TR-I-0293

Easier C programming Input/output facilities

Yves Lepage

November 1991

Abstract

This report describes the input/output facilities developed for our studies of distances between linguistic objects. We propose two general functions, similar to the standard C library functions `fscanf` and `fprintf`. For input, description of an object syntax is possible, thanks to a set of macros which allow one to transform a BNF description into a fragment of C code. For interpretation, a backtracking mechanism has been implemented.

Keywords

Programming in C, input, output, BNF, backtracking mechanism.

Contents

Contents

List of Figures

Introduction

This report describes part of the implementation work done in the frame of a more general study. The aim of the general study is to implement basic objects and functions to allow experiments with pattern matching and distance calculation on trees and strings.

The report describes the basic input/output functions which have been implemented. They allow one to program more quickly the various read/write functions needed for each class of objects. The input facility accepts grammar descriptions in a form equivalent to that of BNF grammars.

The advantages of such facilities are:

- the simplicity of use of the input/output functions. The burden of calling different read/write functions for each different object has been eliminated by proposing general functions which respect the spirit of the standard C library functions;
- direct transcription of BNF grammars, which are are simple and immediately understandable;
- elimination of the burden of writing small *ad hoc* parsers for each new small grammar. Writing such small parsers often hinders quick design of new objects;
- no increase in the number of code pieces. Using such programs as YACC [Johnson 79] or LEX [Lesk & Schmidt 79] (or even GA [Lepage 88]!) unnecessarily increases the number of files for an application;
- compatibility between the relative simplicity of input format for basic objects and the simplicity of the facility. The complexity of the LEX and YACC programming languages is not justified for the facilities proposed here;
- drastic reduction in development time stemming from simplicity of use.

In the following, we first describe the input/output functions proposed, and then the input facility for grammar descriptions. This latter is illustrated using examples. Its backtracking mechanism is detailed and the macros for it are defined.

1 Input/output facilities

In this section, we describe the programming work intended to facilitate the input and output of the objects we manipulate. First, we recall the functions which the C programming language offers, and then we show how we programmed similar general functions.

1.1 The predefined C functions

The usual functions for input and output in C (see for reference [Kernighan & Ritchie 88, pages 243-246]) are `fscanf` and `fprintf`, and their variants `scanf`, `sscanf`, ... The two first functions have the general format:

```
function(FILE *stream, const char *format, ...)
```

where `stream` is the stream from which to read or on which to write, `format` is the format string, and the remaining parameters are the C variables read or written.

The format string contains two types of objects: ordinary characters, which are copied to the output stream or expected to match exactly the input stream, and conversion specifications. Conversion specifications begin with the character % and end with a conversion character. Between the two, one can define adjustment, field length, precision, and so on. We are only interested in the simplest form: % followed by a conversion character.

1.2 Extending the C functions

We propose to extend (and limit) the previous functions in the following way. The new input and output functions are called `fscan` and `fprint` and their variants for the standard streams `stdin` and `stdout` are called `scan` and `print`. They will accept the simplest form of the conversion specifications of `fscanf` and `fprintf`, but, in addition, they will also accept conversion specifications in the syntax < a > where a is a predefined class designator.

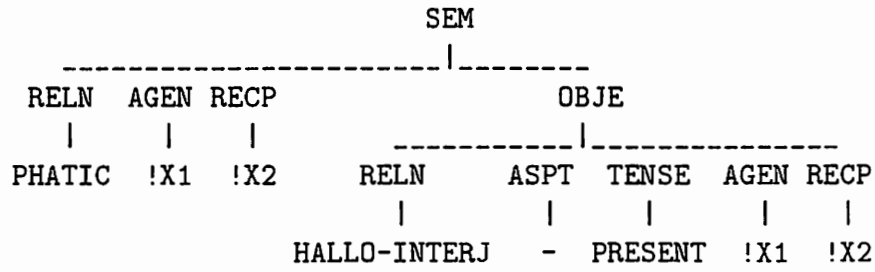
1.2.1 Class designators

For each object class created for our application, we have defined a class designator. Here is the list of the class designators defined so far (see also [Lepage 92b] for a description of these classes:

- `b` for booleans (class `BOOLEAN`). Two printed values are possible: `TRUE` and `FALSE`;
- `a` for atoms (class `ATOM`). `a`, `12`, `HEARER-SIDE`, `+` are atoms;
- `L` for lists (class `LIST`), *e.g.* `(a,b,c)`;
- `S` for sets (class `SET`), *e.g.* `{a,b,c}`;
- `A` for strings (not C strings, but strings of class `STR`), *e.g.* `"Is this the conference office '?'"` where `Is`, ..., `office`, `'?'` are atoms.
- `T` for trees (of class `TREE`) in the parenthesised form, *e.g.*

```
SEM(RELN(PHATIC)
   ,AGEN(!X1 )
   ,RECP(!X2 )
   ,OBJE(RELN(HALLO-INTERJ)
        ,ASPT(-)
        ,TENSE(PRESENT)
        ,AGEN(!X1)
        ,RECP(!X2)))
```

- `D` for trees in the drawn form, *e.g.*



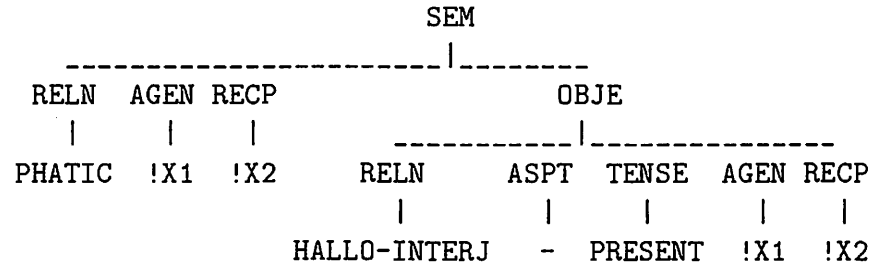
- `R` for tree transformational rules (represented with class `TREE`), *e.g.*

```
:root(RELN(:reln),$1) == :reln($1)
```

- F for feature structures (same class as trees, TREE), *e.g.*

```
[[SEM [[RELN PHATIC]
       [AGEN !X1[]]
       [RECP !X2[]]
       [OBJE [[RELN HALLO-INTERJ]
              [ASPT -]
              [TENSE PRESENT]
              [AGEN !X1]
              [RECP !X2]]]]]]
```

- B for boards (class BOARD), a pair consisting of a tree and a string. When displayed, the tree is drawn, *e.g.*




```
"Hello '.'"
```

- W for woods (class WOOD). For example: a(<b=c=d>) represents a wood with root a dominating b or c or d.

- O for objects, only in use in the predefined functions fscan and fprint.

As an example of use of the scan and print functions, consider the following program fragment. The user is asked to input a boolean, a set and a parenthesised tree. The program outputs them respectively in the form of a boolean, a list and a drawn tree. The output of the set as a list is possible because sets are constrained lists.

```
BOOLEAN boolean = FALSE ;
SET *set = NULL ;
```

```
TREE *tree = NULL ;

scan("<b>, <S>, <T>", &boolean, &set, &tree) ;
print("values: <b>,\n<L>,\n<D>",boolean,set,tree) ;
```

If the users types in the following line:

```
TRUE, {a, b}, a(b,c(d,e))
```

the output is:

```
values: TRUE,
(a, b),
  a
__|_
b  c
  _|_
  d e
```

This program fragment shows that the constant string format accepts the usual codes for line feed, tabulations, ... as in standard C.

1.2.2 The `fscan` and `fprint` functions

The `fscan` and `fprint` functions handle the format string in the following way. On a normal character, they just call standard C library functions; on a < character, an object of class `OBJECT` is created. It contains the type of the object to be read or written. This type is given by the class designator following the < character. The specialised functions for reading and writing objects of type `OBJECT`, i.e. `readObject` and `writeObject`, are then called.

These functions manage the choice of the specialised functions for the definite object class using a `switch` instruction on the type class.

1.2.3 The char2file function

No direct equivalent for sscanf and sprintf is provided. However, a general function is available which allows one to "transform" a string into a file. It is called char2file and takes a constant string as its first argument. It returns an opened stream.

With this function the equivalent of a call to sscanf is written in the following way:

```
fscan(char2file("foo"),"<a>",&atom) ;
```

Because the number of opened streams is limited, a safer programming style would be:

```
FILE *tmp = NULL ;
...
fscan(tmp = char2file("foo"),"<a>",&atom) ;
...
fclose(tmp) ;
```

1.2.4 The #include directive

In order to allow sharing of data from different files, the functions `fscan` and `scan` interpret the `#include` directive.

To illustrate this facility, suppose we have stored the displayed form of a tree in the file `tree.data` and a string in the file `string.data` (see figure 1 and figure 2).

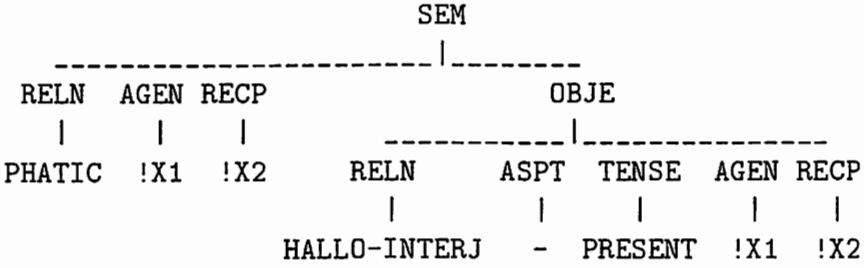


Figure 1: Content of the file `tree.data`

```
"Hello '.'"
```

Figure 2: Content of the file `string.data`

The file `board.data` may contain references to these two files (see figure 3).

```
#include "tree.data"
#include "string.data"
```

Figure 3: Content of the file `board.data`

In a C program, the set of instructions

```
FILE *stream = NULL;
BOARD *board = NULL;
...
stream = fopen("board.data","r");
fscanf(stream,"<B>",&board);
```

will have the expected effect. It will assign the tree contained in the file `tree.data` to the tree part of the variable `board` and the string contained in the file `string.data` to its string member.

2 The grammar facility

A set of macros, using predefined functions which implement a backtracking mechanism, allows the description of grammars along with the description of actions written in C code. This section describes this facility. The backtracking mechanism will be described in the next section.

2.1 An example

The input function `readList` for the class `list` describes the syntax a list has to follow and specifies the actions to be performed for the actual creation of the list.

A list is noted as a sequence of elements separated by commas. The whole sequence itself is enclosed in parentheses. Using the BNF notation, this definition can be noted as follows:

$$<list>::=([<element>,]^* <element>)$$

It would be convenient to automatically replace the BNF symbols (::=, [,],]*, *etc.*) by predefined C-code pieces. Unfortunately, this is not possible, because the ANSI-C preprocessor allows the directive `#define` to be applied only on identifiers.

Thus it is necessary to transcribe the BNF notation into an equivalent one. The grammar facility proposes the following transcriptions:

::= ...	is rewritten as	`rule ...end_rule ;`
...|...		`rhs ...end_rhs rhs ...end_rhs`
[...]		`ruleOptn ...end_ruleOptn`
[...]*		`ruleStar ...end_ruleStar`

Each non-terminal or terminal has to be replaced by the adequate function for its reading. These calls are separated by `&&`. The transcription of the grammar for list gives the following program fragment:

```
rule
   rhs fscan(stream," (") &&
       ruleOptn
          rhs
             ruleStar
                rhs readElt(stream,&elt) &&
                    fscan(stream," ,")
                end_rhs
             end_ruleStar &&
             readElt(stream,&elt)
          end_rhs
       end_ruleOptn &&
       fscan(stream," )")
   end_rhs
end_rule ;
```

The `rule ...end_rule ;` expression yields a boolean value. A `TRUE` value means that the input has been read correctly according to the grammar. The next reading operation will start after the portion of input read. A `FALSE` value means that the attempt to read on the input according to the grammar has failed. The input pointer is backtracked to the position it had before the attempt, and the next reading attempt will start from there. This backtrack mechanism will be described below.

For the completion of our example, C code has to be inserted in the previous code in order to actually create the objects. A function `addtoList`, which adds an element to a possibly empty list, suffices for this.

C code can be added only after the `end_rhs` key-word and must respect the syntax of a C statement. Using this possiblity, the input of a list can be completely programmed. It is as follows:

```
BOOLEAN readList(FILE *stream, LIST **list)
{
   BOOLEAN result = FALSE ;
   ELT *elt = NULL ;

   *list = NULL ;
   result = rule
      rhs fscan(stream," (") &&
          ruleOptn
             rhs
                ruleStar
                   rhs readElt(stream,&elt) &&
                       fscan(stream," ,")
                   end_rhs
                      *list = addtoList(elt,*list) ;
                end_ruleStar &&
                readElt(stream,&elt)
             end_rhs
                *list = addtoList(elt,*list) ;
          end_ruleOptn &&
          fscan(stream," )")
      end_rhs
   end_rule ;

   return result ;
}
```

2.2 The backtracking mechanism

We now describe the backtracking mechanism implemented for the grammar facility. Its source code can be found in the `gram` module.

2.2.1 The global variables

The backtracking mechanism requires the memorisation of the characters of the input stream in a `buffer` so that they can be reused several times. It is also necessary to stack the backtracking points in a `stack`. At any time in the computation, one can handle the stack if one knows the position of the `next` character to be read, and the `last` character in the `buffer`. In addition, the `top` variable gives the last backtracking point pushed onto the `stack`.

2.2.2 The raz_align function

This function resets the values of all the global variables. It has been implemented for consistency but it is not used in the grammar facility.

```
void raz_align(void)
{
   buffer[0] = 0 ;
   stack[0] = 0 ;
   next = 0 ;
   last = -1 ;
   top = -1 ;
}
```

2.2.3 The set_align function

This function is called when a new backtracking point is created. It corresponds to the `rule` keyword in the grammar facility.

This function basically adds the position of the next character to be read as a new backtracking point in the stack. Of course, this action is impossible if the stack is full. In this case, the stack is cleaned up by eliminating all the position values which are no longer relevant. These positions are recognised because they have become negative during execution (see the `shift` function). If this cleaning of the stack cannot be completed, an error occurs.

```
void set_align(void)
{
   int shift = 0, i = 0 ;

   if ( ++top < MAX )
      stack[top] = next ;
   else
   {
      for ( shift = 0 ;
            (shift < MAX) && (stack[shift] < 0) ;
             shift++ ) ;
      if ( shift > 0 )
      {
         for ( i = shift ; i < top ; i++ )
            stack[i-shift] = stack[i] ;
         top -= shift ;
         stack[top] = next ;
      }
      else
         error(MODULE,"set_align",ERR_STACKOVF) ;
   } ;
}
```

2.2.4 The rm_align function

This function is called when the input stream has been successfully analysed according to a rule description using the grammar facility. It corresponds to the end_rule keyword, and is also the last action to be taken by end_ruleOptn and end_ruleStar.

This function removes the last backtracking point from the stack. For its successful operation, of course, the stack must not be empty.

```
void rm_align(void)
{
   if ( top < 0 )
      error(MODULE,"rm_align",ERR_STACKEMPTY) ;
   else
      top-- ;
}
```

2.2.5 The align function

This function is called when a new right-hand side is tried. It corresponds to the `rhs` keyword of the grammar facility, which introduces alternative right-hand sides in rules. As `ruleOptn` and `ruleStar` are factored alternatives, they also call this function.

This function performs backtracking. The position of the next character to be read on the buffer is given by the top of the backtracking stack. This function is considered as a function reading an empty string on the input stream. This is why:

- it returns the boolean value `TRUE` because it always succeeds.
- the backtracking point is not removed from the stack; if removed, it would only be popped from and immediately pushed back on the stack.

```
BOOLEAN align(void)
{
   if ( stack[top] < 0 )
      error(MODULE,"align",ERR_BACKTRACK) ;
   else
      next = stack[top] ;
   return TRUE ;
}
```

2.2.6 The fget function

The communication between the stream and the buffer is established through the `fget` function. This function is used in the basic functions of the `char` module for skipping blanks, comments, reading until a character or a word, reading a word, an integer, kanji strings, and so on.

This function reads one character from the buffer. If there are no more characters in the buffer, the next character on the stream is read using a call to the C-library function `fgetc`, and is moved into the buffer. This function may call the `shift` function, described below, if no more space is available in the buffer.

```
int fget(FILE *stream)
{
   if ( next > last )
   {
      if ( ! (last+1 < MAX) )
         shift() ;
      buffer[last+1] = fgetc(stream) ;
      next = ++last ;
   } ;
   return buffer[next++] ;
}
```

2.2.7 The shift function

This function is called when the buffer is full. It creates space in the buffer by shifting all characters in the buffer by a certain offset SHIFT. The values of the next character to be read and of the last character in the buffer have to be modified appropriately. Also, the values in the stack, representing the backtracking points on the buffer, have to be decremented by the value of the offset. They may become negative, which means they have no more relevant values.

```
static void shift(void)
{
   int i = 0 ;

   for ( i = SHIFT ; i < MAX ; i++ )
      buffer[i-SHIFT] = buffer[i] ;
   next -= SHIFT ;
   last -= SHIFT ;
   for ( i = 0 ; i < MAX ; i++ )
      stack[i] -= SHIFT ;
}
```

2.3 The grammar facility macros

2.3.1 General principle

The keywords defined for the grammar facilities are grouped by pairs. For each opening keyword, there is a closing keyword.

The keyword pairs containing the word `rule` define boolean values. A `TRUE` value means that the input has been correctly analysed by the grammar section enclosed between the beginning and ending keywords. An opening keyword containing `rule` opens a condition, and its corresponding closing keyword ends a statement.

A `rhs` keyword pair is supposed to be placed in the middle of a `rule` keyword pair. Consequently, the `rhs` keyword ends a statement and opens an `else if` condition, whereas `end_rhs` closes a condition and opens a statement.

The C syntax allows one to enclose a sequence of statements in curly brackets to form a compound statement. Moreover, a statement enclosed in parentheses returns a value which is the value of the last statement of the sequence. The following macros make extensive use of these possibilities.

2.3.2 The rule and end_rule macros

Before a rule is invoked, a backtracking point is memorised by calling the set_align function. The exploration of the different right-hand sides is done as a sequence of if ...else if. At the end, the backtracking point is removed. If none of the alternatives applied, the backtracking point is also removed, but the current function is exited by returning FALSE (recall that a rule ...end_rule sequence returns a boolean value).

The definition of the rule and end_rule macros is as follows:

```
#define rule        ({                          \
                        set_align() ;           \
                        if (FALSE)              \
                        {

#define end_rule        }                       \
                        else                    \
                        {                       \
                           rm_align() ;         \
                           return FALSE ;       \
                        } ;                     \
                        rm_align() ;            \
                        TRUE ;                  \
                    })
```

2.3.3 The rhs and end_rhs macros

Before trying a new alternative, backtracking is performed by calling the `align` function. The result of reading a right-hand side is a boolean value returned to an `if` instruction. If it is true, i.e. the input has been correctly read according to the right-hand side, the actions can be executed.

The interpretation of the `rhs` and `end_rhs` macros is as follows:

```
#define rhs            } else if ( align() &&

#define end_rhs        ) {
```

2.3.4 The ruleOptn and end_ruleOptn macros

Before exploring an optional sequence in the grammar, a backtracking point has to be memorised. The ruleOptn is thus the same as a rule keyword. But leaving a ruleOptn section with end_ruleOptn is slightly different from a end_rule keyword. An eventual failure must not be taken into consideration, because the grammar section is optional. Thus, in any case, a TRUE value must be returned.

Consequently, the interpretation of the ruleOptn and end_ruleOptn macros is as follows:

```
#define ruleOptn        rule

#define end_ruleOptn        }                          \
                            else if ( align() ) {} ;   \
                            rm_align() ;               \
                            TRUE ;                     \
                        })
```

2.3.5 The ruleStar and end_ruleStar macros

The ruleStar keyword invokes a while loop. So the same grammar section and actions are executed until the input cannot be analysed by the grammar section. Then, the end_ruleStar keyword has to remove the backtracking point and return TRUE in any case.

The definition of the ruleStar and end_ruleStar macros is thus as follows:

```
#define ruleStar        ({ while ( rule

#define end_ruleStar            }                       \
                                else if ( align() )     \
                                {                       \
                                   break ;              \
                                } ;                     \
                                rm_align() ;            \
                                TRUE ;                  \
                           }) ) ;                       \
                           TRUE ;                       \
                      })
```

2.3.6 The generated code

We illustrate the previous definitions of the grammar facility macros using an example. Consider the following grammar fragment,

```
begin[one|two]end
```

where either one or two must be read.

Suppose we want certain variables to be assigned values according to the integer read. This can be translated in the following program fragment:

```
return
  rule
     rhs scan(" begin") &&
        rule
           rhs scan(" one") end_rhs
              i = 1 ;
           rhs scan(" two") end_rhs
              { i = 2 ; j = 0 ; }
        end_rule &&
        scan(" end")
     end_rhs
        { printf("i = %d\n",i) ; }
  end_rule ;
```

The code generated by the application of the macros is as follows:

```
   return
   ({ set_align() ;
      if (FALSE) { }
      else if ( align() && scan(" begin") &&
          ({ set_align() ; if (FALSE) { }
             else if ( align() && scan(" one") )
                { i = 1 ; }
             else if ( align() && scan(" two") )
                { { i = 2 ; j = 0 ; } }
             else
                { rm_align() ; return FALSE ; } ;
             rm_align() ; TRUE ;
          }) &&
```

```
        scan(" end") )
    )
       { printf("i = %d\n",i) ; }
    else { rm_align() ; return FALSE ; } ;
    rm_align() ; TRUE ;
}) ;
```

The example code demonstates that a rule returns a boolean function, and that the application of alternatives is realised by a sequence of `if ... else if` instructions. At each of these `if` or `else` points, backtracking functions are called.

Conclusion

This report has described general functions which allow objects to be input and output more easily than with standard C library functions. It has also described a grammar facility for the rapid description of input formats for objects.

These facilities have been used to implement the read/write functions of the objects we work on: atoms, lists, strings, sets, trees (or forests), woods.

They have proved to be simple to use and of great utility for the rapid implementation of new objects, as they make overly specific functions unnecessary and eliminate the burden of programming small specialised parsers for simple input formats.

References

[Johnson 79] Stephen C. Johnson
YACC Yet Another Compiler-Compiler
UNIX Programmer's Manual, Seventh Edition, Volume 2B
Bell Telephone Laboratories Murray Hill, New-Jersey, January 1979.

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

[Lepage 88] Yves Lepage
GA Un générateur d'analyseurs (version 2.1)
Document interne GETA, Grenoble, janvier 1988.

[Lepage 92b] Yves Lepage
Easier C programming
Some useful objects
ATR report TR-I-0294, Kyoto, November 1992.

[Lesk & Schmidt 79] M.E. Lesk and E. Schmidt
LEX A Lexical Analyzer Generator
UNIX Programmer's Manual, Seventh Edition, Volume 2B
Bell Telephone Laboratories Murray Hill, New-Jersey, January 1979.

Index