TR-I-0294

Easier C programming
Some useful objects

Yves Lepage

November 1992

Abstract

This report describes the basic work done to pave the way for our studies about pattern-matching and distances applied to linguistic objects. We propose a library of basic objects ranging from booleans to forests. These objects come with a set of basic functions.

Keywords

Programming in C, object oriented programming, booleans, atoms, strings, lists, forests, feature structures, transformational rules.

Contents

List of Figures

Introduction

This document describes the basic foundations of the programming we have done to pave the way for our experiments concerning pattern-matching on linguistic objects and the measurement of distances between such objects.

This programming work has been done in an object-oriented spirit. Its result is a set of modules in C describing classes of objects. For each class `class`, there is a `class.h` file where the class is defined and where certain functions on the class are declared. The definitions of these functions are to be found in the file `class.c`.

Below, we describe the lattice of classes defined so far. Because the definition of some classes required the definition of classes of lower level, the classes are organised in a lattice, in the following way:

```
                     booleans
                          |
                    characters
                          |
                       atoms
         -----------------|-------------
       lists                           |
  -------|------                       |
sets stacks strings_________________forests
                        |        ------|---------
                     boards   feature  transf  woods
                            structures  rule
```

This organisation is expected to change. During our development work, we have experimented with different techniques. Mainly we tend to use more and more general techniques which allow us to implement faster. But this work is a long-term task which requires frequent reprogramming of the work already done. Each time this happens, we see an increase in the possibilities we implement and a relative decrease in the length in lines of our programs.

To mention current work, we are at the moment introducing a general class element which will be used as a temporary descriptor in the definition of new classes. We hope it will allow an implementation of the class `wood` in a way closer to its formal definition.

1 Classes

The following is a description of some of the classes which are the most important for our work on natural language processing.

The set of classes already implemented can be divided into three types:

- basic classes deal with primitive objects. They are *booleans*, *characters* and *atoms*.

- the string side. All these classes are based on class *list*. They are used for computational purposes (*set* and *stack*), but *string* is intended for linguistic treatment.

- the tree side. These classes are for linguistic purposes. They are used for representing linguistic structures (*forests*, *feature structures*) or for manipulating them (*transformational rules*).

Another class, that of *woods*, is currently being reimplemented in a way closer to its formal definition. Its place is above that of strings and trees. Because this data structure is unusual and needs justification, it will be described, together with its associated operations, in a separate document.

1.1 Basic classes

1.1.1 Booleans

This is the simplest class. It consists only of the definition of the type *boolean* as an integer with two values: TRUE and FALSE. Only two functions are avaliable, *i.e.* the methods to input and output the values of a boolean variable.

The standard C operations provide the operations for this class.

1.1.2 Characters

The class of characters is fundamental for input and output. The detection of kanji is realised by checking whether the first bit is set. A set of functions allows one to read the input according to requirements. Here are some of them:

- skipping text:
  - `skipUntil` skips everything until a given constant string is reached;
  - `skipComment` skips the following on the input if it is a comment following the C syntax;
  - `skipSpace` skips the following on the input if it is a sequence of space characters (blank, tab or newline);
  - *etc.*
- read definite text:
  - `readChar` reads the next character if it is the given constant character, else fails;
  - `readString` reads the next character if it is the given constant string, else fails;
  - *etc.*

1.1.3 Atoms

The class of atoms was designed to cover that of words used in natural language sentences, and a range of symbols used in the feature structures output by the various phases of the NADINE system.

The syntax of *atom* is given in Figure 1.

```
<atom> ::= <qatom>
         | <natom>

<qatom> ::= <qsubatom> [ <natom> ]

<natom> ::= <nsubatom> [ <qatom> ]

<qsubatom> ::= ' <character>* '

<character> ::= any character except '

<nsubatom> ::= <form>*

<form> ::= <letter>
         | <kanji>
         | <digit>
         | <sign>

<sign> ::= * | - | + | _ | / | | | . | : | ^ | $ | ? | !
```

Figure 1: Syntax of atoms

This syntax allows one to write the four atoms given as examples in Figure 2.

```
office'?'         ' blanks '         +         SEM-FEATURE
```

Figure 2: Four atoms

1.2 Lists, sets, stacks and strings

1.2.1 Lists

Definition A list is recursively defined as something having a head and a tail which is of type `list`. The tail may be empty. The head is an element. The usual syntax of lists, using parentheses and commas, is used. When elements are atoms, lists can be strings of words, *i.e.* sentences.

```
<string> ::= ( [ <atom> [ , <atom> ]* ] )
```

Figure 3: Syntax of lists

Functions Apart from the functions for creation, deletion and copying, functions reflecting the structure of a list are proposed:

- insertion of an element as head of a list;
- concatenation of two lists;
- extraction of the n^{th} element of a list.

Other functions are:

- test whether an element is in a list;
- test whether a list is empty;
- length of a list;

Moreover, an abbreviated control structure is proposed which carries out a predefined traversal of the list structure. In case the coding of the structure is changed, the exploration of a list will still bear the same name, independent of its implementation.

1.2.2 Sets

Sets are implemented here as lists with constraints: in the internal representation, an element must not occur more than once. Furthermore, order has no importance. The usual syntax, using curly brackets and commas, is used.

```
<set> ::= { [ <atom> [ , <atom> ]* ] }
```

Figure 4: Syntax of sets

When inputting a set, the user can enter an element more than once. The input program will check for repetitions and eliminate them. Figure 5 shows some examples of sets.

{a,a,a,a}	stands for	{a}
{a,c,b,a}		{a,b,c}
{b,c,a}		{a,b,c}
{}		the empty set

Figure 5: Examples of sets

1.2.3 Stacks

The class of stacks is in some respects a weaker version of on the class of lists. Apart from the creation and deletion functions, only three functions are proposed for it. They are the usual functions for stack handling:

- isemptyStack tests whether the stack is empty;
- push pushes an element on the top of the stack;
- pop pops the top element from the stack;

1.2.4 Strings

This is not a true class. It is just a different interface for the class `list`. All the methods defined for lists are available for strings. This pseudo-class was designed for the natural language strings extracted from the conversation dialogues. Figure 6 shows the syntax of strings and Figure 7 shows a string of six atoms.

```
<string> ::= "  <atom>* "
```

Figure 6: Syntax of strings

```
"Is this the conference office '?'"
```

Figure 7: Example of a string

1.3 Forests, feature structures

1.3.1 Forests

Definition The syntax of forests we adopted is given in Figure 8. It defines a forest as a node dominating a daughter and having a sister. The daughter and the sister are both forests. They are optional.

```
<forest> ::= <atom> [ ( <forest> ) ] [ , <forest> ]
```

Figure 8: Syntax of forests

With this definition, a tree is a forest whose top node has no sister.

Parenthesised and drawn forms A forest is input or output either in a parenthesised form or in a drawn form.

The parenthesised form is the one given by the syntax above. It is the usual one with parentheses and commas.

Figure 9 gives an example of a tree drawn by a user. It is recognised by the system as long as nodes are touched by an underscore or by a vertical bar. The second tree is the same tree output, formatted in a standardised way by the system.

```
        OBJE
  ------|---------------------
 ASPT  TENSE    PARM    RESTR
  |     |        |      |____
  _  PRESENT   !X1    RELN  ENTITY
                      |        |
                YES-ADV-1     !X1
                              _|_
                             P1 P2

               OBJE
  ---------------|--------
ASPT  TENSE  PARM       RESTR
  |     |      |      ____|____
  _  PRESENT  !X1    RELN   ENTITY
              _|_     |        |
             P1 P2 YES-ADV-1   !X1
```

Figure 9: Forest drawn by the user and reformatted by the system

Variables As illustrated in Figure 9, it is possible to designate a subforest by a variable name. Thus the set of forests we work with contains graphs. In fact, it contains rational trees, as [Colmerauer et al. 83] defined them, and direct acyclic graphs, as defined in [Shieber 86].

Figure 10 gives an example of a rational tree. It gives an idealised representation of the sentence *The man who saw the man who saw the man* ... This representation is not linguistically valid because, from a formal point of view, the index `i` of the trace (`Wh`, `e`) should be incremented for each new clause.

```
        !x_i ...............
          |                :
          NP               :
 ---------|----            :
DET  N        S            :
 |   |   -----|--          :
the man Wh_i    S          :
         |   ---|_         :
        who NP   VP        :
             |  --|-       :
            e_i V !x_i ....:
                |
               saw
```

Figure 10: Example of a non-trivial rational tree

Figure 11 gives an example of a (non-connex) direct acyclic graph. It is a feature structure extracted from *Asura*'s current German generation grammar and simplified.

```
        syn                subcat      sem   control
 --------|_                ---|-----          --|--------
cat       agr            first    rest       syn       sem
  |        |             --|----              |         |
 vp       !x5.........  syn   sem            agr       !x1
    -------|------   : --|--   |       -------|------   :
 gender number person:cat agr !x1   gender number person:
    |      |      |  :  |  |   :       |      |      |  :
   !x2    !x3    !x4 : np !x5  :      !x2    !x3    !x4 :
    :      :      :  :.....:   :...... : .... : .... : .:
    :..... : .... : ...................:      :      :
           :..... : ..........................:      :
                  :..................................:
```

Figure 11: Example of a direct acyclic graph

1.3.2 Feature structures

This is not a true class. It is just a different input or output format for the general structure of forests. It was necessary for us to enable this format in order to use outputs directly from the NADINE system. The enablement, in turn, influenced our definition of atoms.

The transformation between a feature structure and a forest is just a matter of input/output for our system.

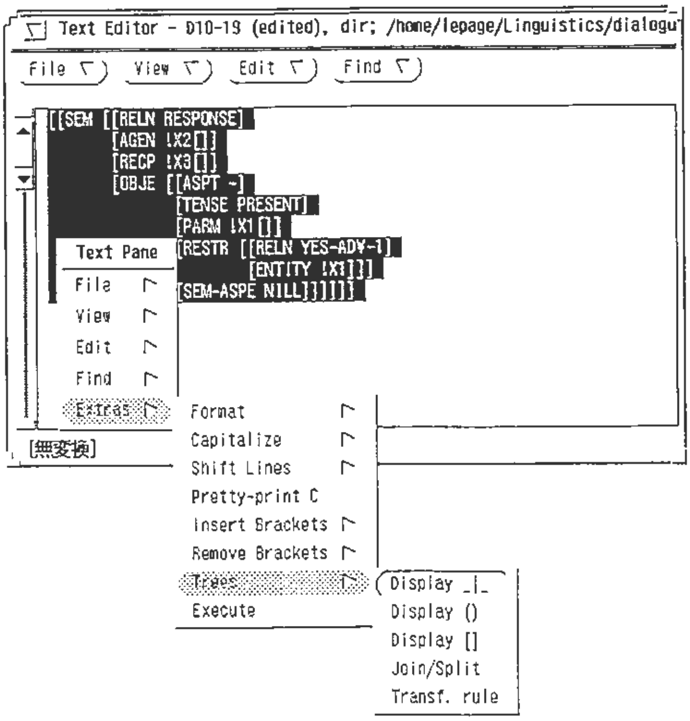
To program this transformation, we simply tell the input/output functions which class to use. This simple program (listed in Figure 12) is a few lines long, thanks to the grammar facility we have implemented [Lepage 92a], and to our object-oriented approach.

```
#include <stdio.h>
#include "gram.h"
#include "tree.h"
#include "fs.h"

void main(void)
{
   TREE *tree = NULL ;

   if ( scan("<F> ",&tree) )
      print("<D>\n",tree) ;
}
```

Figure 12: Program to read a tree and display it as a feature structure

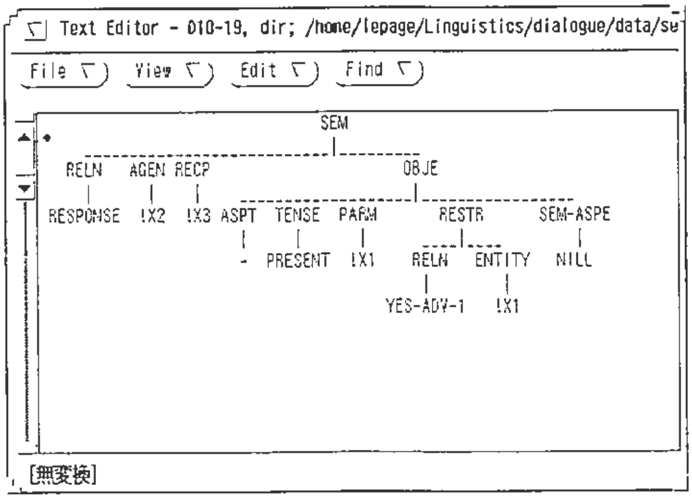
As an application of this easy programming, conversions between drawn trees, trees in parenthesised format and feature structures are available under two text editors: Text Editor under the OpenWindows system and Emacs.

Figure 13 shows how this facility works in the Text Editor under the OpenWindows environment. In this example, the user first selected the feature structure and then selected the command `Display _|_` in the Text Pane. The result will be a drawn representation of the feature structure (Figure 14). Similarly, the commands `Display ()` and `Display []` ask for a parenthesised tree and a feature structure respectively.



```
[[SEM [[RELN RESPONSE]
       [AGEN !X2[]]
       [RECP !X3[]]
       [OBJE [[ASPT -]
              [TENSE PRESENT]
              [PARM !X1[]]
              [RESTR [[RELN YES-ADV-1]
                      [ENTITY !X1]]]
              [SEM-ASPE NILL]]]]]]
```

Figure 13: Converting a feature structure into a tree


Text Editor - D10-19, dir; /home/lepage/Linguistics/dialogue/data/se
File
View
Edit
Find
[無変換]

```
                                 SEM
        ____________________________|_________
  RELN    AGEN RECP                            OBJE
   |       |    |          ___________________|_________________
RESPONSE  !X2  !X3 ASPT  TENSE  PARM        RESTR        SEM-ASPE
                    |      |      |        ____|____         |
                    -   PRESENT  !X1     RELN    ENTITY     NILL
                                          |         |
                                      YES-ADV-1    !X1
```

Figure 14: A feature structure viewed as a tree

1.3.3 Transformational rules

In order to manipulate the feature structures which are produced by the NADINE system and convert them into a form suitable for our work, we needed a tool to transform trees or forests. Extensive work has been done by GETA people on tree transformations ([Boitet 82], [Chauché 74], [Chauché 86], [Durand 88]). The present work is simpler by far.

A forest transformational rule consists of two parts. The left-hand side is the schema to be found in the object tree. Variables may be of three types:

- node variables begin with a colon;
- tree variables begin with a circumflex;
- and forest variables begin with a dollar sign.

Variables may occur several times in a schema. If so, they must get the same value at each different location.

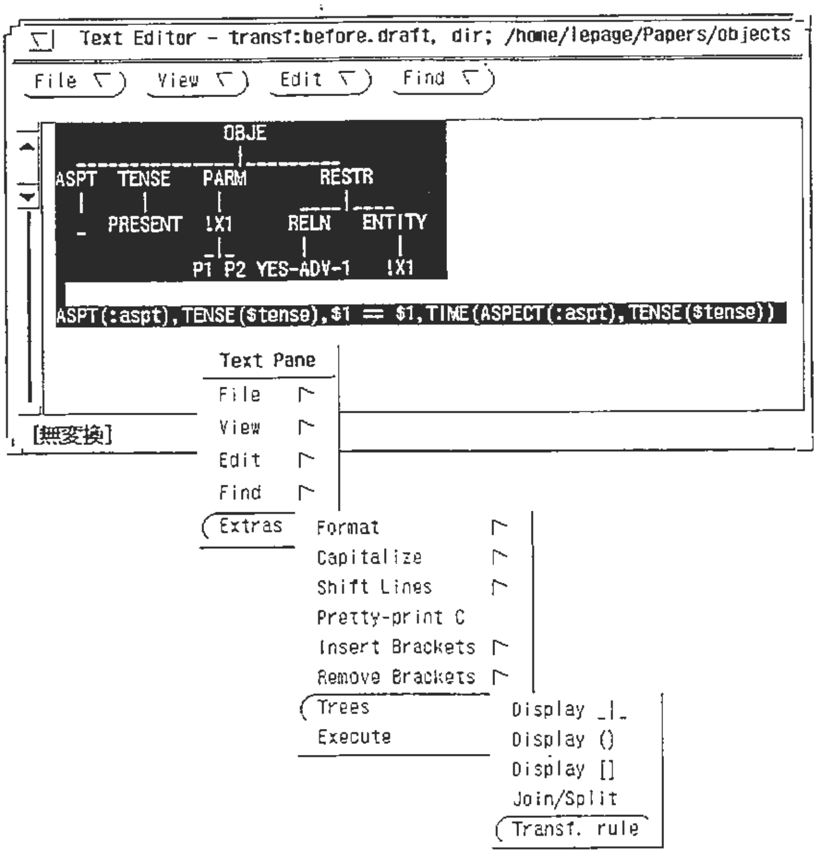
The right-hand side of the rule is the transformed schema. It is separated from the left-hand side by a double equal sign. Variables occurring in the left-hand side are replaced by the value they received during instantiation. New variables receive no value and appear in the resulting object as variables.

Figure 15 gives an example of a transformational rule. It says that the aspect/tense information has to be gathered under a TIME node. The schema says that ASPT, renamed ASPECT in the right hand side, dominates a node, and that TENSE dominates a forest. Moreover, the tense/aspect information is shifted to the right.

```
ASPT(:aspt),TENSE($tense),$1 ==
                    $1,TIME(ASPECT(:aspt),TENSE($tense))
```

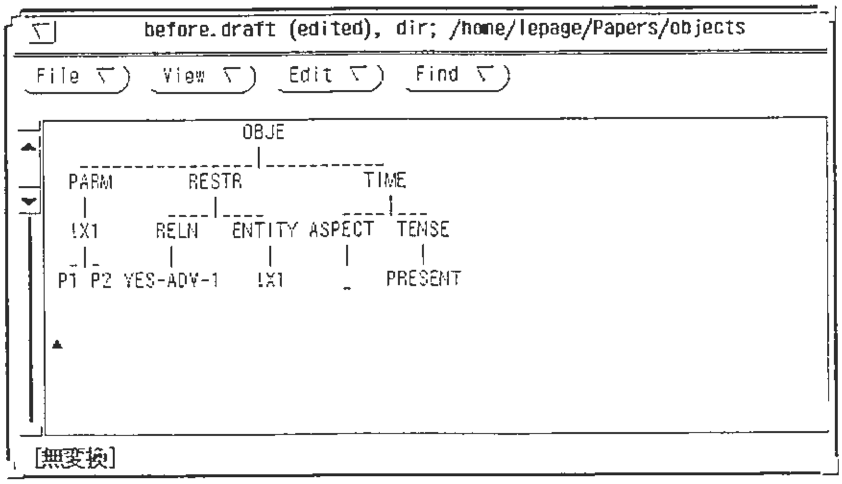
Figure 15: A transformational rule

Like the facilities described above, the forest transformational rule facility has been integrated under the Text Editor of the OpenWindows environment and under Emacs. Figure 16 and 17 show the application of the previous rule on the semantic tree structure of sentence 19 of ATR dialogue 10.



```
               OBJE
 _______________|_________
ASPT  TENSE    PARM        RESTR
 |      |       |       ____|____
 _   PRESENT   !X1     RELN    ENTITY
               _|_      |        |
              P1 P2 YES-ADV-1   !X1

ASPT(:aspt),TENSE($tense),$1 => $1,TIME(ASPECT(:aspt),TENSE($tense))
```

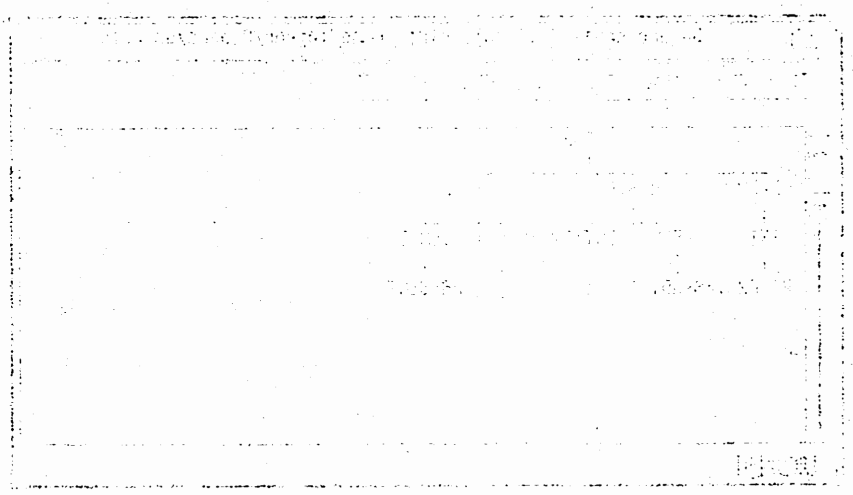
Figure 16: Applying a transformational rule to a tree



```
                    OBJE
 ___________________|___________
 PARM        RESTR              TIME
  |       ____|____         ____|___
 !X1     RELN    ENTITY ASPECT  TENSE
 _|_      |        |      |       |
P1 P2 YES-ADV-1   !X1     _    PRESENT
```

Figure 17: The tree after transformation by the rule

[illegible]



[illegible]

2 Basic functions on objects

Some basic functions are the same or only slightly different for all the objects. Following our object-oriented approach, we implemented them in a generic way using "false genericity". We explain here what false genericity is and present some of the basic functions.

2.1 Genericity

2.1.1 Principle

A function is said to be generic if it can be applied to a range of classes without considering the specificity of the classes.

2.1.2 False genericity

In real genericity, the class of the argument object is contained in the object. By retieving this information, a generic function can apply. Hence, the source code of the function is written once, and only one excutable program is compiled. In false genericity, as in the ADA programming language, the source code is written only once, but different executable programs are produced for different classes. Although obviously less clean from the theoretical point of view, this solution is more efficient in terms of execution time. Most arguments favouring C over C++ originate in this exact point.

2.1.3 Implementation

A generic function is implemented for a generic class `GEN`. The C preprocessor makes the necessary replacement to instantiate the function for a definite class. In a class module, the generic function is used via the `#include` directive.

Figure 18 shows the definition of the *creation* function as a generic function. The macro `fctGen` creates instantiation of function names for definite class names. The `trace(( ...))` directive is a trace of our own which can be simply set by a flag. It is for development purposes.

This function can then be used in a module through the directive `#include "generic/new.c"`.

```
#ifndef _new_c
#define _new_c

#ifndef _error_h
#include "error.h"
#endif

#define newGen fctGen(new,Gen)

GEN *newGen(void)
{
   GEN *result = NULL ;

trace(("in  "MODULE"::newGen()\n"))

   if ( ! (result = (GEN *) calloc(sizeof(void *),
                                   sizeof(GEN))) )
      error(MODULE,"new",ERR_MALLOC) ;

trace(("out  "MODULE"::newGen()\n"))

   return result ;
}

#endif /*!_new_c*/
```

Figure 18: A generic function

2.2 Some basic object functions

2.2.1 Creation

This function creates a new instance of the class. It reserves storage space and initialises it to NULL. Initialisation is carried out using the standard calloc function which initialises the reserved place (compare malloc which does not initialises the reserved space).

For each object, the creation function is called new suffixed by the name of the class. Hence for Atom, List, *etc.* we have newAtom, newList, *etc.*

The following piece of program shows the use of the newForest function in the copyElt function. After a new node is created, its members are assigned the member values of the function argument. The copyForest function produces a copy of a forest by application of copyElt on each node according to a preorder traversal.

```
static FOREST *copyElt(FOREST *forest)
{
   FOREST *result = NULL ;

   if ( forest )
   {
      result            = newForest() ;
      result->elt       = forest->elt ;
      result->daughter  = forest->daughter ;
      result->sister    = forest->sister ;
   } ;
   return result ;
}

FOREST *copyForest(FOREST *forest)
{
   return = preorder(forest,copyElt) ;
}
```

Figure 19: Use of the newForest function

2.2.2 Deletion

This function frees the space reserved for an object. It just applies the standard `free` function to the object.

As an example of use and definition of `free` functions, the deletion of a list is based on a recursive call to the `freeList` function. This is shown in the following piece of program.

```
/*
 * free a list
 */

void freeList(LIST *list)
{
   if ( list )
   {
      freeList(list->tail) ;
      free(list) ;
   } ;
}
```

Figure 20: Definition of the `freeList` function

2.2.3 Storage

For each object, a structure storing the various instances is created. It is an AVL tree. We have very briefly described this technique in [Lepage 92b]. The programs used for the dynamic programming technique and those for object storage are the same.

As an illustration, converting a character string to an atom is just adding the candidate string in the AVL tree (if the string was already there, its address is just returned). Limits checking and error handling constitute the main part of this function as shown below.

```
/*
 * convert string into atom
 */

ATOM *char2Atom(char *string)
{
   ATOM *result = NULL ;

   if ( string )
   {
      if ( strlen(string) > MAXINATOM )
         error(MODULE,"char2Atom",ERR_ATMLEN) ;
      if ( ! (result = (ATOM *) calloc(strlen(string)+1,
                                       sizeof(ATOM))) )
         error(MODULE,"char2Atom",ERR_MALLOC) ;
      result = strcpy(result,string) ;
      if ( ! (result = addinAvl(result,(AVLCMP) cmpAtom,
                                &atomAvl)) )
         error(MODULE,"char2Atom",ERR_ADDR) ;
   } ;
   return result ;
}
```

Figure 21: Use of the addinAvl function

2.2.4 Copy

This function creates copies of the objects. These copies are not stored in the object storage base, and consequently they are not seen by some functions which access objects through this storage base only. This behaviour is useful for handling temporary objects, for example during application of transformational rules.

The definition of the copyForest function has been given in Figure 19.

As an example of use, the algorithm for tree drawing makes use of a forest similar in structure but with larger information on the nodes (location in a matrix of characters, offset, *etc.*). Hence, at first, the tree is copied. After the tree has been drawn, this copy is deleted.

```
/*
 * draw a forest
 */

void drawForest(FILE *stream, FOREST *forest)
{
   FOREST *xforest = NULL ;

   initialise(stream) ;
   xforest = preorder(copyForest(forest),copyElt) ;
   xforest = preorder(preorder(postorder(xforest,
                                          frame),
                                reframe),
                       draw) ;
   freeForest(xforest) ;
}
```

Figure 22: Use of the copyForest function

2.2.5 Distance

This generic function implements a distance measurement for a variety of objects. It makes use of the recusive definition of these distances and of the dynamic programming facility we implemented [Lepage 92b].

On atoms and strings, this generic function calculates the Wagner and Fischer distance [Wagner & Fischer 74]. On forests, it calculates an extension of the Selkow distance [Selkow 77]. For discussion on the relation between the Wagner and Fischer string distance and the Selkow tree distance see [Lepage et al. 92].

Here are two semantic tres input of the German generation of the ASURA system for sentences 10 and 17 of ATR dialogue 1, whose structures are enough different. With the unit being a one character difference, the Selkow distance is 42.

```
*** TEST ON TYPE: FOREST ***

A 1st forest:
                    SEM
    -----------------|---------
  RELN   AGEN RECP           OBJE
    |      |    |    ---------|--
RESPONSE  !X2  !X3 PARM        RESTR
                     |       ----|-----
                    !X1     RELN    ENTITY
                              |        |
                        NEIN-INTERJ   !X1
A 2nd forest:
                        SEM
    ---------------------|---------
  RELN   AGEN RECP               OBJE
    |      |    |       -----------|---------------
RESPONSE  !X1  !X2    RELN        ASPT  TENSE  SEM-ASPE
                        |           |     |        |
               ICH_VERSTEHE-IDIOM STAT PRESENT  STATIVE

Selkow distance : 42
```

Figure 23: Distance between two semantic trees

In our work, a pair, a string and a tree, constitutes a basic object called a *board*. The distance between two boards is defined as the sum of the distances between the strings and the trees. Hence the following piece of program implements the distance between two boards.

```
/*
 * distance between two boards
 */

int distBoard(BOARD *board1, BOARD *board2)
{
   return   distForest(board1->forest,board2->forest)
          + distList(board1->list,board2->list) ;
}
```

Figure 24: Definition of the `distBoard` function

2.2.6 Unification

Here, *unification* means the unification found in the Prolog programming language [Colmerauer et al. 83]. This function is available for atoms, lists, strings and forests.

As was mentioned earlier (page 16), rational trees can be described in our implementation. Moreover, they can be used as arguments of unification. This is possible thanks to the dynamic programming facility [Lepage 92b]. As a demonstration, the following unification yields an infinite rational tree. In the result, both arguments are reexpressed after unification has been performed. The first appearance of a variable is separated from its value by a space. Hence, the first argument sets the following equation on trees: `$x1=a($x1)`. Its unique solution is the infinite rational tree `a(a(a(a(...))))`.

```
*** TEST ON TYPE: FOREST ***

A 1st forest: a($x1);
A 2nd forest: $x1;

unif( $x1 a($x1) , $x1 ) = TRUE
```

Figure 25: Unification of forests

Conclusion

This document has briefly presented the various objects we have designed for our study of distances between, and pattern-matching on, linguistic objects. Various classes of objects have been defined. The important classes can be divided into two subsets.

The first subset is concerned with string types. It includes lists, sets, and stacks. Of course, the aim is to facilitate the handling of natural language strings.

The second subset deals with tree structures. It includes forests, feature structures, and tree transformational rules. It was designed to handle linguistic representations. We also handle the feature structures output by the NADINE system, and transform them as desired. Visualisation in a user-friendly drawn format is possible.

Programming the functions associated with these classes required the implementation of various programming facilities. In particular, this document has described a false genericity facility. Since it produces several executable codes from one source code, execution time is not negatively affected. Other programming facilities necessary for the implementation of the abovementioned class functions, such as the grammar facility and the dynamic programming facility, have been described in other documents.

References

[Boitet 82] Christian Boitet, rédacteur
Le point sur Ariane-78, début 1982
(Volume 1, Partie 1 : le logiciel)
Convention ADI no 81/423 Cap Sogeti Logiciel – GETA-Champollion, Grenoble, avril 1982.

[Chauché 74] Jacques Chauché
Transducteurs et arborescences.
Etude et réalisation de systèmes appliqués aux grammaires transformationnelles
Thèse d'Etat, Université de Grenoble I, décembre 1974.

[Chauché 86] Jacques Chauché
Déduction automatique et systèmes transformationnels
Proceedings of COLING-86, pp 408-411, published by IKS, Bonn, 1986.

[Colmerauer et al. 83] Alain Colmerauer, Henry Kanoui et Michel Van Caneghem
Prolog, bases théoriques et développements actuels
Techniques et Sciences Informatiques, vol. 2, no 4, pp 271-311, 1983.

[Dewhurst & Stark 89] Stephen C. Dewhurst and Kathy T. Stark
Programming in C++
Prentice Hall, Inc, 1989.

[Durand 88] Jean-Claude Durand
TTEDIT Un éditeur transformationnel d'arbres
Thèse, Université Joseph Fourier, mars 1988.

[Kernighan & Ritchie 88] Brian W. Kernighan and Dennis M. Ritchie
The C programming language, 2nd Ed.
Prentice Hall, Inc, 1988.

[Lepage et al. 92] Yves Lepage, Furuse Osamu and Iida Hitoshi
Relation between a pattern-matching operation and a distance: On the path to reconcile two approaches in Natural Language Proceessing
Proceedings of the First Singapore International Conference

on Intelligent Systems, Singapore, November 1992, pp. 513-518.

[Lepage 92a] Yves Lepage
Easier C programming
Input/output facilities
ATR report TR-I-0293, Kyoto, November 1992.

[Lepage 92b] Yves Lepage
Easier C programming
Dynamic programming
ATR report TR-I-0295, Kyoto, November 1992.

[Selkow 77] Stanley M. Selkow
The Tree-to-Tree Editing Problem
Information Processing Letters, Vol. 6, No. 6, December 1977, pp. 184-186.

[Shieber 86] Stuart M. Shieber
An Introduction to Unification-Based Approaches to Grammar
CSLI Lectures Notes no 4, Leland Stanford Junior University, 1986.

[Wagner & Fischer 74] Robert A. Wagner and Michael J. Fischer
The String-to-String Correction Problem
Journal for the Association of Computing Machinery, Vol. 21, No. 1, January 1974, pp. 168-173.

Index