1. NEW LIBRARY DESIGN

This chapter is dedicated to the description of the new library architecture which is composed of four main levels: the non-specific low-level tools, the specific low-level tools, the lexical level and the syntactic level. The contents of each of these levels are described in the following subsections. Since the complete *API* documentation can be found in the tool distribution, only relevant changes in relation to the former architecture are mentioned here. Let also say that the new library has been renamed to `SlpTK`, instead of `SlpToolKit`. Schematic overview of the proposed design is shown in figure 1, with the four stated levels and their corresponding modules.

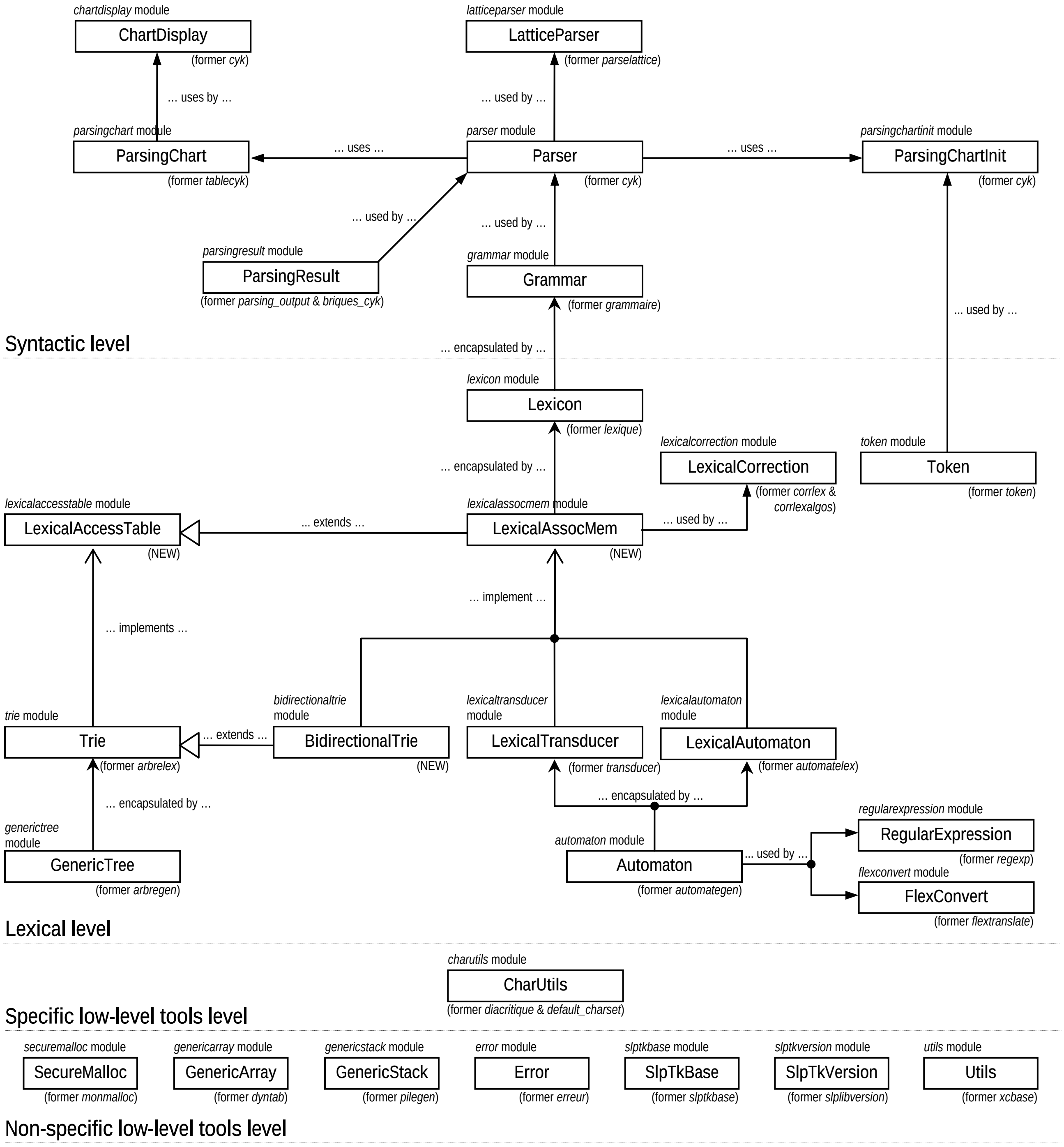

figure 1 - Schematic representation of the new software architecture

1.1 Low-level modules

This chapter gathers the description of the so-called low-level modules, which are general tools widely used inside the library. Many of these modules are not domain specific and may be used by other applications.

1.1.1. securemalloc module

`securemalloc` is a redesigned version of former **`monmalloc`** module. It allows to dynamically allocate memory in a more secure way than the functions offered by the standard **`malloc`** header.

List of exported functions:

```
/* Former monmalloc */
void* secureMalloc(size_t);

/* Former moncalloc */
void* secureCalloc(size_t, size_t);

/* Former monrealloc */
void* secureRealloc(void*, size_t);

/* Former monfree */
void secureXFree(void**);
```

1.1.2. genericarray module

`genericarray` is a redesigned version of former **`dyntab`** module which offers dynamic arrays for several basic types. *GLib* library proposes such structure with the **`GArray`** type. Unfortunatly, one must cast its **`data`** field to the appropriate datatype if the stored elements are wider than a byte. To avoid this, **`GArray`** is encapsulated inside a structure (***`type`*`Array`**) with a well-typed pointer (**`elements`**) that is equal to the array **`data`** field.

```
typedef struct {
    GArray* array;              /* Encapsulated GArray */
    type*   elements;           /* Former tab - equals array->data */
} typeArray;                    /* Former Dyntab_type */

typedef struct …  CharArray;           /* Former Dyntab_char */
typedef struct …  IntArray;            /* Former Dyntab_int */
typedef struct …  LongArray;           /* Former Dyntab_longint */
typedef struct …  DoubleArray;         /* Former Dyntab_double */
typedef struct …  PointerArray;        /* Former Dyntab_pointeur */
typedef struct …  StringArray;         /* NEW – Used by Grammar */

/* Former Augmente_Dyntab_type */
void typeArrayAppendVal(typeArray* array, const type value);

/* Former Ajoute_Dyntab_type */
void typeArraySetVal(typeArray* array, const size_t index, const type value);

/* Former Ajoute_Dyntab_type */
typeArray* typeArrayDuplicate(const typeArray* array);

/* Former Init_Dyntab_type */
typeArray* typeArrayCreate();

/* Former Libere_Dyntab */
#define arrayFree(stack) …

/* NEW – StringArray type needs a dedicated destructor */
void stringArrayFree(StringArray*);
```

Deleted function:

`Libere_Dyntab_pointer` – Array elements can reference memory blocks allocated by **`secureMalloc`** or by *GLib* allocators (for example **`g_string_new`**). Thus, we initially don't know which function must be called to release the memory blocks referenced by the elements of any **`PointerArray`**.

1.1.3. genericstack module

genericstack is a redesigned version of former pilegen module which offers stacks for several basic types. As pilegen was based on the former dyntab module, genericstack can be also considered as an interface to new genericarray module.

```
typedef CharArray    CharStack;    /* Former Pile_char */
typedef IntArray     IntStack;     /* Former Pile_int */
typedef LongArray    LongStack;    /* Former Pile_longint */
typedef DoubleArray  DoubleStack;  /* Former Pile_double */
typedef PointerArray PointerStack; /* Former Pile_pointeur */

/* Former Empile_type */
void typeStackAdd(typeStack* stack, type value);

/* Former Depile_type */
type typeStackRemove(typeStack* stack);

/* Former Init_Pile_type */
typeStack* typeStackCreate();

/* Former Vide_pile */
#define stackEmpty(stack)          …

/* Former Pile_Top */
#define stackGetTopValue(stack)    …

/* Former Pile_NonVide */
#define stackIsNotEmpty(stack)     …

/* Former Pile_Vide */
#define stackIsEmpty(stack)        …

/* Former Libere_Pile */
#define stackFree(stack)           …
```

Deleted function:

Libere_Pile_pointeur_and_elements – Remark concerning Libere_Dyntab_pointer (section 1.1.2 page 2) also applicable.

1.1.4. error, slptkbase, slptkversion and utils modules

error is a redesigned version of the former erreur module that provides a standard way to print error messages. Please have a look to section Erreur ! Source du renvoi introuvable. (page Erreur ! Signet non défini.) for more information.

Exported function:

```
/* Former Affiche_Message_Erreur */
void printErrorMessage(const int, const char*, ...);
```

slptkbase *header* file contains few declarations that vary from one supported platform to an other. Its content has almost not changed compared to the former architecture. slptkversion corresponds to the former slplibversion module. It declares only one function that prints the current library version.

Exported function:

```
/* Former SlpLibVersion */
const char* getSlpTkVersion();
```

utils module, which can be considered as a continuation of former xcbase, gathers a set of general purpose low-level functions that are not especially related to natural language processing.

List of exported functions:

```
/* Former Clean_Spaces (from former XCString module) */
GString* g_string_CleanSpaces(GString*, int (*)(int));

/* Former Fichier_Importe_Une_Entree_Lexico (from former arbrelex module) */
int readFileEntry(FILE*, GString*);
```

1.1.5. charutils module

`charutils` is a redesigned version of former **`default_charset`** and **`diacritique`** modules. It gathers a set of low-level functions related to the character-class recognition. Although **`const unsigned int`** would be a more appropriate, the parameter of the functions prefixed by **`defaultIs`** stays an **`int`** for matching purpose with the standard **`isspace`** function.

List of exported functions:

```
/* Former Default_Separateur */
gboolean defaultIsDelimiter(int);

/* Former Default_Homogene */
gboolean defaultIsNeverDelimiter(int);

/* Former Default_Collable */
gboolean defaultIsGlueable(int);

/* Former majuscule */
gboolean oneIsUppercaseOfOther(const unsigned char,
                               const unsigned char);

/* Former diacritique */
gboolean oneIsDiacriticOfOther(const unsigned char,
                               const unsigned char);

/* Former diacritiquemajuscule */
gboolean oneIsUppercaseDiacriticOfOther(const unsigned char,
                                        const unsigned char);
```

1.2 Lexical modules

This chapter gathers the description of the modules dedicated to data structures and treatments related to the lexical level. One of the major contributions to the former lexical tools is the introduction of the access table and associative memory abstractions.

1.2.1. lexicalaccesstable module

This brand new module defines the data structure and the list of functions that lexical access table must implement. Such lexical memories allow to recover the index associated to a particular graphy (unidirectional). `LexicalAccessTableFunctions` structure stores a set of pointers to functions that can be classified into 6 categories:

- construction – constructor (`create`) and destructor (`free`);
- file input and output – loading (`load`, `import`) and saving (`save`, `export`) from or into files;
- search – search based on graphy (`searchFirst`, `searchNext`);
- insertion – insertion of new entries (`insert`, `getSize`);
- review – list the stored graphies (`dump`, `getNextEntry`, `goToFirstEntry`);
- graphy exploration – character by character content exploration (`getNextAvailableCharacter`, `goToRoot`, `goCharacterForward`, `goCharacterBackward`, …).

```
typedef enum {
    CHARACTER = 0,
    UNSIGNED_LONG
} LexicalDataType; /* NEW */

typedef void*         LexicalMemoryHandler; /* NEW */
typedef unsigned long InsertionResult;      /* NEW */
typedef unsigned long LexicalEntryIndex;    /* NEW */
typedef void*         LexicalEntry;         /* NEW */
typedef unsigned long LexicalCharacter;     /* NEW */

typedef struct {
                   LexicalMemoryHandler (*create)(const LexicalDataType);
                                     void (*free)(LexicalMemoryHandler);
                                    int (*export)(const LexicalMemoryHandler,
                                                  const char*);
                                    int (*import)(LexicalMemoryHandler,
                                                  const char*);
                                      int (*save)(const LexicalMemoryHandler,
                                                  const char*);
                                      int (*load)(LexicalMemoryHandler,
                                                  const char*);
                                      int (*dump)(const LexicalMemoryHandler,
                                                  int (*)(const char*, ... ));
                        InsertionResult (*insert)(LexicalMemoryHandler,
                                                  const LexicalEntry);
                                size_t (*getSize)(const LexicalMemoryHandler);
                  LexicalEntry (*getGraphyFromId)(const LexicalMemoryHandler,
                                                  const LexicalEntryIndex);
                     LexicalSearch (*searchFirst)(const LexicalMemoryHandler,
                                                  const LexicalEntry);
                           gboolean (*searchNext)(LexicalSearch*);
                     LexicalEntry (*getNextEntry)(LexicalMemoryHandler);
                           void (*goToFirstEntry)(LexicalMemoryHandler);
    LexicalCharacter (*getNextAvailableCharacter)(LexicalMemoryHandler);
                                 void (*goToRoot)(LexicalMemoryHandler);
                       void (*goCharacterForward)(LexicalMemoryHandler);
                      void (*goCharacterBackward)(LexicalMemoryHandler);
                      gboolean (*isAtEndOfGraphy)(const LexicalMemoryHandler);
                 LexicalEntryIndex (*getGraphyId)(const LexicalMemoryHandler);
                        size_t (*getGraphyLength)(const LexicalMemoryHandler);
                   LexicalDataType (*getDataType)(const LexicalMemoryHandler);
} LexicalAccessTableFunctions;
```

```
typedef enum {
    TREE,
    AUTOMATON,
    TRANSDUCER
} LexicalMemoryType; /* NEW */

typedef struct {
    LexicalMemoryHandler         data;
    LexicalMemoryType            type;
    LexicalAccessTableFunctions* functions;
} LexicalAccessTable;               /* NEW */

typedef struct {
    LexicalMemoryHandler source;    /* NEW - Used by searchNext */
    gboolean          found;        /* Former Retour_recherche.trouve */
    short             found_length; /* Former Retour_recherche.indice_chaine */
    LexicalEntryIndex key;          /* Former Retour_recherche_lex.valeur */
    TrieSearchInfo    specific;     /* todo */
    LexicalEntryIndex identifier;   /* Former Retour_recherche.indice_valeur */
} LexicalSearch;                    /* Former Retour_recherche_valeur_lexico */
```

List of exported functions:

```
void         lexicalEntryToString(LexicalEntry, const LexicalDataType, GString*);
void         lexicalEntryFree(LexicalEntry, const LexicalDataType)
LexicalEntry lexicalEntryFromString(const char*, const LexicalDataType);

void              LATCreate(LexicalAccessTable*, const LexicalDataType);
int               LATExport(const LexicalAccessTable*, const char*);
void              LATFree(LexicalAccessTable*);
int               LATImport(LexicalAccessTable*, const char*);
int               LATLoad(LexicalAccessTable*, const char*);
int               LATSave(const LexicalAccessTable*, const char *);
int               LATDump(const LexicalAccessTable*, int(*)(const char* ,...));
InsertionResult   LATInsert(LexicalAccessTable*, const LexicalEntry);
size_t            LATGetSize(const LexicalAccessTable*);
LexicalSearch     LATSearchFirst(const LexicalAccessTable*, const LexicalEntry);
gboolean          LATSearchNext(const LexicalAccessTable*, LexicalSearch*);
LexicalCharacter  LATGetNextAvailableCharacter(LexicalAccessTable*);
void              LATGoToRoot(LexicalAccessTable*);
void              LATGoCharacterForward(LexicalAccessTable*);
gboolean          LATIsAtEndOfGraphy(const LexicalAccessTable*);
LexicalDataType   LATGetDataType(const LexicalAccessTable*);
LexicalEntry      LATGetGraphyFromId(const LexicalAccessTable*,
                                     const LexicalEntryIndex);
void              LATGoCharacterBackward(LexicalAccessTable*);
size_t            LATGetGraphyLength(const LexicalAccessTable*);
LexicalEntryIndex LATGetGraphyId(const LexicalAccessTable*);
void              LATGoToFirstEntry(LexicalAccessTable*);
LexicalEntry      LATGetNextEntry(LexicalAccessTable*);
```

Remark: `LAT` abbreviation stands for *Lexical Access Table*.

1.2.2. lexicalassocmem module

This brand new module defines the data structure and the list of functions that a lexical associative memory must implement. Such lexical memories allow to recover the index associated to a particular graphy, and the graphy associated to a particular index (bidirectional). In fact, a lexical associative memory is fundamentally a lexical access table which also provides the reverse access mechanism. Therefore, the `LexicalAssocMem` structure encapsulates a `LexicalAccessTable` and a pointer to the function (`access`) which supplies that reverse access. Moreover, all the exported functions (except `LAMAccess`) are equivalent to those exported by the `lexicalaccesstable` module.

```
typedef struct {
    LexicalAccessTable* access_table;
    LexicalEntry (*access)(const LexicalMemoryHandler, const LexicalEntryIndex);
} LexicalAssocMem; /* NEW */
```

List of exported functions:

```
void               LAMCreate(LexicalAssocMem*, const LexicalDataType);
int                LAMExport(const LexicalAssocMem*, const char*);
void               LAMFree(LexicalAssocMem*);
int                LAMImport(LexicalAssocMem*, const char*);
int                LAMLoad(LexicalAssocMem*, const char*);
int                LAMSave(const LexicalAssocMem*, const char*);
int                LAMDump(const LexicalAssocMem*, int(*)(const char* ,...));
InsertionResult    LAMInsert(LexicalAssocMem*, const LexicalEntry);
size_t             LAMGetSize(const LexicalAssocMem*);
LexicalSearch      LAMSearchFirst(const LexicalAssocMem*, const LexicalEntry);
gboolean           LAMSearchNext(const LexicalAssocMem*, LexicalSearch*);
LexicalCharacter   LAMGetNextAvailableCharacter(LexicalAssocMem*);
void               LAMGoToRoot(LexicalAssocMem*);
void               LAMGoCharacterForward(LexicalAssocMem*);
LexicalEntry       LAMAccess(LexicalAssocMem*, const LexicalEntryIndex);
LexicalDataType    LAMGetDataType(const LexicalAssocMem*);
LexicalEntry       LAMGetGraphyFromId(const LexicalAssocMem*,
                                      const LexicalEntryIndex);
void               LAMGoCharacterBackward(LexicalAssocMem*);
size_t             LAMGetGraphyLength(const LexicalAssocMem*);
LexicalEntryIndex  LAMGetGraphyId(const LexicalAssocMem*);
void               LAMGoToFirstEntry(LexicalAssocMem*);
LexicalEntry       LAMGetNextEntry(LexicalAssocMem*);
gboolean           LAMIsAtEndOfGraphy(LexicalAssocMem*);
```

Remark: `LAM` abbreviation stands for *Lexical Associative Memory*.

1.2.3. generictree module

generictree is a redesigned version of former **arbregen** module which implements a generic tree data structure and its related functions. This module supplies the internal basis for the **trie** (former **arbrelex**) module.

```
typedef unsigned long TreeIndex; /* Former Indice */
typedef unsigned char TreeNode;  /* Former Noeud */
typedef TreeNode*     Tree;      /* Former Arbre */

typedef {
    COMPRESSED_TREE,             /* Former COMPRESSE */
    GROWING_TREE,                /* Former FABRICATION */
} TreeFormat;                    /* Former Format_arbre */

typedef struct {                 /* Former _espece_ */
    TreeFormat format;           /* Former type */
    gboolean node_overflow;      /* Former deb_val */
    gboolean address_overflow;   /* Former deb_adr */

    /* Former Decode_adresse */
    TreeIndex        (*decodeAddress)(const TreeNode*);

    /* Former affiche_noeud */
    void                  (*dumpNode)(const TreeNode*,
                                      int (*)(const char*, ... ));

    /* Former Ordre_Noeud */
    int                  (*nodeOrder)(const TreeNode*,
                                      const TreeNode*);

    /* Former Vers_Suivant_A */
    TreeIndex (*getNextSiblingOffset)(const TreeNode*,
                                      const TreeSpecies*);

    /* Former Vers_Suivant_N */
    TreeIndex   (*getNextChildOffset)(const TreeNode*,
                                      const TreeSpecies*);

    unsigned char value_size;    /* Former value_length */
    unsigned char address_size;  /* Former addr_length */

    size_t memory_size;          /* Former taille_memoire */
    size_t allocated_size;       /* Former taille_allouee */
} TreeSpecies;                   /* Former Espece */

/* Former A_faire */
typedef void (*toDoFunc)(const TreeNode*, const TreeSpecies*);

/* RIEN_A_FAIRE */
#define NOTHING_TO_DO  ((toDoFunc)  0)
```

Introduced function:

treeNodeDecodeChar (former **Decode_Char** from former **arbrelex** module)

1.2.4. trie module

`trie` is a redesigned version of former `arbrelex` module which proposes a trie (lexical tree) data structure and its related functions. This module provides a concrete implementation of a lexical access table, or in other words, supplies implementations for all the functions of the `LexicalAccessTable` structure. The introduction of the `LexicalEntry` abstraction allows to merge peers of equivalent functions (one for `Trie` of `char`, the other for `Trie` of `unsigned long`) from the original architecture. For example, `Insere_Lexico` and `Insere_Lexico_Ulong` become a single `trieInsert` function in the new *API*.

```
typedef unsigned long TrieLeaf;         /* Former Feuille_arbre_lexico */

typedef struct {
    Tree*            tree;              /* Former arbre */
    TreeSpecies*     species            /* Former espece */
    size_t           size;              /* Former nb_entrees */
    LexicalDataType  type;              /* NEW - trie type (char or ulong) */
    TreeIndex        actual_index;      /* NEW - used by trieGoToCharacter */
    TreeIndex        next_index;        /* NEW - used by trieGoToCharacter */
    LongStack*       exploration_stack; /* NEW - used by trieGetNextGraphy */
    LongStack*       index_stack;       /* NEW - used by trieGoCharacterForward */
} Trie;                                 /* Former Arbre_lexico */
```

List of brand-new functions:

- `trieImplementAccessTable` – implement a `LexicalAccessTable` with the `Trie` data structure.
- `trieAdd` – implement `LexicalAccessTableFunctions->insert` field.
- `trieDump` - implement `LexicalAccessTableFunctions->dump` field.
- `trieGetSize` - implement `LexicalAccessTableFunctions->getSize` field.
- `trieGoToRoot` - implement `LexicalAccessTableFunctions->goToRoot` field.
- `trieGoToFirstEntry` - implement `LexicalAccessTableFunctions->goToFirstEntry` field.
- `trieGetNextEntry` - implement `LexicalAccessTableFunctions->getNextEntry` field.
- `trieGoCharacterForward` - implement `LexicalAccessTableFunctions->goCharacterForward` field.
- `trieIsAtEndOfGraphy` - implement `LexicalAccessTableFunctions->isAtEndOfGraphy` field.
- `trieGetNextAvailableCharacter` - implement `LexicalAc…->getNextAvailableCharacter` field.
- `trieGetDataType` - implement `LexicalAccessTableFunctions->getDataType` field.
- `trieGoCharacterBackward` - implement `LexicalAccessTableFun…->goCharacterBackward` field.
- `trieGetGraphyLength` - implement `LexicalAccessTableFunctions->getGraphyLength` field.
- `trieGetGraphyId` - implement `LexicalAccessTableFunctions->getGraphyId` field.

List of internal-use functions that are not exported:

- `Get_Fils` - encapsulated by the more convenient `Cherche_Fils` function.
- `Fichier_Read_Arbre_Lexico` – encapsulated by more convenient `Read_Arbre_Lexico` function.
- `Fichier_Write_Arbre_Lexico` – encapsulated by more convenient `Write_Arbre_Lexico` function.
- `Fichier_Importe_Arbre_Lexico` – encapsulated by more handy `Importe_Arbre_Lexico` function.
- `Recherche_Lexico` & `Recherche_Lexico_Ulong` – encapsulated by the more useful `GetFirst_Valeur_Lexico` & `GetFirst_Valeur_Lexico_Ulong` functions.

List of useless functions that are deleted:

- `Fichier_Importe_Une_Entree_Lexico` - function not specific to tries and moved to `utils` module.
- `Libere_Code_Inversion` - function never used.
- `Decode_Valeur_Lexico` – exact copy of `Decode_Ulong_BdD`.

The remaining functions are in general simply renamed.

1.2.5. bidirectionaltrie module

This brand new module implements a lexical associative memory based on lexical tree, by providing implementations to each field of **LexicalAssocMem**. **BidirectionalTrie** data structure encapsulates a simple trie (**trie** field) and an array (**inversion_code_array** field) that supplies the reciprocal access (graphy from index).

```
typedef struct {
    Trie*      trie;
    LongArray* inversion_code_array;
} BidirectionalTrie; /* NEW */
```

This module is strongly linked to the **trie** module and most of the provided functions are simple calls on the encapsulated **trie** field. Here is the list of the functions that offer extra treatments:

bidirectionalTrieDump – dump the bidirectional trie content in a slightly different manner from **trieDump**, i.e. the key followed by its corresponding graphy.

bidirectionalTrieCreate – create the encapsulated trie (with **trieCreate**), as well as the inversion code array.

bidirectionalTrieFree – free the encapsulated trie (with **trieFree**), as well as the inversion code array.

bidirectionalTrieExport – save the list of valid keys (from **inversion_code_array** field) in a textual file, and theirs corresponding graphies (from **trie** field) in another one.

bidirectionalTrieImport – load a bidirectional trie with the couple of files generated by a **bidirectionalTrieExport** call.

bidirectionalTrieSave – save the encapsulated trie (with **trieSave**), and write the inversion code array content in a binary file.

bidirectionalTrieLoad – load the encapsulated trie (with **trieLoad**), and fill the inversion code array with the content of a second binary file.

bidirectionalTrieInsert – perform an insertion in the encapsulated **trie** (with **trieInsert**), and set the **inversion_code_array** element indexed by the given key to the resulting insertion code.

List of brand new functions:

bidirectionalTrieImplementAssociativeMemory - implement a **LexicalAssocMem** with the **BidirectionalTrie** data structure.

bidirectionalTrieAccess – implement **LexicalAssocMem->access** field.

1.2.6. automaton module

`automaton` is a redesigned version of former **`automategen`** module that implements a characters finite state automaton (*FSA*) data structure and its related functions. The implemented automaton doesn't allow duplicates, i.e. only one index can be associated to a particular graphy. **`automaton`** uses **`reprmem`** and **`autoconstr`** modules that have not been redesigned since they are only internally used and invisible to the *API* users.

```
typedef unsigned char AutomatonState;      /* Former Etat */
typedef unsigned char AutomatonTransition; /* Former Trans */
typedef unsigned long AutomatonLabel;      /* NEW */
typedef unsigned long AutomatonEntryIndex; /* Former IndiceDict */

typedef struct {
    LexicalDataType data_type;             /* NEW */
    unsigned char   automaton_info;        /* Former infoAutomat */
    unsigned char   label_size;            /* Former nrOctAlpha */
    size_t          recognized_sequences;  /* Former nrSecReconnues */
    size_t          memory_size;           /* Former tailleUtilisee */
    AutomatonState* first_state;           /* Former premierEtat */

    AutomatonState* current_state;     /* NEW - used  by GetNextCharacter, ... */
    AutomatonState* next_state;        /* ... GoToRoot, EndOfWord */
    LongStack*      explore_stack;     /* NEW - used by automatonGetNextGraphy */

    /* Former match */
    gboolean      (*equals)(const AutomatonTransition*, const AutomatonLabel);

    /* Former matchLess */
    gboolean      (*isLess)(const AutomatonTransition*, const AutomatonLabel);

    /* Former decodeDsUlong */
    AutomatonLabel (*decode)(const AutomatonState*);

    /* Former assign */
    void          (*assign)(AutomatonTransition*, const AutomatonLabel);
} Automaton; /* Former Automate */

typedef struct {
    gboolean found;
    AutomatonEntryIndex index;
} AutomatonSearchResult; /* NEW */
```

List of the exported functions :

```
/* Former InitAutomateChar, InitAutomateUlong & InitAutomateTypeModel */
void automatonCreate(Automaton* automaton,
                     const LexicalDataType data_type);

/* Former LibereAutomate */
void automatonFree(Automaton* automaton);

/* Former RetrouveSeqAutomate - return NULL if entry is not found */
LexicalEntry automatonAccess(const Automaton* automaton,
                             const AutomatonEntryIndex index);

/* Former ConcatAutomate */
Automaton* automatonApplyUnion(const Automaton* automaton1,
                               const Automaton* automaton2);

/* Former ConcatAutomate */
Automaton* automatonApplyConcatenation(const Automaton* automaton1,
                                       const Automaton* automaton2);

/* Former KleeneClosAutomate */
Automaton* automatonApplyKleeneClosure(const Automaton* automaton);

/* Former IntersectAutomate */
Automaton* automatonApplyIntersection(const Automaton* automaton1,
                                      const Automaton* automaton2);

/* Former ReverseAutomate */
Automaton* automatonReverse(const Automaton* automaton);

/* Former CompleteAutomate */
Automaton* automatonComplete(const Automaton* automaton,
                             const size_t alphabet_size);

/* Former ComplementAutomate */
Automaton* automatonComplement(const Automaton* automaton,
                               const size_t alphabet_count);

/* Former RechercheSeqChar & RechercheSeqUlong */
AutomatonSearchResult* automatonSearch(const Automaton* automaton,
                                       const LexicalEntry sequence);

/* Former EcrireAutomateCharWithNum & ListAutomateChar */
int automatonDumpContents(const Automaton* automaton,
                          int (*print)(const char*, ... ));

/* Former ListStructAutomate */
int automatonDumpGraph(const Automaton* automaton,
                       int (*print)(const char*, ... ));

/* Former ExporteAutomateGraph */
int automatonExportGraph(const Automaton* automaton,
                         const char* filename);

/* Former ExporteAutomateChar */
int automatonExportContents(const Automaton* automaton,
                            const char* filename);

/* Former WriteAutomate */
int automatonSave(const Automaton* automaton,
                  const char* filename);

/* Former ReadAutomate */
int automatonLoad(Automaton* automaton,
                  const char* filename);

/* Former DupliqueAutomate */
int automatonDuplicate(const Automaton* source,
                       Automaton* destination);

/* Former NumeroteAutomate */
int automatonNumber(Automaton* automaton);

/* Former DeNumeroteAutomate */
int automatonDenumber(Automaton* automaton);
```

```
/* Former MinimiseAutomate */
int automatonMinimize(Automaton* automaton);


/* Former EpsilonRmAutomate */
int automatonRemoveEpsilon(Automaton* automaton);


/* Former DeterminAutomate */
int automatonMakeDeterminist(Automaton* automaton);

/* Former RajouteSequenceAutomate */
int automatonInsert(Automaton* automaton,
                    const LexicalEntry sequence);

/* Former ImporteAutomateGraph */
int automatonImportGraph(Automaton* automaton,
                         const char* filename,
                         const gboolean minimized,
                         const gboolean numbered);

/* Former ImporteAutomateDictionnaire(a,b,c,d, FAUX) */
int automatonImportContents(Automaton* automaton,
                            const char* filename,
                            const gboolean minimized,
                            const gboolean numbered);

/* Uses former SequenceSuivante */
LexicalEntry automatonGetNextGraphy(Automaton* automaton);


/* Former DecodePileToChar */
int automatonDecodeTransitionStack(const Automaton* automaton,
                                   const LongStack* transitions,
                                   LexicalEntry* word);
```

List of brand new functions:

```
/* NEW */
void automatonGoToFirstGraphy(Automaton* automaton);

/* NEW */
LexicalCharacter automatonGetNextCharacter(Automaton* automaton);

/* NEW */
void automatonGoToRoot(Automaton* automaton);

/* NEW */
void automatonGoToCharacter(Automaton* automaton);

/* NEW */
gboolean automatonEndOfWord(Automaton* automaton);

/* NEW */
gboolean automatonIsMinimized(const Automaton* automaton);

/* NEW */
gboolean automatonIsNumeroted(const Automaton* automaton);

/* NEW */
gboolean automatonIsDeterminist(const Automaton* automaton);
```

List of useless functions that are deleted:

`FichierWriteAutomate` – functionality already implemented by more useful `automatonSave` function.

`FichierReadAutomate` – functionality already implemented by more useful `automatonLoad` function.

1.2.7. lexicalautomaton module

`lexicalautomaton` is a redesigned version of former `automatelex` module which provides a lexical finite state automaton that allows duplicates. It supplies a concrete lexical associative memory based on *FSA* by implementing all fields of the `LexicalAssocMem` data structure.

```
typedef size_t LexicalAutomatonState;    /* NEW */

typedef struct {
    LexicalAutomatonState start_state;   /* Former lEtat */
    AutomatonTransition*  transition;    /* Former pTr */
} LexicalAutomatonTransition;            /* Former trans */

typedef struct {
    Automaton*                 automaton;     /* Former lAuto */
    LongArray*                 graphy;        /* Former graphie */
    LongArray*                 duplicate;     /* Former doublon */
    LexicalAutomatonTransition current_state; /* NEW */
    LexicalAutomatonTransition next_state;    /* NEW */
    LongStack*                 explore_stack; /* NEW */
} LexicalAutomaton;                           /* Former AutomateLex */
```

List of exported functions:

```
/* Former LibereAutomateLex - implements "free" function */
void lexicalAutomatonFree(LexicalAutomaton* automaton);

/* Former InitAutomateLex* - implements "create" function */
void lexicalAutomatonCreate(LexicalAutomaton* automaton,
                            const LexicalDataType data_type);

/* Former ExporteAutomateLex - implements "export" function */
int lexicalAutomatonExport(const LexicalAutomaton* automaton,
                           const char* filename);

/* Former WriteAutomateLex - implements "save" function */
int lexicalAutomatonSave(const LexicalAutomaton* automaton,
                         const char* filename);

/* Former ReadAutomateLex - implements "load" function */
int lexicalAutomatonLoad(LexicalAutomaton* automaton, const char* filename);

/* Former ListAutomateLex - implements "dump" function */
int lexicalAutomatonDumpContents(const LexicalAutomaton* automaton,
                                 int (*print)(const char*, ...));

/* Former ImporteAutomateLex - implements "import" function */
int lexicalAutomatonImport(LexicalAutomaton* automaton, const char* filename);

/* Former RechercheLex & RechercheLexUlong - implements "searchFirst" function */
LexicalSearch lexicalAutomatonSearchFirst(const LexicalAutomaton* automaton,
                                          const LexicalEntry entry);

/* Former RechercheSuivantDansLex - implements "searchNext" function */
gboolean lexicalAutomatonSearchNext(LexicalSearch* result);

/* Former DupliqueAutomateLex */
int lexicalAutomatonDuplicate(const LexicalAutomaton* source,
                              LexicalAutomaton* destination);

/* Former CEstTrans */
gboolean lexicalAutomatonCharIsChild(const LexicalAutomaton* automaton,
                                     const LexicalCharacter c,
                                     const LexicalAutomatonTransition parent,
                                     LexicalAutomatonTransition* child);

/* Uses former GetTrans (extra code required) */
gboolean lexicalAutomatonGetNextChild(const LexicalAutomaton* automaton,
                                      const LexicalAutomatonTransition parent,
                                      LexicalAutomatonTransition* child,
                                      LexicalCharacter* character);
```

```
/* Former FinAutomate */
gboolean
  lexicalAutomatonTransitionHasNoSuccessor(const LexicalAutomaton* automaton,
                                           const LexicalAutomatonTransition tr);

/* Former EtatFinal (used by endOfWord function) */
gboolean lexicalAutomatonTransitionOnFinalState(
             const LexicalAutomaton* automaton,
             const LexicalAutomatonTransition tr);

/* Implements "endOfWord" (uses lexicalAutomatonTransitionOnFinalState) */
gboolean lexicalAutomatonEndOfWord(LexicalAutomaton* automaton);

/* Implements getNextCharacter function (uses lexicalAutomatonGetNextChild) */
LexicalCharacter lexicalAutomatonGetNextCharacter(LexicalAutomaton* automaton);

/* Implements "getNextGraphy" function (uses SequenceSuivante) */
LexicalEntry lexicalAutomatonGetNextGraphy(LexicalAutomaton* automaton);
```

List of brand new functions:

```
/* NEW */
LexicalAssocMem* lexicalAutomatonGetAssociativeMemory
                     (const LexicalAutomaton* source);

/* Implements "getSize" function */
size_t lexicalAutomatonGetSize(const LexicalAutomaton* automaton);

/* Implements "inser"t function */
int lexicalAutomatonInsert(LexicalAutomaton* automaton,
                           const LexicalEntry entry);

/* Implements "access" function */
LexicalEntry lexicalAutomatonAccess(const LexicalAutomaton* automaton,
                                    const LexicalEntryIndex index);

/* Implements "goToFirstGraphy" function */
void lexicalAutomatonGoToFirstGraphy(LexicalAutomaton* automaton);

/* Implements "goToCharacter" function */
void lexicalAutomatonGoToCharacter(LexicalAutomaton* automaton);

/* Implements "goToRoot" function */
void lexicalAutomatonGoToRoot(LexicalAutomaton* automaton);
```

List of useless former functions that are deleted:

`OptimiserTab` – Low-level implementation specific function that is not necessary anymore.

`FichierWriteAutomateLex` – functionality already implemented by the more general `lexicalAutomatonSave` function.

`FichierReadAutomateLex` – functionality already implemented by the more general `lexicalAutomatonLoad` function.

1.2.8. lexicaltransducer module

`lexicaltransducer` is a redesigned version of former **`transducer`** module which supplies a lexical finite state transducer. It provides a concrete lexical associative memory based on *FST* by implementing all fields of the **`LexicalAssocMem`** data structure.

```
typedef enum {
    IDENTITY,
    RELATION,
    NONE
} TransducerType;                 /* Former IdType */

typedef struct {
    TransducerType type;          /* Former id */
    Automaton*     automaton;     /* Former automaton */
    Automaton*     sigma1;        /* Former sigma1 */
    Automaton*     sigma2;        /* Former sigma2 */
} LexicalTransducer;              /* Former Transducer */
```

List of exported functions:

```
/* Former InitTransducer - implements "create" function */
void lexicalTransducerCreate(LexicalTransducer* transducer,
                             const LexicalDataType type);

/* Former DeleteTransducer - implements "free" function */
void lexicalTransducerFree(LexicalTransducer* transducer);

/* Former ReadTransducer - implements "load" function */
int lexicalTransducerLoad(LexicalTransducer* transducer,
                          const char* filename);

/* Former WriteTransducer - implements "save" function */
int lexicalTransducerSave(const LexicalTransducer* transducer,
                          const char* filename);

/* Former CopyTransducer */
void lexicalTransducerDuplicate(const LexicalTransducer* source,
                                LexicalTransducer* destination);

/* Former UnionTransducer */
LexicalTransducer* lexicalTransducerUnion(const LexicalTransducer* t1,
                                          const LexicalTransducer* t2),

/* Former IntersectTransducer */
LexicalTransducer* lexicalTransducerIntersection(const LexicalTransducer* t1,
                                                 const LexicalTransducer* t2),

/* Former ConcatTransducer */
LexicalTransducer* lexicalTransducerConcatenation(const LexicalTransducer* t1,
                                                  const LexicalTransducer* t2),

/* Former KleeneStarTransducer */
LexicalTransducer* lexicalTransducerKleeneClosure(const LexicalTransducer* t1),

/* Former CompTransducer */
LexicalTransducer* lexicalTransducerComposition(const LexicalTransducer* t1,
                                                const LexicalTransducer* t2),

/* Former CrossTransducer */
LexicalTransducer* lexicalTransducerCrossProduct(const LexicalTransducer* t1,
                                                 const LexicalTransducer* t2);

/* Former ComplementTransducer - return an error code */
int lexicalTransducerComplement(LexicalTransducer* t1);

/* Former TransductWord */
PointerArray* lexicalTranscuderTransductionUpper(const LexicalTransducer* td,
                                                 const LexicalEntry entry);


/* Former TransductWord */
PointerArray* lexicalTranscuderTransductionLower(const LexicalTransducer* td,
                                                 const LexicalEntry entry);
```

```
/* Former ExportTransducerDict - implements "export" function */
int lexicalTransducerExport(const LexicalTransducer* transducer,
                            const char* filename);


/* Former ImportTransducerGraph */
int lexicalTransducerImportGraph(LexicalTransducer* transducer,
                                 const char* filename);


/* Former ExportTransducerGraph */
int lexicalTransducerExportGraph(const LexicalTransducer* transducer,
                                 const char* filename);

/* Former ImportTransducerDict -  implements "import" function */
/* The last parameter of former ImportTransducerDict is defined by the type of
   the transducer given as first parameter */
int lexicalTransducerImport(LexicalTransducer* transducer, const char* filename);


/* Former PrintTransducer - implements "dump" function */
int lexicalTransducerDump(const LexicalTransducer* transducer,
                          int (*print)(const char*, ... ));

/* NEW - implements "getSize" function */
/* Return the number of transduced sequences */
size_t lexicalTransducerGetSize(const LexicalTransducer* transducer);
```

List of brand new functions:

```
/* Implement "insert" function - Add a graphy ⇨ graphy entry */
int lexicalTransducerInsertIdentity(LexicalTransducer* transducer,
                                    const LexicalEntry graphy);

/* Standard transducer insertion */
void lexicalTransducerInsert(LexicalTransducer*,
                             const LexicalEntry entry1,
                             const LexicalEntry entry2);

/* Implement "searchFirst" function - seach into the upper alphabet */
LexicalSearch lexicalTransducerSearchFirst(const LexicalTransducer* transducer,
                                           const LexicalEntry entry);

/* Implement "searchNext" function - search into the upper alphabet */
gboolean lexicalTransducerSearchNext(LexicalSearch* search);

/* Implement "getNextGraphy" function - list the upper alphabet */
LexicalEntry lexicalTransducerGetNextGraphy(LexicalTransducer* transducer);

/* Implement "goToFirstGraphy" function - list the upper alphabet */
void lexicalTransducerGoToFirstGraphy(LexicalTransducer* transducer);

/* Implement "getNextCharacter" function - explore the upper alphabet */
LexicalCharacter lexicalTransducerGetNextCharacter
                     (LexicalTransducer* transducer);

/* Implement "goToRoot" function - explore the upper alphabet */
void lexicalTransducerGoToRoot(LexicalTransducer* transducer);

/* Implement "goToCharacter" function - explore the upper alphabet */
void lexicalTransducerGoToCharacter(LexicalTransducer* transducer);

/* Implement "endOfWord" function - explore the upper alphabet */
gboolean lexicalTransducerEndOfWord(LexicalTransducer* transducer);

/* NEW */
LexicalAssocMem* lexicalTransducerGetAssociativeMemory
                     (const LexicalTransducer* source);
```

1.2.9. lexicon module

`lexicon` is a redesigned version of former `lexique` module that offers a lexicon data structure and its related functions. Each entry is possibly characterized by a probability, a frequency, a lemma and a part-of-speech (string or numerical). The main evolution in relation to the former architecture is the explicit definition of two types of lexicon:

- « The textual lexicon » - it stores `char`-based strings, allows to treat the part-of-speech field as string and is used for vocabulary words. To create such lexicon, one specifies the `CHARACTER` value to `lexiconCreate`.
- « The numerical lexicon » - it stores `unsigned long`-base strings (most often used as right part of rules), treats the part-of-speech field (most often used as left part of rules) as a numerical value and is used to list rules. To create such lexicon, one specifies the `UNSIGNED_LONG` value to `lexiconCreate`.

The main reason of this contribution is that some treatments provided by *SlpToolKit* are concretely available or appropriate only on one of the two lexicon types. For example, lexical correction is only suitable on textual lexicon. Specifying two explicit and strict lexicon types permits to control and check the access to those treatments.

Another major modification is the fact that the graphy field is now stored by an instance of lexical associative memory. The concrete implementation is chosen during the lexicon creation. Attentive readers may also notice the lack of genericity between the type of the lexicon fields (for example `unsigned int` for `Frequency`) and the corresponding storing array (for example `IntArray*` for `frequencies`). Since no satisfactory solution to this problem has been found, possible field type alterations must be explicitly transferred to the corresponding lexicon field array.

```
typedef double             Probability;        /* Former Type_proba */
typedef unsigned int       Frequency;          /* Former Type_freq */
typedef LexicalEntryIndex  Lemma;              /* Former Type_lemme */
typedef unsigned short     Morpho;             /* Former Type_morpho */

typedef struct {
    Morpho                 part_of_speech;     /* Replace opencms */
    Probability            probability;        /* Replace opencmsproba */
} OpenPosTableItem;                            /* NEW - Item of a OpenPosTable */

typedef struct {
    IntArray*              values;             /* Former morpho */
    PointerArray*          open_pos_table;     /* Former opencms & opencmsproba */
    Morpho                 max_pos;            /* Former cms_max */
    BidirectionalTrie*     pos_table;          /* NEW - POS conversion table */
} LexiconMorpho;

typedef struct {
    LexicalAssocMem*       associative_memory; /* Replaces arbre_lexico */
    LongArray*             lemma;              /* Former lemme */
    IntArray*              frequency;          /* Former freq */
    DoubleArray*           probability;        /* Former proba */
    LexiconMorpho*         morpho;             /* NEW */
} Lexicon;                                     /* Former lexique */

typedef struct {
    LexicalSearch          search;             /* Former recherche */
    LexiconAccess          access;             /* Former acces */
} LexiconSearch;                               /* Former Retour_recherche_lex */

typedef struct {
    Lexicon*               source;             /* NEW - used by GetGraphy */
    LexicalEntryIndex      index;              /* NEW - used by GetGraphy */
    Lemma                  lemma;              /* Former lemme */
    Morpho                 part_of_speech;     /* Former morpho */
    Frequency              frequency;          /* Former freq */
    Probability            probability;        /* Former proba */
} LexiconAccess;                               /* Former Retour_accede_lex */
```

List of exported functions:

```
/* Former Get_Graphie */
LexicalEntry lexiconAccessGetGraphy(const LexiconAccess*);

/* Former Init_Lexique */
int lexiconCreate(Lexicon*,
                  const LexicalMemoryType,
                  const LexicalDataType);

/* Former Exporte_Lexique */
int lexiconExport(const Lexicon*,
                  const char*);

/* Former Importe_Lexique */
int lexiconImport(Lexicon*,
                  const char*);

/* Former Write_Lexique */
int lexiconSave(const Lexicon*,
                const char*);

/* Former Write_proba */
int lexiconSave_probability(const Lexicon*,
                            const char*);

/* Former Write_lemme */
int lexiconSave_lemma(const Lexicon*,
                      const char*);

/* Former Write_morpho */
int lexiconSave_morpho(const Lexicon*,
                       const char*);

/* Former Write_freq */
int lexiconSave_frequency(const Lexicon*,
                          const char*);

/* Former Read_Lexique */
int lexiconLoad(Lexicon*,
                const char*);

/* Former Log_Proba */
void lexiconApplyLogOnProba(Lexicon*);

/* Former Libere_Lexique */
void lexiconFree(Lexicon*);

/* Former Liste_Lexique */
void lexiconDump(Lexicon*,
                 int (*)(... ),
                 const char*,
                 gboolean,
                 gboolean);

/* Former Recherche_Lexique */
LexiconSearch lexiconLookFor_CHARACTER(const Lexicon*,
                                       const char*);

/* Former Recherche_Lexique_Ulong */
LexiconSearch lexiconLookFor_UNSIGNED_LONG(const Lexicon*,
                                           const LongArray*);

/* Former Accede_Lexique */
LexiconAccess lexiconAccess(const Lexicon*,
                            const LexicalEntryIndex);

/* Former Suivant_Lexique */
gboolean lexiconSearchNext(LexiconSearch*);

/* Former Normalise_Proba(..., VRAI) */
gboolean lexiconIsNormalized(const Lexicon*);

/* Former Normalise_Proba(..., FAUX) */
gboolean lexiconNormalizeProba(Lexicon*);
```

```
/* Former Insere_Lexique & Insere_Lexique_De_Force */
gboolean lexiconInsert_CHARACTER(Lexicon*,
                                 const char*,
                                 const Lemma*,
                                 const char*,
                                 const Frequency*,
                                 const Probability*,
                                 LexicalEntryIndex*,
                                 const gboolean);

/* Former Insere_Lexique_Ulong & Insere_Lexique_De_Force_Ulong */
gboolean lexiconInsert_UNSIGNED_LONG(Lexicon*,
                                     const LongArray*,
                                     const Lemma*,
                                     const Morpho*,
                                     const Frequency*,
                                     const Probability*,
                                     LexicalEntryIndex*,
                                     const gboolean);


/* Former Dans_Lexique */
gboolean lexiconContains_CHARACTER(const Lexicon*,
                                   const char*,
                                   const Lemma*,
                                   const char*,
                                   const Frequency*,
                                   const Probability*);

/* Former Dans_Lexique_Ulong */
gboolean lexiconContains_UNSIGNED_LONG(const Lexicon*,
                                       const LongArray*,
                                       const Lemma*,
                                       const Morpho*,
                                       const Frequency*,
                                       const Probability*);
```

List of brand new functions:

```
/* NEW */
char* lexiconAccessGetPartOfSpeech(const LexiconAccess*);

/* NEW */
size_t lexiconGetSize(const Lexicon* lexicon);
```

1.2.10. regularexpression module

regularexpression is a redesigned version of former **regexp** module that allows to manipulate symbolic regular expressions.

```
typedef int RegExp; /* Former Regexp */
```

List of exported functions:

```
/* Former AutomatonToRE */
int regExpAutomatonToString(const Automaton* automaton,
                            GString* regular_expression);

/* Former OpRegexp(a,b,cat) */
RegExp regExpApplyContenate(const RegExp reg_exp1, const RegExp reg_exp2);

/* Former OpRegexp(a,b,star) */
RegExp regExpApplyKleeneClosure(const RegExp reg_exp);

/* Former OpRegexp(a,b,or) */
RegExp regExpApplyOr(const RegExp reg_exp1, const RegExp reg_exp2);

/* Former CreateRegexpTree(null,x)->myNr */
RegExp regExpNull();

/* Former CreateRegexpTree(epsilon,x)->myNr */
RegExp regExpEpsilon();

/* Former DestroyRegexp */
void regExpFree(RegExp reg_exp);

/* Former RegexpToXCString */
void regExpToString(const RegExp reg_exp, GString* string);

/* Former PrettyPrint */
void regExpPrettyDump(const char* reg_exp, int (*print)(const char*, ... ));

/* Former CreateRegexpTree(chr, value) */
RegExp regExpCreateAtomic(const char value);

/* Former CreateRegexpTree(cat, x) */
RegExp regExpCreateConcatenationOperation();

/* Former CreateRegexpTree(or, x) */
RegExp regExpCreateOrOperation();

/* Former CreateRegexpTree(star, x) */
RegExp regExpCreateKleeneClosureOperation();

/* Former CreateRegexpTree(plus, x) */
RegExp regExpCreatePlusOperation();
```

List of internal-use functions that are not exported:

DecNrReferences & **IncNrReferences** – Low-level functions that must be hidden to the end-user.

RemoveNode – Low-level function that must be hidden to the end-user.

ExistsRegexp – Low-level function that must be hidden to the end-user.

MergeStarToPlus – Only used by **FactorOutEpsilon** & **OpRegexp**.

RemoveDoubleOrOperands – Only used by **FactorOutEpsilon** & **OpRegexp**.

CopySons – Only used by **FactorOutEpsilon** & **OpRegexp**.

FactorOutEpsilon – Only used by **OpRegexp**.

List of useless functions that are deleted:

`StartRegexpModule` & `StopRegexpModule` – Should be initially executed before using the module and when the module is no more used, but this is now made automatically. Indeed, initialization occurs when first regular expression is created and finalization happens when last regular expression is destroyed.

`RegisterRegexpTree` – Used only once, so its code can be directly copied.

1.2.11. flexconvert

`flexconvert` is a redesigned version of former `flextranslate` module that converts from *flex* format into finite state automata (*FSA*).

List of exported functions:

```
/* Former ExportAutomatonGraph */
int flexConvertToGraph(const char* input, char* output);

/* Former ExportAutomatonDict */
int flexConvertToContents(const char* input, char* output);

/* Former ReadArrays (exported for test purpose only) */
int flexConvertReadArrays(FILE* inFile, FlexArrays* arrays);

/* Former FreeArrays (exported for test purpose only) */
void flexConvertFreeArrays(FlexArrays* arrays);
```

List of internal-use functions that are not exported:

`ParseToInt` – Low-level function for internal use only.

`NextState` – Module specific low-level function for internal use only.

`OpenFiles` & `CloseFiles` – Functions internally used by exportation functions.

`BuildCharacterTable` & `FreeCharacterTable` – Functions internally used by exportation functions.

`SetFileNames` – Function internally used by exportation functions.

`GetInternalAutomatonInfo` – Function internally used by exportation functions.

`HasCycles` & `AppendToWordlist` – Functions only used in former `ExportAutomatonDict` function.

1.2.12. lexicalcorrection module

`lexicalcorrection` is a redesigned version of former `corrlex` and `corrlexalgos` modules, and also encapsulates the contribution of Christophe de Benoit. This module implements some algorithms related to the lexical spelling error correction, and declares its associated data structures.

The new architecture allows `spellCorrectFlat` function (former `Correction_Lexico`) to be called with any lexical memory that implements the `LexicalAccessTable` structure (and not only `Trie`). The only restriction is that the access table data type must be `CHARACTER`. A source field has been added to the `SolutionSet` structure to avoid the call of `solutionGetString` function with a wrong lexical memory.

```
enum {
    BASIC_CORRECTION = 0,            /* Former CORRECTION_BASE */
    WEIGHTED_CORRECTION,             /* Former CORRECTION_PONDEREE */
    SPLITED_CORRECTION,              /* Former CORRECTION_SEPARE */
} CorrectionMode;                    /* Former Mode_Correction */

/* Former CORRECTION_DEFAUT */
#define DEFAULT_CORRECTION WEIGHTED_CORRECTION

/* Former Type_cout */
typedef double Weight;

/* Former MODE_0_OFFSET */
#define COST_RANGE 0.5

/* Former FMT_COUT */
#define COST_FORMAT "%4.2f"

typedef struct {
    LexicalEntryIndex entry_index;     /* NEW - replaces pos_arbre */
    short             string_position; /* Former pos_chaine */
} SolutionPart;

typedef struct {
    GSList* parts;       /* Former constituants - array of SolutionPart* */
    Weight  cost;        /* Former cout */
} Solution;              /* Former Solution */

typedef struct {
    LexicalAccessTable* source;    /* NEW - used by solutionGetString function */
    PointerArray*       solutions; /* Former ensemble (array of Solution) */
} SolutionSet;                     /* Former Ensemble_solutions */

/* Former Liste_pos_chaine */
typedef GSList PositionList;
```

List of exported functions:

```
/* Former Correction_Lexico */
void spellCorrectFlat(const char*,
                      const LexicalAccessTable*,
                      const int,
                      const CorrectionMode,
                      const Weight,
                      const Weight,
                      const Weight,
                      const Weight,
                      SolutionSet*);

/* Former Solution_Vers_String */
void solutionGetString(const SolutionSet*,
                       const char*,
                       GString*);

/* Former ajoute_liste_pos_chaine */
void positionListAdd(PositionList**,
                     const short int);

/* Former Libere_Ensemble_Solutions */
void solutionSetFree(SolutionSet*);
```

```
/* Former Correction_Treillis (from Christophe de Benoit's project */
ParsingChart* spellCorrectChart(const char*,
                                const Lexicon*,
                                const Weight,
                                const Weight,
                                const Weight,
                                const Weight);

/* Former Solution_Max_Treillis (from Christophe de Benoit's project) */
PointerArray* parsingChartGetMaxLexemes(const ParsingChart*,
                                        const char*);

/* Former Lexematise & Correction_Zero (from Christophe de Benoit's project) */
ParsingChart* lexematize(const char*,
                         const Lexicon*,
                         gboolean (*)(int),
                         gboolean (*)(int),
                         gboolean (*)(int));
```

As supplied by Christophe de Benoit, the former `Correction_Zero` function doesn't work properly because it doesn't manage *glueable* characters. The problem is that the internal `emplit_table_cyk` function fills the resulting chart incorrectly when such character is met. Indeed, its algorithm needs joinable sentence cut-outs to detect upper-level cells to fill.

Although it wasn't the goal of the current project, this function has been corrected in such a way *lexematize* utility works correctly. The simple (but not pleasing) solution used is to introduce each part of the words with a *glueable* character to the sentence cut-outs. An example of what the supplied solution returns is provided at figure 2.

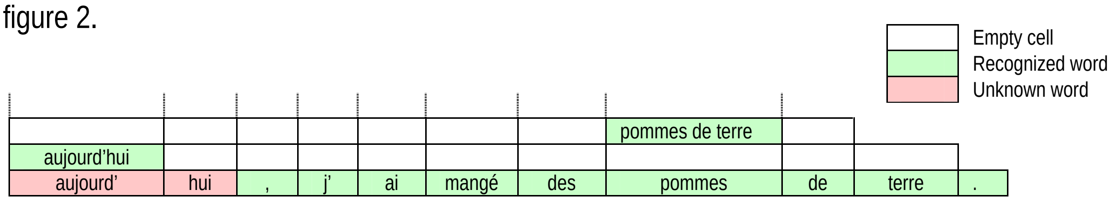

figure 2 – corrected spellCorrectChart output with the "aujourd'hui, j'ai mangé des pommes de terre." sentence

1.2.13. token module

token module offers a function that extracts the tokens from an input text. The redesigned *API* is presented below.

```
typedef struct {
    const char* start;          /* Former debut_token */
    const char* finish;         /* Former fin_token */
} Token;

typedef GSList  TokenList;      /* Former Token (GSList of Token) */

typedef struct {
    TokenList*  tokens;         /* Former liste_tokens */
    const char* process_ending; /* Former dernier */
} Tokenization;                 /* Former Tokenisation */
```

List of exported functions:

```
/* Former Tokenise */
Tokenization tokenize(const char*, const gboolean,
                      int (*)(int),
                      int (*)(int),
                      int (*)(int));

/* Former Libere_Tokenisation */
void tokenizationFree(Tokenization* tokenization);

/* Former Affiche_Tokenisation */
void tokenizationDump(Tokenization*, const char*, int (*)(const char*, ...));

/* Former Affiche_Token */
void tokenDump(const Token*, const char*, int (*)(const char*, ...));

/* Former Join_Token */
gboolean tokenMerge(TokenList*, TokenList*);

/* Former Token2String */
void tokenGetString(const Token*, GString*);
```

1.3 Syntactic modules

This chapter gathers the description of the modules dedicated to data structures or treatments of the syntactical level. The number of grammar related modules has basically increased in relation to the prior architecture since the former **cyk** module is now exploded into several smaller and more logical modules.

1.3.1. grammar module

grammar is a redesigned version of former **grammaire** module. The main conceptual improvement proposed by the new architecture is the adjunction of **words** lexicon to **Grammar** data structure. The vocabulary lexicon content is not saved by **grammarSave** (former **Write_Grammaire**) to avoid lexical resources duplication. As suggested in the previous architecture, the name of the binary file that stores the vocabulary lexicon is printed in the grammar header file to allow reciprocal **grammarLoad** function to automatically load the right **words** lexicon. **grammarLoad** causes a memory allocation for **words** field that must be freed when **grammarFree** is ultimately called. So internal flag **self_lexicon_alloc** notifies the library if such allocation must be released when the grammar is freed.

On the other hand, **grammarExport** doesn't store any information or reference on the vocabulary lexicon. So the developer has to make sure that the **words** field of a grammar is set with the right lexicon **before** calling **grammarImport**. Furthermore, since it has been instantiate elsewhere, the vocabulary lexicon of a grammar set up with **grammarImport** isn't desallocated by **grammarFree**.

```
typedef struct {
    Lexicon*           elements;      /* UNSIGNED_LONG tree based lexicon */
    BidirectionalTrie* conversion;
} RulesLexicon;                       /* NEW - see section Erreur ! Source du
renvoi introuvable. page Erreur ! Signet non défini. */

typedef struct {
    Lexicon*      words;              /* NEW */
    RulesLexicon* rules;              /* Former regles */
    Morpho        top_level_nt;       /* Former valeur_initiale */
    double        equality_ratio;     /* Former equality_ratio */
    gboolean      self_lexicon_alloc; /* NEW */
} Grammar;                            /* Former Grammaire */
```

List of exported functions:

```
/* Former Init_Grammaire */
int grammarCreate(Grammar*);

/* Former Write_Grammaire */
int grammarSave(const Grammar*,
                const char*,
                const char*);

/* Former Read_Grammaire */
int grammarLoad(Grammar*,
                const char*);

/* Former Importe_Grammaire */
int grammarImport(Grammar*,
                  const char*,
                  const gboolean);

/* Former Exporte_Grammaire */
int grammarExport(const Grammar*,
                  const char*);

/* Former Convert_NT */
void grammarGetNTString(const Grammar*,
                        const LexicalEntryIndex,
                        GString*);

/* Former Libere_Grammaire */
void grammarFree(Grammar*);
```

```
/* Former Liste_Grammaire */
void grammarDump(const Grammar*,
                 int (*)(const char*, ... ),
                 const gboolean);

/* Former Numero_vers_Regle */
void grammarGetRuleFromIndex(const Grammar*,
                             const LexicalEntryIndex,
                             char**,
                             StringArray*,
                             Probability*);

/* Former Regle_vers_Numero */
LongArray* grammarGetIndexFromRule(const Grammar*,
                                   const char*,
                                   const StringArray*);

/* Former Convert_NT_char */
LexicalEntryIndex grammarGetNTInternalCode(const Grammar*,
                                           const char*);

/* Former Ajoute_Regle */
LexicalEntryIndex grammarAddRule(Grammar*,
                                 const char*,
                                 StringArray*,
                                 const Probability);

/* Former Ajoute_Une_Occurence_Regle */
Probability grammarIncrementRuleFrequency(Grammar*,
                                          LexicalEntryIndex*,
                                          const char*,
                                          StringArray*);
```

List of internal-use functions that are not exported:

`Absolute_Name_and_Id` – This function is only used internally.

List of useless functions that are deleted:

`Ajoute_NT` – in the new architecture, non-terminals don't have to be added beforehand anymore.

`lexicalNT_to_ulong` – equivalent functionality offered by `grammarGetNTInternalCode`.

1.3.2. parsingresult module

`parsingresult` corresponds to the redesigned version of former `briques_cyk` and `parsing_output` modules. It declares `ParsingResult` data structure that allows to store one particular syntactic (parsing) derivation (interpretation).

```
typedef enum {
    INDENTED,
    FLAT_BRACKETING
} AnalysisFormat;                              /* NEW */

typedef struct {
    const char*       original_word;           /* Former mot */
    long              correction_index;        /* Former index_cor */
    Morpho            part_of_speech;          /* Former cms */
    Weight            correction_cost;         /* Former cout */
    Probability       probability_knowing_pos; /* Former proba */
} Brick;                                       /* Former Brique_cyk */

typedef PointerArray BrickArray;               /* Former Briques_cyk */

typedef struct {
    LongArray*      rules_list;                /* Former regles */
    BrickArray*     brick_array;               /* Former bp */
    GString*        analysis;                  /* Former analyse */
    size_t          output_shifting;           /* Former tab */
    gboolean        output_indented;           /* Former decale */
    AnalysisFormat  analysis_format;           /* Former flat */
    Probability     probability;               /* NEW */
} ParsingResult;                               /* Former Parsing_Output */
```

List of exported functions:

```
/* Former Delete_Parsing_Output */
void parsingResultFree(ParsingResult**);

/* Former Set_Parsing_Output_to_Extended &
          Set_Parsing_Output_to_Bracketed */
void parsingResultSetAnalysisFormat(ParsingResult*,
                                    const AnalysisFormat);

/* Former Ajoute_Brique_cyk */
void brickArrayAddBrick(BrickArray*,
                        const char*,
                        const long,
                        const Morpho,
                        const Probability,
                        const Weight);

/* Former Init_Parsing_Output,
          Init_Parsing_Output_XCString,
          Init_Parsing_Output_Briques_cyk &
          Init_Parsing_Output_Dyntab_longint */
ParsingResult* parsingResultCreate(const AnalysisFormat,
                                   const gboolean,
                                   const gboolean,
                                   const gboolean);
```

New function:

```
void brickArrayRemoveAll(BrickArray*);
```

List of useless former functions that are deleted:

`Init_Briques_Cyk` – As `BrickArray` (former `Briques_cyk`) is implemented by a dynamic array, dedicated constructor is not necessary anymore.

1.3.3. parsingchart module

parsingchart is a redesigned version of former **tablecyk** module that declares and offers all functions to create, manipulate and destroy parsing chart. A parsing chart is basically a triangular matrix that stores all lexical and syntactical information about a given parsed sentence. The major contribution compared to the prior architecture is the merging of former **Table_CYK**, **Table_CYK_Ancree**, **Retour_lexicalise** and **Infos_phrase** data structures into the single **ParsingChart** type.

```
typedef unsigned long Counter; /* Former Nb_interp_type */

typedef struct {
    NTItemList*               nt_item_list;           /* Former liste_l1 */
    PrefixItemList*           prefix_item_list;       /* Former liste_l2 */
    signed long               rule;                   /* Former regle */
    NTItemDerivationList*     next;                   /* Former suivant */
} NTItemDerivationList;                               /* Former Element_L1 */

typedef struct {
    NTItemList*               nt_item_list;           /* Former liste_l1 */
    PrefixItemList*           prefix_item_list;       /* Former liste_l2 */
    PrefixItemDerivationList* next;                   /* Former suivant */
} PrefixItemDerivationList;                           /* Former Element_L2 */

typedef struct
{
    TreeNode*                 grammar_node;           /* Former noeud */
    PrefixItemDerivationList* derivation_list;        /* Former liste_el */
    PrefixItemDerivationList* current_derivation;     /* Former current_interp */
    Counter                   interpretation_counter; /* Former nb_interp */
    Probability               maximum_probability;    /* Former max_probas */
    Probability               sum_probability;        /* Former somme_probas */
    Counter                   ties_counter;           /* Former nb_ties */
    PrefixItemList*           next;                   /* Former suivant */
} PrefixItemList;                                     /* Former Liste_L2 */

typedef struct
{
    NTItemDerivationList*     derivation_list;        /* Former liste_el */
    NTItemDerivationList*     current_derivation;     /* Former current_interp */
    Counter                   interpretation_counter; /* Former nb_interp */
    gboolean                  is_on_top;              /* Former is_on_top */
    Probability               maximum_probability;    /* Former max_probas */
    Probability               sum_probability;        /* Former somme_probas */
    Counter                   ties_counter;           /* Former nb_ties */
    NTItemList*               next;                   /* Former suivant */
} NTItemList;                                         /* Former Liste_L1 */

typedef struct {
    NTItemList*               nt_item_list;           /* Former L1 */
    PrefixItemList*           prefix_item_list;       /* Former L2 */
} ParsingChartCell;                                   /* Former Case_CYK */

typedef struct {
    unsigned int              start;                  /* Former debut */
    unsigned int              end;                    /* Former fin */
} SentenceCut;                                        /* Former Type_decoupage */

typedef struct {
    unsigned int              start_column;           /* Former pos_dep */
    unsigned int              end_column;             /* Former pos_fin */
    Weight                    correction_cost;        /* Former cout */
    LexicalEntryIndex         correction_index;       /* Former index_corr */
    Morpho                    part_of_speech;         /* Former cms */
    Probability               probability;            /* Former proba */
} ParsingChartBasis;                                  /* Former Base_cyk */

typedef struct {
    size_t            chart_size; /* Former Table_CYK->taille */
    ParsingChartCell* cells;      /* Former Table_CYK->table */
    Grammar*          grammar;    /* Former Infos_phrase->grammaire & lexique */
    const char*       sentence;   /* Former Infos_phrase->phrase */
    SentenceCut*      cutting;    /* Former Retour_lexicalise->decoupage */
} ParsingChart; /* Former Table_CYK(_Ancree), Retour_lexicalise, Infos_phrase */
```

List of exported functions:

```
        /* Former ajoute_mot_inconnu_L1 */
        void ntItemListAddUnknownWord(NTItemList**,
                                      const unsigned int,
                                      const unsigned int,
                                      const Lexicon*,
                                      const Weight,
                                      const double);

        /* Former L1_est_suivant_de_L2 */
        gboolean ntItemDerivationFollowsPrefixItem(const NTItemDerivationList*,
                                                   const PrefixItemList*,
                                                   const Grammar*,
                                                   const TreeNode**);

        /* Former est_partie_droite */
        gboolean ntItemDerivationIsRightPart(const NTItemDerivationList*,
                                             const Grammar*,
                                             TreeNode**);

        /* Former Affiche_Element_L1 */
        int ntItemDerivationDisplay(const NTItemDerivationList*,
                                    const ParsingChart*,
                                    ParsingResult*);


        /* Former extrait_plus_probable_sous_arbre_L1 */
        void ntItemListExtractMostProbableSubtree(NTItemList*,
                                                  const ParsingChart*,
                                                  ParsingResult*);

        /* Former get_mot */
        void parsingChartBasisGetGraphy(const ParsingChartBasis*,
                                        const ParsingChart*,
                                        GString*);

        /* Former partiellement_lexical */
        void ntItemListAddLexicalizedItem (NTItemList**,
                                           const unsigned int,
                                           const unsigned int,
                                           const Grammar*,
                                           const TreeIndex,
                                           const Weight);

        /* Former Ajoute_Une_Cms_L1 */
        void ntItemListAddLexicalDerivation(NTItemList**,
                                            const unsigned int,
                                            const unsigned int,
                                            const LexicalEntryIndex,
                                            const int,
                                            const Weight,
                                            const Probability,
                                            const double);

        /* Former ajoute_chaine_lexicale_L1 */
        void ntItemListAddPOSFromWord(NTItemList**,
                                      const unsigned int,
                                      const unsigned int,
                                      const Lexicon*,
                                      const LexicalEntryIndex,
                                      const Weight,
                                      const double);

        /* Former Libere_Table_CYK */
        void parsingChartFree(ParsingChart**);

        /* Former parcours_iteratif_sous_arbre_L1 */
        void ntItemListExploreSubtree(const NTItemList*,
                                      const ParsingChart*,
                                      PointerStack*,
                                      CharStack*,
                                      Probability*,
                                      ParsingResult*);
```

```
/* Former Cree_Table_Cyk */
ParsingChart* parsingChartCreate(const size_t,
                                 const Grammar*,
                                 const char*,
                                 SentenceCut*);

/* Former Recupere_NT */
LexicalEntryIndex ntItemDerivationGetNT(const NTItemDerivationList*,
                                        const Grammar*);

/* Former Taille_Table_CYK */
size_t parsingChartGetMemorySize(const ParsingChart*);

/* Former Case_Contient_NT */
gboolean parsingChartCellContainsNT(const ParsingChart*,
                                    const Morpho,
                                    const unsigned int,
                                    const unsigned int);

/* Former Case_Contient_P */
gboolean parsingChartCellContainsP(const ParsingChart*,
                                   const unsigned int,
                                   const unsigned int);

/* Former Proba_Max_Element_L1 */
Probability ntItemDerivationListGetMaxProbability(const NTItemDerivationList*,
                                                  const Grammar*);

/* Former Proba_Somme_Element_L1 */
Probability ntItemDerivationListGetSumProbability(const NTItemDerivationList*,
                                                  const Grammar*);

/* Former Proba_Max_Element_L2 */
Probability prefixItemDerivationListGetMaxProbability
            (const PrefixItemDerivationList*);

/* Former Proba_Somme_Element_L2 */
Probability prefixItemDerivationListGetSumProbability
            (const PrefixItemDerivationList*);
```

List of internal-use functions that are not exported:

```
/* Former parcours_iteratif_sous_arbre_L2 */
static void prefixItemListExploreSubtree(const PrefixItemList*,
                                         const ParsingChart*,
                                         PointerStack*,
                                         CharStack*,
                                         Probability*,
                                         ParsingResult*)

/* Former taille_liste_L1 */
static void ntItemListFree(NTItemList**)

/* Former taille_liste_L2 */
static void prefixItemListFree(PrefixItemList**)

/* Former taille_liste_L1 */
static size_t ntItemListGetSize(NTItemList*)

/* Former taille_liste_L2 */
static size_t prefixItemListGetSize(PrefixItemList*)

/* Former extrait_plus_probable_sous_arbre_L2 */
static void prefixItemListExtractMostProbableSubtree(PrefixItemList*,
                                                     const ParsingChart*,
                                                     ParsingResult*)
```

List of functions that are moved in an other module:

```
/* Former affiche_correction moved in lexicalcorrection module */
void getCorrection(const Lexicon*,
                   const LexicalEntryIndex,
                   const Weight,
                   const gboolean,
                   GString*);
```

1.3.4. parser module

parser is a redesigned version of the former **cyk** module that supplies a syntactical parsing algorithm. **cyk** content has been splited into several smaller modules in such a way that **parser** only contains functions and declarations dedicated to parsing algorithm.

The major contribution in relation to the prior architecture is the introduction of the interpretation iterator. This data structure allows to output all syntactical derivations resulting from a parsing process in a more secure and convenient way. All required information about the considered solutions are stored in a single **InterpretationIterator** structure, so simultaneous extractions may be performed with no trouble.

Two parsing modes are available: **ALL_SOLUTIONS** looks for all valid interpretations, whereas **ONE_BEST** searches only the most likely one (but in a more efficient way). Thus, original architecture proposes peers of functions (one for each parsing mode) with the same purpose. To avoid systematic mode testing and to make the code more generic, such functions are now placed in an array (**parsingChartApplyParsing** & **parsingChartAutoFill**).

```
typedef enum {
    ALL_SOLUTIONS = 0,      /* Former ALLSOLUTIONS */
    ONE_BEST                /* Former ONEBEST */
} ParsingMode;              /* Former Type_Analyse */

typedef struct {
    ParsingChart*   chart;
    unsigned int    row;
    unsigned int    column;
    gboolean        only_top_nt;
    NTItemList*     exploration_list;
} InterpretationIterator; /* NEW */
```

List of exported functions:

```
/* Former Cyk_plus_plus & Cyk_plus_plus_1best */
const void (*parsingChartApplyParsing[ParsingMode])(ParsingChart*);

/* Former Autoremplissage & Autoremplissage_1best */
const void (*parsingChartAutoFill[ParsingMode])(ParsingChart*,
                                                const unsigned int,
                                                const unsigned int,
                                                const unsigned int);

/* Former Analyse_Syntaxique & Analyse_Syntaxique_Type */
ParsingChart* parse(const char*,
                    const Grammar*,
                    const ParsingChartInitConfig*,
                    const ParsingMode);

/* Former RaZ_Parcours_Cyk_Iteratif */
void interpretationIteratorGoToFirst(InterpretationIterator*);

/* Former Init_Cyk_For_MPP */
void interpretationIteratorGoToFirstMPP(InterpretationIterator*);

/* Former Extrait_Plus_Probable */
void parsingChartGetMostProbableInterpretation(const ParsingChart*,
                                               const unsigned int,
                                               const unsigned int,
                                               const gboolean,
                                               ParsingResult*);

/* Former equal */
gboolean equal(const Probability, const Probability, const double);

/* Former Parcours_Cyk_Iteratif */
gboolean interpretationIteratorGetNext(InterpretationIterator*,
                                       ParsingResult*);

/* Former Print_MPP_Iteratively */
gboolean interpretationIteratorGetNextMPP(InterpretationIterator*,
                                          ParsingResult*);
```

List of brand new functions:

```
/* InterpretationIterator structure constructor */
InterpretationIterator* interpretationIteratorCreate(const ParsingChart*,
                                                     const unsigned int,
                                                     const unsigned int,
                                                     const gboolean);

/* InterpretationIterator structure destructor */
void interpretationIteratorFree(InterpretationIterator**);
```

List of internal-use functions that are not exported:

`ajout_feuille_L1` – renamed to `ntItemListAddLeaf_ALL_SOLUTIONS`.

`ajout_1best_feuille_L1` – renamed to `ntItemListAddLeaf_ONE_BEST`.

`ajout_un_element_L1` – renamed to `ntItemListAddDerivation_ALL_SOLUTIONS`.

`ajout_1best_un_element_L1` – renamed to `ntItemListAddDerivation_ONE_BEST`.

`ajout_un_element_L2` – renamed to `prefixItemListAddElement_ALL_SOLUTIONS`.

`ajout_1best_un_element_L2` – renamed to `prefixItemListAddElement_ONE_BEST`.

`Autoremplissage` – renamed to `parsingChartAutoFill_ALL_SOLUTIONS`.

`Autoremplissage_1best` – renamed to `parsingChartAutoFill_ONE_BEST`.

`Cyk_plus_plus` – renamed to `parsingChartApplyParsing_ALL_SOLUTIONS`.

`init_cyk_plus_plus` – renamed to `initializeParsing`.

`Ordonne_L1` – renamed to `ntItemListSort`.

`Ordonne_L1_pour_autoremplissage` – renamed to `ntItemListSortForAutoFill`.

`Ordonne_L1_recursif` – renamed to `ntItemListSort_core`.

List of functions transferred to parsingchart module:

`extrait_plus_probable_sous_arbre_L2` – renamed to `prefixItemListExtractMostProbableSubtree`.

`parcours_iteratif_sous_arbre_L2` – renamed to `prefixItemListExploreSubtree`.

`taille_liste_*` - renamed to `*ItemListGetSize`.

List of functions transferred to parsingchartinit module:

`ajoute_un_lexemme` – renamed to `solutionSetAppendLexeme`.

`corrige_et_ajoute` – renamed to `correctAndAppendLexeme`.

`emplit_table_cyk` – renamed to `parsingChartInit`.

`mot_inconnu` – renamed to `solutionSetAppendUnknowWord`.

List of functions transferred to chartdisplay module:

`affiche_analyses_case` – renamed to `parsingChartCellDisplayAnalysis`.

`affiche_sous_arbre_L1` – renamed to `ntItemListDisplaySubtree`.

`affiche_sous_arbre_L2` – renamed to `prefixItemListDisplaySubtree`.

`print_coord_el1` – renamed to `ntItemPrintCoordinates`.

`print_coord_el2` – renamed to `prefixItemPrintCoordinates`.

List of useless functions that are deleted:

`Affiche_NTPG` – Used only once, so its code can be directly copied.

`Libere_Resultat_lexicalise` – `Retour_lexicalise` datatype has disappeared, so this function is nomore necessary.

1.3.5. parsingchartinit module

parsingchartinit is a redesigned version of a subset of former **cyk** module. It is dedicated to initialize a parsing chart with a particular sentence to analyze. This chart is then treated by the **parser** module.

```
typedef struct {
    ParsingChart*     (*init)(const char*, Grammar*, Weight,
                              int(*)(int), int(*)(int),
                              int(*)(int), int(*)(int)); /* Former call */

    int       (*is_delimiter)(int);          /* Former strict_sep */
    int           (*is_space)(int);          /* Former space */
    int (*is_never_delimiter)(int);          /* Former homogene */
    int          (*is_sticky)(int);          /* Former collable */
    Weight max_correction_cost;              /* Former max_cost */
} ParsingChartInitConfig;                    /* Former GetInput_Function_Type */


/* Former DEFAULT_MAXIMUM_COST */
#define DEFAULT_CORRECTION_COST ((Weight)1.5)


/* Former Default_GetInput_Function */
const ParsingChartInitConfig DEFAULT_PARSINGCHART_INIT_CONFIG = {
    lexicalizeToChart,
    defaultIsDelimiter,
    isspace,
    defaultIsNeverDelimiter,
    NULL,
    DEFAULT_CORRECTION_COST
};
```

List of exported functions:

```
/* Former Lexicalise */
ParsingChart* lexicalizeToChart(const char*, Grammar*, const Weight,
                                int (*)(int), int (*)(int),
                                int (*)(int), int (*)(int));

/* Former Decoupe_Simple */
ParsingChart* tokenizeToChart(const char*, Grammar*, const Weight,
                              int (*)(int), int (*)(int),
                              int (*)(int), int (*)(int));
```

List of internal-use functions that are not exported:

```
/* Former augmente_decoup */
static void sentenceCutEnlarge(SentenceCut**, size_t*, size_t*);

/* Former augmente_ens_sol */
static void solutionSetEnlarge(SolutionSet**, size_t*, size_t*);

/* Former ajoute_un_lexemme */
static void solutionSetAppendLexeme(size_t*, size_t*, SolutionSet**,
                                    size_t*, size_t*, SentenceCut**,
                                    const LexicalEntryIndex,
                                    const unsigned short,
                                    LexicalAccessTable*);

/* Former mot_inconnu */
static void solutionSetAppendUnknowWord(size_t*, size_t*, SolutionSet**,
                                        size_t*, size_t*, SentenceCut**,
                                        const unsigned short, const char*);

/* Former corrige_et_ajoute */
static void correctAndAppendLexeme(const char*, LexicalAccessTable*,
                                   const Weight,
                                   size_t*, size_t*, SolutionSet**,
                                   size_t*, size_t*, SentenceCut**,
                                   unsigned short*, const unsigned short);

/* Former emplit_table_cyk */
static void parsingChartInit(ParsingChart*, const size_t, const SolutionSet*);
```

1.3.6. chartdisplay module

`chartdisplay` is a redesigned version of a subset of former `cyk` module. It gathers functions dedicated to print or dump parsing chart under various format.

List of exported functions:

```
/* Former Affiche_Resultat_Analyse_Syntaxique & Affiche_Arbre_Table_Cyk */
void parsingChartDisplayParsing(const ParsingChart*,
                                int (*)(const char*, ... ), const gboolean);

/* Former Affiche_Arbre_Case_Cyk */
void parsingChartDisplayForestFromElement(const ParsingChart*,
                                          const unsigned int,
                                          const unsigned int,
                                          int (*)(const char*, ... ),
                                          const gboolean);

/* Former Dump_CNF */
void parsingChartDumpCNF(const ParsingChart*, int (*)(const char*, ... ));

/* Former Dump_CFG */
void parsingChartDumpCFG(const ParsingChart*, int (*)(const char*, ... ));

/* Former Dump_nb_L1_L2 */
void parsingChartDumpNbItems(const ParsingChart*, int (*)(const char*, ... ));

/* Former Affiche_Stat_Case_Cyk */
void parsingChartDisplayCellStat(const ParsingChart*,
                                 const unsigned int, const unsigned int,
                                 int (*)(const char*, ... ), const gboolean);

/* Former Affiche_Couverture_Cyk */
void parsingChartDisplayCoverage(const ParsingChart*,
                                 int (*)(const char*, ...), const gboolean);

/* Former Dump_Table_CYK */
void parsingChartDump(const ParsingChart*,
                      int (*)(const char*, ... ),
                      const gboolean);

/* Former affiche_correction_case_cyk (Christophe de Benoit’s project) */
void parsingChartDumpSpellCorrection(const ParsingChart*);
```

List of internal-use functions that are not exported:

```
/* Former print_coord_el2 */
void prefixItemPrintCoordinates(const ParsingChart*,
                                const unsigned int, const unsigned int,
                                const PrefixItemList*,
                                int (*)(const char*, ... ));

/* Former print_coord_el1 */
void ntItemPrintCoordinates(const ParsingChart*,
                            const unsigned int, const unsigned int,
                            const NTItemList*,
                            int (*)(const char*, ... ));

/* Former affiche_sous_arbre_L1 */
void ntItemListDisplaySubtree(NTItemList*,
                              const ParsingChart*,
                              int (*)(const char*, ... ),
                              ParsingResult*);

/* Former affiche_sous_arbre_L2 */
void prefixItemListDisplaySubtree(PrefixItemList*,
                                  const ParsingChart*,
                                  int (*)(const char*, ... ),
                                  ParsingResult*);

/* Former affiche_analyses_case */
void parsingChartCellDisplayAnalysis(ParsingChart*,
                                     const unsigned int, const unsigned int);

/* Former affiche_correction_case_cyk (Christophe de Benoit’s project) */
void parsingChartCellDumpCorrection(const ParsingChart*,
                                    const unsigned int, const unsigned int);
```

1.3.7. latticeparser module

`latticeparser` is a redesigned version of the former `parselattice` module that allows to perform speech lattice parsing. Except the renaming task of declared structures and functions, the only relevant contribution in relation to the prior *API* is the introduction of the `parsingLatticeSave` function.

```
typedef struct {
    unsigned int start;       /* Former begin */
    unsigned int end;         /* Former end */
    Probability probability;  /* Former prob */
    char* label;              /* Former label */
} ParsingLatticeLink;         /* Former Arc */


typedef struct {
    size_t number_of_nodes;   /* Former number_of_nodes */
    PointerArray* links;      /* Former arcs */
    IntArray* silence_links;  /* Former silences */
    IntArray* deleted_nodes;  /* Former deleted_nodes */
} ParsingLattice;             /* Former Lattice */


/* Former Accede_Arc */
#define GET_LINK(lattice,index)       …
```

List of exported functions:

```
/* Former Libere_Lattice */
void parsingLatticeFree(ParsingLattice**);

/* Former DeleteSilences */
void parsingLatticeDeleteSilences(ParsingLattice*);

/* Former DeleteUnreachedNodes */
void parsingLatticeDeleteUnreachedNodes(ParsingLattice*);

/* Former Lattice_to_Chart */
ParsingChart* parsingLatticeToChart(const Grammar*,
                                    const ParsingLattice*,
                                    const gboolean);

/* Former Init_Lattice */
ParsingLattice* parsingLatticeCreate();

/* Former ReadLatticeFile */
ParsingLattice* parsingLatticeLoad(const char*,
                                   const double);

/* Former Analyse_Syntaxique_Lattice & Analyse_Syntaxique_Lattice_Type */
ParsingChart* parsingLatticeParse(const ParsingLattice*,
                                  const Grammar*,
                                  const gboolean,
                                  const ParsingMode);
```

Brand new function:

```
/* Reciprocal to parsingLatticeLoad */
void parsingLatticeSave(const ParsingLattice*, const char*);
```

1.4 File architecture

The intended file architectures looks roughly like any other *open source* project distribution. The *HTML* library documentation can be found in the `doc` sub-directory. Notice that all command line softwares of the `utils` sub-directory have been renamed to english in a more convenient and consistent manner.

Here is the file architecture of the development oriented distribution that is typically used by developers who want to contribute to the evolution of the *SlpTK* (former *SlpToolKit*) library. Although this work has **not** been adapted to the new version of the library, the *Java* interface layer has been added to this particular distribution. The related files may be found in the `javaInterface` sub-directory:

slptk/	ChangeLog			Modifications logfile
	COPYING			Licence
	INSTALL			Installation instructions
	MANIFEST.MF			Manifesto file
	README			Introduction file
	TODO			List of "to do" tasks file
	Makefile			-
	doc/			Tool documentation
	javaInterface/	src/	java/SlpToolKit/*.java	Java source files
			C/*.c;*.h	C source/header files
		classes/	*.class	class files
		doc/		Java interface documentation
		README		Introduction file
		ChangeLog		Modifications logfile
		INSTALL		Installation instructions file
		TODO		List of "to do" tasks file
	bin/	parse (former anagram)	lexematize (former lexematise)	
		automaton2re (former autoToRE)	listfsa (former listautomate)	
		checkword (former checklex)	listgram (former listcompiledgram)	
		compilegram (former compilgram)	listlex(former listlexique)	
		convertgram (former tradgram)	listtrie (former listarbrelex)	
		spellcorrect (former correction)	listtriestats (former statarbrelex)	
		createfsa (former creeautomate)	normalizelex (former normprob)	
		createlex (former creelex)	transducerapply (former binary_op_test)	
		flex2fsa (former flexExport)	transduct (former transduct_test)	
		lemmatize (former lemmatise)	unionfsa (former unionautomate)	
	lib/	libslptk.so/dll		The library file
	src/	main/*.h;*.c		Library header & source files
		test/*.c		Source files of test programs
		utils/*.c		Source files of utils
		todo/*.*		Not integrated sources & scripts

Here is the file architecture of the standard install distribution that is typically used by any developer who wants take advantage of *SlpTK* (former *SlpToolKit*) library without contributing to its developpement. It's basically a smallest version of the previous file architecture:

`slptk/`	**`ChangeLog`**	Modifications logfile
	`COPYING`	Licence
	`MANIFEST.MF`	Manifesto file
	`README`	Introduction file
	`doc/`	Identical to **`slpTk/doc`** directory from developpement file architecture
	`lib/`	Identical to **`slpTk/lib`** directory from developpement file architecture
	`bin/`	Identical to **`slpTk/utils`** directory from developpement file architecture
	`include/`	Identical to **`slpTk/src/core/*.h`** directory from developpement file architecture

1.5 Backward compatibility

Altering the library *API* reveals the problem of adapting the software based on the previous architecture. Although this trouble will disappear sooner or latter, several solutions are, however, conceivable for transitory needs:

- to continue using the former version of the library. Of course, it is the simplest alternative that doesn't allow taking advantage of any update or evolution of the new library version.
- to provide some interface modules between the former and the new *API*. This solution has two significant drawbacks. First of all, some functions of the new architecture don't have a direct equivalent in the former one. Moreover, implementation of such modules seems to be an overwhelming task according to their utility and use span.
- to name for every function of the new architecture the equivalent (or closest) original function in the documentation. Since the search for equivalence was already done during the design process, it is probably the most acceptable solution.