor form>)
GE is unchanged: (LovePP,<mor form>)

Step c)

Love:<mor form> matches the rule <>==VERB in node Love.
The temporary result becomes VERB.

Step d)
Update VERB to VERB:<mor form>
Update local environment to: (VERB,<mor form>)
The global environment is unchanged: (LovePP,<mor form>)

Step e)
VERB:<mor form> matches the rule <mor form> in node VERB.
The temporary result becomes "<mor "<syn form>">"

Step f)
The term is recursive. Solve "<syn form>" first
Update "<syn form>" to LovePP:<syn form>
Update LE to (LovePP, <syn form>)
GE is unchanged: (LovePP, <syn form>)

Step g)
Get LovePP:<syn form> == present participle
Update temporary result to: "<mor present participle>"

Step h)
Update term to LovePP:<more present participle>
Both LE and GE: (LovePP,<mor present participle>)

Step i)
Based on the rules above, the engine now flattens the recursive term and derives the final result is love ing.

A DATR program can have more complicated rules with recursion, such as those in the Swahili program (shown later). The DATR engine may need many steps to solve the query. However, the rules it uses are the same. Because the rules that the engine uses to solve the query are the heart of the DATR, we present another example here.

It is highly recommended for any reader to go through this example without referring to the trace first. If you can understand this example, then you have grasped 80% of DATR.

Example 1–11: Last Local/Global Environment Example:

```
Node1:
  <choice a> == "<choice b>"
  <choice b> == you are kidding.
```

```
Node2:
  <choice a> == Node1
  <choice b> == should be.

Node3:
  <choice a> == no way to be here
  <choice b> == is here ?
  <query>    == "Node2:<choice a>".
```

A brief trace for the query `Node3:<query>` is:

Step a)

Temporary result: `Node2:<choice a>`
LE and GE: `Node2:<choice a>.`

Step b)

Temporary result: `Node1`
Term updated to: `Node1:<choice a>`
LE: `(Node1,<choice a>)`
GE: `(Node2,<choice a>)`

Step c)

Temporary result: `"<choice b>"`
Term updated to: `Node2:<choice b>`
LE and GE: `Node2,<choice a>)`

Step d)

Final result: `should be.`

Clauses

DATR allows some clauses that help us to write DATR program more easily.

We have already mentioned atom and node clauses. These two clauses override the default naming convention of the DATR program.

```
Example 2−1:
      #atoms NODE, UPPER.
      #node  small, lower.
```

In example 2−1, we define two atoms `NODE` and `UPPER` beginning with upper−case letters and two nodes `small` and `lower` beginning with lower−case letters.

Hide, ***show*** and ***showif*** clauses are very useful for querying. Consider the

following case: We have a theory of 10000 different verbs and want to ask for the present participle form for every single verb. Instead of writing these 10000 queries down, we can achieve the same goal by the following.

```
Example 2-2: Hide and Show Statements
    VERB:
      <present participle> == "<root>" ing.

    Love:
      <> == VERB
      <root> == love.

    Hate:
      <> == VERB
      <root> == hate.

    ... % 9998 other verbs defined here.

    #hide VERB.
    #show <present participle>.
```

The ***hide*** statement lists the nodes that we don't want to see when the full theory is presented. DATR programs typically hide all nodes from which others inherit (that is, internal nodes in the inheritance tree). The ***show*** statement lists the paths that we would like to see when the full theory is presented.

The ***showif*** statement lists paths to be presented for nodes that satisfy given conditions. We just give an example without explanation because it is quite straightforward.

```
Example 2-3: Showif Statement
    VERB:
      <syntactic category> == verb
      <present participle> == "<root>" ing.

    NOUN:
      <syntactic category> == noun
      <plural form>  == "<root>" s.

    Love:
      <> == VERB
      <root> == love.

    Student:
      <> == NOUN
      <root> == student.

    #hide VERB NOUN.
    #showif <syntactic category> == VERB
```

```
    #then <present participle>.
    #showif <syntactic category> == NOUN
    #then <plural form>.
```

The ***vars*** clause allows the programmer to define variables. Wherever a defined variable appears in the program, the rule in which it appears is replicated for all possible values of that variable. A variable must start with the $ sign.

```
Example 2−4: Var Statement
   #vars $vowel:a e i o u.
   #vars $nonv: a b c d f g h j k l m n p q r s t v w x y z.

   DoubleVowel:
      <>  ==
      <$vowel> == $vowel $vowel <>
      <$nonv>  == $nonv <>.

   Love:
      <strange> == DoubleVowel:<l o v e>.
```

This program doubles all the vowels in the word while keeping the non−vowel letters unchanged. For example: Query `Love:<strange>` yields the result `l o o v e e`.

Unused parts of queries

It is impossible to derive the trace to get `l o o v e e` by just using the rule we mentioned before. The rules we introduced to update local/global environment are not 100% right; they need some small modification. We have six rules corresponding to the six different basic forms, `Node, path, Node:path, "Node", "path"` and `"Node:path"`. The modification affects the four that involve paths. The change we make is that the DATR engine appends the unused part of the query path (the part not matching the prefix on the left−hand side) to every path on the right−hand side.

```
Example 2−5: Old/New Rule Comparison
        Node:
           <atom1 atom2> == Node1 Node2:<>.
        Query:  Node:<atom1 atom2 append1 append2>
```

Temporary result under the old rule:

```
      Node1:<atom1 atom2> Node2:<>
```

Temporary result under the new rule:

```
Node1:<atom1 atom2> Node2:<append1 append2>
```

The transformations of `Love:<strange>` are:

```
Love:<strange> ==> DoubleVowel:< l o v e >
               ==> l < o v e >
               ==> l o o < v e >
               ==> l o o v < e >
               ==> l o o v e e.
```

Chapter 2 KATR

This project implements the DATR language with enhancements using the platform−independent programming language Java.

Sets

In addition to basic DATR functionality, we add sets to KATR to make it more useful in describing languages such as Swahili. The set idea comes from the following.

Suppose a theory says that `Love:<3rd−person present>==loves`. The DATR query `Love:<3rd−person present>` will return the result `loves`, but the query `Love:<present 3rd−person>` does not work as we hope. DATR requires that queries be in the right order because of a shortcoming in its matching−rule definition. A matching rule is the longest prefix of the query path, which takes into account the explicit order of the atoms in the matching rule.

One way to solve this problem is to list all rules in all possible query orders. This solution is not practical because the number of combinations increases exponentially. Until now, researchers have accepted the restriction that queries must be carefully ordered. However, Swahili demonstrates that this restriction is too confining, as shown in Example 3−1.

```
Example 3−1 Particular order is not enough.
          Word:
           <> ==
           <1sg> == n i <>
           <neg 1sg> == s i <>
           <past>  == l i <>
           <neg past> == k u <>.
```

We have three properties here: `1sg`, `neg` and `past`. In this case, no matter what kind of particular order we request for queries, it is impossible to solve the query `<neg 1sg past>` correctly without replicating `neg` in the query (the desired result is `s i k u`).

We realized this problem when we tried to use DATR to describe Swahili, Bulgarian and other languages. We tried to solve this shortcoming of DATR by using replication but found that sets provide a more elegant and general solution. The basic idea is that the left−hand side of a rule may contain a *set* of atoms instead of an ordered path; such a rule matches any query that contains

all the given atoms.

In order to build sets into KATR, we slightly modify the syntax of DATR. We use #<atom1 atom2># to indicate a set rule in KATR. The semantics of DATR are modified also. The DATR engine has two main algorithms, the matching algorithm and the flatten algorithm. The DATR matching algorithm finds a rule whose path part is the longest prefix of the query's path. In the case of set rules, matching finds the rule whose set is the largest subset of the query's path, where the length is still defined as the number of elements in this rule's path. Therefore, DATR considers all rules whose path is a prefix of the query if the rule is a regular rule or whose path is a subset of the query if the rule is a set rule. Then DATR selects the longest rule among them as the matching rule. If an element (an atom or a variable) appears more than once in a set rule, it must match the query as many times as it appears. In this regard, KATR sets are really multisets.

```
Example 3-2: Set Rule
    Love:
    #<3rd-person present-tense># == loves.
```

Both queries

```
  Love:<3rd-person present-tense>
  Love:<present-tense 3rd-person>
```

produce result loves.

We provide another real-world KATR program here to show how to use sets. (Thanks to Dr. Greg Stump.)

```
Example 3-3: Swahili Example
#vars $abc: a b c d e f g h i j k l m n o p q r s t u v w x
y z.

SANDHI: % elisions
  <> ==              %default: succeed, no replacement
  <a a> == a <>
  <a u> == u <>      %some combinations are replaced
  <$abc> == $abc <>. %other alphabetic characters retained

VERB1:
  #<># ==
  #<negative># == h a.

VERB2:
  #<1 sg># == n i
  #<2 sg># == u
  #<3 sg># == a
  #<1 pl># == t u
```

```
  #<2 pl># == m
  #<3 pl># == w a.

VERB3:
  #<past># == l i
  #<negative past># == k u
  #<future># == t a.

VERB12:
  #<># == VERB1 VERB2
  #<negative 1 sg># == s i.

VERB:
  #<># == SANDHI:<VERB12 VERB3 "<root>">.

WANT:
  #<># == VERB
  #<root># == t a k a.

#hide SANDHI VERB VERB12 VERB1 VERB2 VERB3.
#show
  <1 sg positive future>.
```

The trace for query `WANT:<1 sg positive future>`

```
1) WANT:<1 sg positive future>
2) VERB:<1 sg positive future>
3) SANDHI:< VERB12<1 sg positive future>
            VERB3<1 sg positive future>
            WANT:<root> >
4)SANDHI:<
    VERB1<1 sg positive future> VERB2:<1 sg positive future>
    t a
    t a k a >
5) SANDHI:<
    n i
    t a
    t a k a >
6) n i t a t a k a
```

If we restrict ourselves to classical DATR, node `VERB12` needs to put `negative, 1 sg` together, and node `VERB3` needs to put `negative, past` together. That's impossible in DATR. However, example 3−3 demonstrates that we can solve this order problem by using sets, and `VERB1`, `VERB2` and `VERB3` all generate correct results.

Negation

Another enhancement we add in KATR is that the ! sign before an atom in a set rule means that this atom must not appear in the query for matching to succeed.

The negated atom is not matched against any particular atom in the query; it merely represents an atom that must not be present.

```
Example 3-4: Negative in Set Rule
    Node:
    #<a b d># == a b d
    #<a b !c d># == a b no-c d.
```

Query `Node:<a b d>` gives answer `a b no-c d`, because both rules match this query and the length of `#<a b !c d>#` is longer. However, query `Node:<a b c d>` gives answer `a b d` because rule `#<a b !c d>#` does not match this query because it has `c`.

The negation sign may also be placed before a variable, in which case it means that no possible value of this variable may appear in the query. If an atom or variable appears negated twice in a set, it is the same as if it appeared only once, but the effective length of any match is longer.

Negation provides a convenient way to represent certain language rules. As a result, negation simplifies the KATR program and makes the program to be easier understandable and maintainable. Currently, Dr. Greg Stump is using negation in his Bulgarian research.

Below is a short KATR program example provided by Dr. Greg Stump that generates present-tense and past-tense paradigms for the verbs `walk` and `be` in English. As you can see, the default past-tense form for `be` is `were`. In the first- and third-person singular of the past tense, however, `be` has the form `was`. Using the notation `!second_person` in the rule introducing `was` simplifies matters; without `!`, it would instead be necessary to have two rules (i.e. `#<past first_person singular># == was` and `#<past third_person singular># == was`).

```
Example 3-5 Negation Example:

VERB:
   #<># == "<stem>" "<suffix>"
   #<suffix present third_person singular># == s
   #<suffix past># == e d
   #<suffix># ==.

Walk:
   #<># == VERB
   #<stem># == w a l k.

Be:
   #<># == VERB
   #<stem present first_person singular># == a m
```

```
   #<stem present># == a r e
   #<stem present third_person singular># == i
   #<past !second_person singular># == w a s
   #<past># == w e r e.

#hide VERB.
#show
  <present first_person singular>
  <present second_person singular>
  <present third_person singular>
  <present first_person plural>
  <present second_person plural>
  <present third_person plural>
  <past first_person singular>
  <past second_person singular>
  <past third_person singular>
  <past first_person plural>
  <past second_person plural>
  <past third_person plural>.
```

Implementation decisions

KATR's design is upward−compatible with DATR: all DATR programs run in KATR without any modification and give the same result.

We also considered the following issues when we designed KATR: free availability, ease of installation and completeness.

When we decided to use DATR as a tool for computational linguistic research at the University of Kentucky, we tried several implementations available at that time: DATR2.7, from the University of Sussex UK, QDATR, from the University of Düsseldorf Germany, and ZDATR, from the University of Bielefeld Germany.

DATR 2.7 is based on Prolog and needs to be customized for the underlying Prolog implementation because there are so many dialects of Prolog. Based on my experience, it is really hard for a linguist (most likely not a computer guru) to find a usable Prolog, install and customize DATR. Moreover, the best−supported Prolog (Sicstus Prolog) is not free. QDATR has versions for Windows, MAC and Unix, but it runs slowly. ZDATR is written in C and provides only source code, which must be compiled for underlying system because of the incompatibility of different C compilers. None of the three implementations has a full implementation of the DATR language. They all lack `showif` functionality, for example.

We decided to provide our own enhanced implementation of DATR after suffering from these problems. We choose to use Java, which is a free, platform−independent language, to implement KATR. People can install KATR easily due to the Java's "write once, run everywhere" property. We also

implemented all functionality of DATR, including `showif`.

Chapter 3 KATR implementation

BNF

A good starting point for implementing a compiler is to have a BNF[1] (Backus Normal Form) syntax definition. We started with the BNF in the ZDATR implementation and modified it to meet our needs.

```
<katr theory>  ::= <vars clauses> <katr theory> |
                   <atom clauses> <katr theory> |
                   <node clauses> <katr theory> |
                   <show clauses> <katr theory> |
                   <hide clauses> <katr theory> |
                   <showif clauses> <katr theory> |
                   <sentence> <katr theory> |
                   <sentence>

<var clause>      ::= #vars <var name> <var value list>
<var name>        ::= <identifier>
<var value list>  ::= <var value> |
                      <var value> <var value list>
<var value>       ::= <identifier>

<atom clauses>    ::= #atom <atom list> .
<atom list>       ::= <null> | <atom> | <atom> <atom list>
<null>            ::= ''
<atom>            ::= <identifier>

<node clauses>    ::= #node <node list> .
<node list>       ::= <null> | <node> | <node> <node list>
<node>            ::= <identifier>

<show clauses>    ::= #show <simple path list> .
<simple path list>::= <simple path> |
                      <simple path> <simple path list>
<simple path>     ::= <regular path> | <set path>
<regular path>    ::= '<' <item list> '>'
<set path>        ::= '#<' <item list> '>#'
<item list>       ::= <null> | <item> | <item> <item list>
<item>            ::= <identifier>

<hide clauses>    ::= #hide <nodelist> .

<showif clauses>  ::= #showif <simple path>
                          == <query result>
```

[1] The proposal that BNF, which begin as an abbreviation of Backus Normal Form, be read as Backus−Naur Form, to recognize Naur's contributions as editor of the Algol 60 report.(Naur 1963) is contained in a letter.(Knuth 1964). The highly influential Algol 60 report (Naur 1963) used Backus Naur Form (BNF) to define the syntax of a major programming language. The equivalence of BNF and context−free grammars was quickly noted, and the theory of formal languages received a great deal of attention in the 1960's.

```
                        #then <simple path list> .
<query result>    ::= <null> | <atom list>


<sentence>        ::=  <node> : <rule list> .
<rule list>       ::=  <rule> | <rule> <rules>
<rule>            ::=  <simple path> == <term list> |
                       #<simple path># == <term list>
<term list>       ::=  <null> | <term> |
                       <term> <term list>
<term>            ::=  <node> | <atom> | <var name> |
                       <node> : '<' <term list> '>' |
                       '<' <term list> '>' |
                        " <term list>"
```

We used JavaCC to help us write the compiler. JavaCC (Java Compiler's Compiler) is a free software tool provided by JavaSoft. For those who have used lex and yacc, JavaCC is an equivalent set of tools tailored to Java. JavaCC takes a BNF specification and generates a parser. JavaCC also provides useful information in debug mode, which can greatly benefit developers.

Data structures

The main data structures used in KATR include `Term`, `Rule`, `EleInNameTable` and `Theory`.

```
class Term {
  String   nodeName;
  Vector   subTerms;
  boolean  isGlobal;
}
```

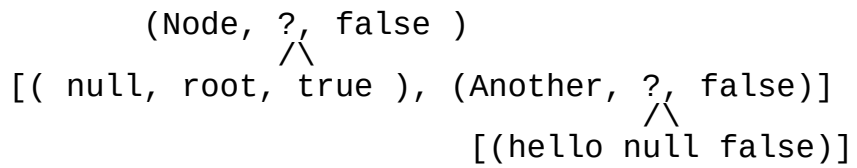
The field `nodeName` represents either the name of the node or the atom if any. The `isGlobal` field represents the difference between a quoted term (global) and an unquoted term (local). The most complicated field is `subTerms`, which is an array of terms within the path.

We use a triple `(nodeName,subTerms,isGlobal)` to represent a term and use `[element1,element2]` to represent a vector of atoms.

```
Example 4-1: Basic Term Representation:
atom                    (atom, null, false)
Node                    (Node, null, false)
"Node"                  (Node, null, true)
<atom1 atom2>           (null, [atom1, atom2], false)
"<atom1 atom2>"         (null, [atom1, atom2], true)
Node:<atom1 atom2>      (Node, [atom1, atom2], false)
"Node:<atom1 atom2>"    (Node, [atom1, atom2], true)
```

KATR allows recursion within a path, so much more complex terms can be constructed.

Example 4−2: Complex Term Representation.
<"root" Another:<hello>> is represented as



The path (left−hand side) of a rule is also represented as a term, a special kind of term whose subterms are all atoms.

Class `Rule` is for representing rules.

```
class Rule {
  Vector  leftside;
  Vector  rightside;
  boolean isSet;
}
```

The left−hand side is an array of atoms, and the right−hand side is an array of terms. Both are represented as a vector of strings instead of Terms after parsing has completed. The purpose of `isSet` is to indicate whether this is a set rule, which uses #<># instead of <>. For example, for the rule

`Node:<p1 p2 p3> == Node1 Node2:<q1 q2>`,

`Rule.leftside` is `[p1, p2, p3]`, `Rule.rightside` is `[(Node1, null, false) (Node2, [q1,q2], false)]`, and `Rule.isSet` is `false`.

Class `EleInNameTable` represents both nodes and atoms.

```
Class EleInNameTable{
   String  strType;
   String  strKey;
   Vector  rules;
}
```

The first member is `strType`, which specifies whether to represent a node or an atom. The second member is `strKey`, which is the node name if

representing a node, and is the atom name if representing an atom. The last member is a vector. The value of this vector is null if the instance represents an atom, and contains all rules associated with the current node if the instance represents a node.

Class `Theory` is built of nodes and other information.

```
class Theory{
   Hashtable nameTable;
   Hashtable varTable;
   Hashtable hideTable;
   Vector    showVector;
   Vector    showifVector;

   long      MAXRECURSIVEDEPTH;
   Hashtable mapTable;
   String    globalNode;
   Vector    globalPath;

   Hashtable identifierTable;
   Hashtable termTable;
}
```

The data members in the first group are used during both the parsing and querying phases. Data member `nameTable` is a hash table for quick reference to all the nodes and atoms defined. The key for `nameTable` is the node or the atom name. The value of `nameTable` is an instance of class `EleInNameTable`. Field `varTable` is a quick reference to all the variables defined by the program. For example,

```
#var $vowel: a e i o u.
```

adds a (key value) pair as `(vowel, [a,e,i,o,u])`. Field `hideTable` contains all the nodes that appear in the hide clauses.

Both `showVector` and `showifVector` are vector types. `ShowVector` contains all paths defined by the show statements. The information carried by the `showif` statement is stored in field `showifVector`. This vector has three components for a single `showif` statement: a vector of atoms in the `if` part, a vector of atoms in the `then` part, and all paths that need to be shown under this condition. These fields are filled during the parsing phase and used only in the query phase.

Example 4−3: the representation of `#showif` statement

```
#showif   <category> == verb
   #then   <present participle>.
#showif   <category> == noun
   #then   <plural>.
```

The vector for storing these showif information is:

```
[ [category] [verb] [present participle]
  [category] [noun] [plural] ]
```

The next group of data is used for querying. Read−only field MAXRECURSIVEDEPTH defines the maximum recursion we allow during the query. This feature protects against mistakes in a program.

Example 4−4: A bad query that needs to be protected against.

```
Node: <atom1> == <atom2>
      <atom2> == <atom1>.
```

Query `Node:<atom1>`

When variables are introduced by `#vars` statements, the `mapTable` field records the value for every variable. Fields `globalNode` and `globalPath` record the global environment.

Other data members in the class `Theory` reduce memory use. `IdentifierTable` and `termTable` hash every identifier (string) and every term that appears in the KATR program. We only build a reference to represent them when the same identifier or term appears again. This reuse saves memory.

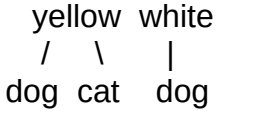
Algorithm

The algorithm implementation is a straight translation from the methods introduced before into the Java implementation. However, we came up with several ideas to improve the speed of matching.

First of all, we use a tree−structure dictionary to represent all the regular rules. For example: If we have the following rules:

```
<yellow dog> == foo
<yellow cat> == bar
<while dog> == baz
```

The logical tree−structure dictionary representation is:



This dictionary−like structure allows matching to stop the comparison early if appropriate.

For each set rule, we use a hash table to store those atoms in this rule before the comparison. Consider the following case: The rule is `#<foo bar baz>#` and the query is `<ha foo ti bar nu baz>`. We first build a hash table for the query. We use this hash table to check whether every atom in the rule is in the query. The cost for this approach is the time for building the hash table plus the time for checking the atoms in the rule, which is linear in the number of atoms in the rule and the query. On average, this cost is less than the cost if we enumerate every possible order of the query and do a regular matching against each of these orders.

Second, because the matching rule is the longest prefix, we always record the current maximum matching length and do not bother to consider those rules whose lengths are smaller than the current maximum matching length. We do not sort the rules at compile time by length to make this optimization faster, because there are generally few rules in a node, so there would not be much improvement, and the speed of KATR appears to be sufficient as it is.

Finally, we only create one instance for each string. Therefore, all the string comparisons become reference comparisons, which are much faster.

Here we provide a brief outline of the algorithm implementation. Please refer to the source code for a more detailed explanation.

```
Matching algorithm:
longest_match_length = −1;
   longest_match_rule   = null;
   locate the appropriate node
   for every rule in this node
       if (length of this rule < longest_match_rule)
           continue;
       if it doesn't match continue
       if ((regular rule && prefix) || (set rule && subset))
           longest_match_rule   = current rule
           longest_match_length = current rule's length
       endfor

Answering query:
   update query if necessary
   initialize/update local/global environment if necessary
   if(find matching rule)
       for every term in the right hand side
            flatten each term according to the rule
   else
       error
```

Chapter 4 Testing

We tested KATR in four steps: parsing−level testing, data−structure verification, algorithm testing, and integration testing. We also created a test suite that can be used as well for other DATR or KATR implementations.

The purpose of the parsing−level test was to make sure the KATR parser actually accepts a KATR program with correct syntax and rejects KATR programs with grammatical errors. The parsing−level test consisted of three parts: Token testing ensured that the parser separates the KATR program into tokens, rule testing checked that the KATR implementation recognizes each rule in the KATR program, and node testing parsed a whole piece of code defining a node.

Data−structure verification ensured that what we store in the computer memory is complete and accurate. It should be possible to construct an equivalent KATR program by using the data structure in memory. We first dumped out the data structure after parsing a KATR program in human−readable format. This readable information is organized according to KATR syntax: The data dump is a well−formatted standard KATR program. Finally, we compared this well−formatted program to the original KATR program to verify that our memory representation of the KATR program is correct.

We have two algorithms in the KATR implementation; the matching algorithm and the flatten algorithm. We tested the matching algorithm in three steps. In the first step, we tested DATR regular rules to see whether the rule with the longest prefix is selected. The second step tested set rules only and ensured the longest subset is selected. In the last step, we mixed both regular and set rules to check whether the result was the desired one.

The flatten algorithm is the lengthiest part of the KATR program and therefore received the most attention during the test phase. Testing began with the seven basic forms. After these basic forms were verified, recursive forms were tested. Test cases for complicated terms were constructed for this purpose to make sure recursion does work.

A final test was integration testing. During this phase, real KATR programs were used to verify the implementation. In this phase, many existing DATR programs were tested while representative and real KATR programs using sets were also constructed. During the integration phase, we also worked on tuning the performance. Most of the ideas about how to speed the matching phase were proposed at that phase. (Many thanks to Dr. Raphael Finkel for his suggestions).

During testing, most errors were picked up in the stages before integration testing. Coding KATR was not hard because its rules are clearly defined.

The parser part of the KATR implementation is about 600 lines, and the engine part is about 2000 lines. Dr. G. Stump at the University of Kentucky is currently using the KATR implementation seriously as a research tool, and he reports that the KATR implementation runs smoothly and quickly. The implementation is available at ftp://ftp.cs.uky.edu/cs/software/katr.tar.gz.

From this project, I learned several things. First, I became more familiar with Java programming, especially in writing a compiler. Second, I learned how to get a detailed specification from a vague starting requirement. Last, I learned how to add a new feature into an existing software package while considering all kinds of tradeoffs. I also learned how to write a technical document.

The current KATR implementation, version 1.0, only provides basic interfaces for compiling a KATR program and getting the whole theory. We will consider providing a more easy−to−use and powerful interface in a future version.

References:

The DATR Web Pages at Sussex
http://www.cogs.sussex.ac.uk/lab/nlp/datr/datr.html

ZDATR implementation
http://coral.lili.uni−bielefeld.de/DATR/Zdatr

QDATR implementation
http://www.phil−fak.uni−duesseldorf.de/sfb282/B3/qdatr.html

Naur. P. (1963)
"Revised report on the algorithms language Algol 60," Comm. ACM **6**:1, 1−17

Knuth, D. E.[1964].
"Backus Normal Form vs. Backus Naur Form," Comm. ACM **7**:12 735−736